Frequent Update and Efficient Retrieval: an Oxymoron on Moving Object Indexes?

Beng Chin Ooi Kian Lee Tan
Department of Computer Science
National University of Singapore
Singapore 117543
{ooibc,tankl}@comp.nus.edu.sg

Cui Yu
Department of Computer Science
Monmouth University
New Jersey 07764, USA
cyu@monmouth.edu

Abstract

Traditionally, indexes have been designed to facilitate fast retrieval of static objects. Moreover, updates are assumed to be infrequent and hence slow update speed can be tolerated. However, this assumption does not hold for new applications fueled by the advancement of GPS, wireless technologies and small but powerful digital devices. In these applications, objects are mobile and to track these objects their locations have to be updated frequently. Existing indexes no longer can keep up with the high update rates while providing speedy retrieval at the same time. There is a need for novel indexes to be designed for moving objects. In this paper, we examine various design issues that need to be addressed to support efficient retrieval of moving objects with frequent updates.

1 Introduction

With improved accuracy in positioning technologies such as GPS (global positioning system) and sophisticated technologies like radars and other communication equipment, it is possible to continuously track moving objects, be it in the digital battle field or real world. Quick access to locations of mobile units enables more efficient deployment of resources and dissemination of information based on locations. In the commercial front, demand has surged for applications that track the locations of moving objects like vehicles, users of wireless devices and even deliveries. By the time the mobile network bandwidth exceeds 2/2.5G, we expect to see hundreds of millions of mobile users worldwide, with their connections almost always on. The main reasons behind tracking moving objects are to improve the quality of service and efficiency in resource management. For example, advertisements can now be pushed to mobile phones based on the proximity of users, and taxis can be dispatched quickly to passengers based on their locations and taxis in the vicinity. In order to provide such services and to facilitate dynamic queries, we need an efficient and accurate way of managing the latest positions of moving objects.

In location-aware mobile services, moving objects such as consumers with WAP-enabled mobile phone terminals and personal digital assistants (PDA), and vehicles with navigation and communication equipment, disclose their positions. The accuracy is dependent on such disclosure; otherwise, current locations are predicted based on some function of time and velocity of the moving objects or some other conditions such as no-movement of objects. Each disclosure will cause an update in the moving object databases. Due to the size of the database, indexes are required to facilitate fast location of objects based on spatial locations and facilitate retrieval of objects within certain distance from the query or current moving objects. The index must be adaptive and dynamic in order to efficiently accommodate movement of objects in the index structure and yet provide the expected answer as one would obtain by searching the database using a sequential scan method.

Moving objects pose new challenges to database systems. The conventional assumptions that objects are fairly static and their values are not frequently updated are being invalidated by the need to capture continuous movement. Frequent updates not only pose the usual contention problems on hot spots such as system catalog, they require the underlying indexes to be frequent-update aware and efficient. To reduce the number of updates on the indexes, strategies such as expressing the objects' positions as function of times, and delaying of updates have been employed. However, to better reflect dynamic changes in objects' positions, existing indexes have to be re-designed to enable fast updates and new concurrency control mechanisms are also required. In this paper, we shall review the requirement for such applications, and discuss various techniques

that could be used to provide both fast updates and retrieval. Frequent update is an additional requirement imposed by moving objects, and fast update is necessary so that the overall throughput of the index, querying and updating, is kept high.

The paper is organized as follows. In the next section, we introduce the representation of moving object positions and describe the queries posed on systems supporting moving objects. In Section 3, we have a quick overview on the problem of indexing moving objects. We then present various feasible methods in reducing the number of updates and speeding up updates while providing fast retrieval in Section 4. We conclude in Section 5.

2 On Representation of Moving Object Positions and Queries

Let us consider objects that are moving in d-dimensional space. The most straightforward way of handling moving objects is to treat them as if they are static objects and store only their currently known positions. To ensure that the stored positions are up-to-date, moving objects must frequently update their positions. While update overhead is high, such an approach has two advantages: (1) it allows one to reuse existing multi-dimensional indexes [3] and query processing strategies for answering range/window, proximity and nearest neighbor queries [25]; (2) it reduces the problem of managing positions of moving objects to a frequent update problem.

An alternative approach is to model the position of an object as a function of time, i.e., an object's position at time t given by $\overline{\mathrm{x}}(t) = (x_1(t), x_2(t), ..., x_d(t))$, is modeled as $\bar{x}(t) = \bar{x}(t_r) + \bar{v}(t - t_r)$, where $\bar{v}$ is the current velocity of the object, $\bar{v} = (v_1, v_2, ..., v_d)$, t_r is the time when the position was last recorded, and t may be larger than the current time (*now*). In this way, an object only needs to update its position (i.e., new position and velocity vector) whenever there is a change in its direction or/and speed. This method not only reduces the number of updates (compared to the simple method), it also allows the current and anticipated future positions of a point to be described by $2d$ parameters - d for the spatial dimensions and d for the velocity vectors. As such, more complex queries involving time and future positions can be supported.

Before we look at the design issues to support moving objects, it is worth noting that both representations essentially provide only approximate answers to queries. For the first method, it is possible that an answer may no longer satisfy a query (if the query arrives before its new position is updated); similarly, it is also possible for an answer to be missed. On the other hand, the second method relies on the accuracy of the functions used. It is therefore possible to have both false positives and negatives as well (especially for queries that involve future positions). As the second model has received much attention in the literature [11], we shall restrict the rest of our discussion to this model.

There are essentially three ways in which we can represent moving objects.

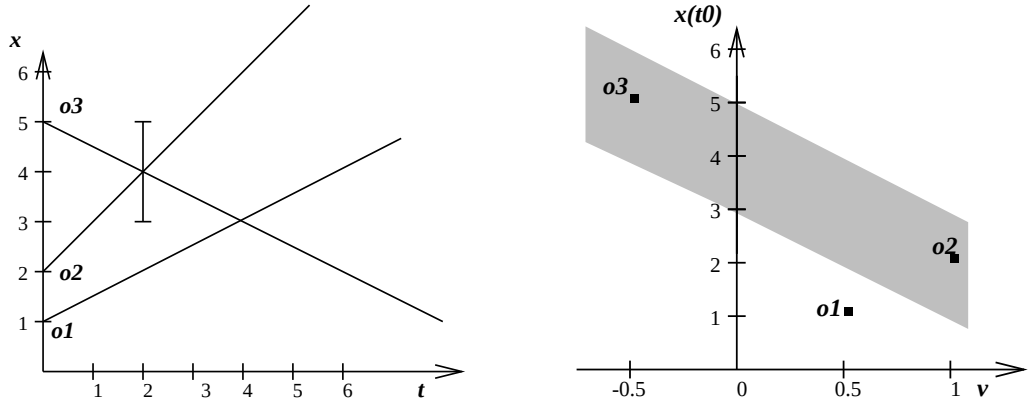
- As lines in (d+1)-dimensional space – d spatial dimensions and 1 time dimension. Figure 1(a) shows an example of three moving objects in 1-d space.
- As points in $2d$-dimensional space – d spatial and d velocity dimensions (function parameters: $\bar{x}(t_0)$, $\bar{v}$) Figure 1(b) shows the three moving objects in Figure 1(a) when represented in $2d$ space.
- As time-parameterized points in d-dimensional space.

A moving object database should be able to answer queries based on the current, past or future positions of the objects. Like traditional static multi-dimensional databases, there are three types of queries, namely *range queries*, *proximity queries* and *k-nearest neighbor (kNN) queries*. Now we shall define each of these types of queries:

- *range queries:* "find all objects whose positions fall within certain given ranges from time t_1 to t_2". Range queries can be formally expressed with respect to the database DB as follows:

  $$\{o \in DB \mid o(t) \in queryW\}$$

  where $queryW$ is a rectangular window range query and $t_1 \leq t \leq t_2$. Note that if $queryW$ is denoted as $[l_i, h_i]$ ($0 \leq i < d$) and the position of an object o at time t, $o(t)$, is represented as $\{x_i(t)\}$ ($0 \leq i < d$), ($o(t) \in queryW$) can be as simple as $[l_i \leq x_i(t) \leq h_i]$ where $0 \leq i < d$. Range query here is a rectangular window range query, which is however often called as window range query or window query. We note that if the time is not specified, then the query refers to the current time. Referring to the example in Figure 1, we see that for the first representation (Figure 1(a)), objects o_2 and o_3 will be returned as answers for the range query that looks for objects within [3,5] at time 2. For the second representation, the same query has to be transformed into the shaded region in Figure 1(b). The third representation will require identifying all new positions of the points and check if they fall in the query region (as we shall see shortly, there are more efficient ways to perform this).
- *proximity queries:* "find all objects in the database which are within a given distance ϵ from a given object from t_1 to t_2". This query is commonly known as



(a) d+1 dimensional space.

(b) 2d dimensional space.

Figure 1. Data representation

similarity range query in multimedia context. It can be formally expressed as follows:

$$\{o \in DB \mid dist(q(t), o(t)) \leq \epsilon\}$$

where q is the given query object, $dist$ is the distance applied, and $t_1 \leq t \leq t_2$. We note that $dist$ is highly application-dependent, and may have different metrics.

- *Nearest neighbor (NN) queries:* "find an object in the database which are closest in distance to a given object from t_1 to t_2". Let NN_j denote sets of moving objects and T_j denote a time interval. The NN query returns the set $\{(NN_j, T_j)\}$ such that $\cup_j T_j = [t_1, t_2]$ and $i \neq j \Rightarrow T_i \cap T_j = \emptyset$. In addition, each point in NN_j is a nearest neighbor to q during all of interval T_j. That is,
  $\forall j \; \forall o \in NN_j \; \forall r \in DB \backslash o(dist(o, q) \leq dist(r, q))$

More recently, there are several more complex queries that have been proposed: reverse NN queries [2], and queries involving moving query ranges.

3 Indexing Moving Objects

Recent advances in hardware technology have reduced the access times of both memory and disk tremendously. However, the ratio between the two still remains at about 4 to 5 orders of magnitude. Hence, optimizing the number of disk I/Os remains important. Further, the indexes are often used as the filtering step in reducing the number of objects that need to be evaluated in main memory, and hence reducing the computational cost. This calls for the use of organizational methods or structures known as indexes to locate

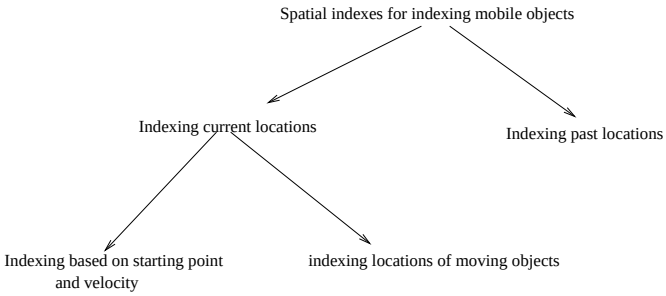
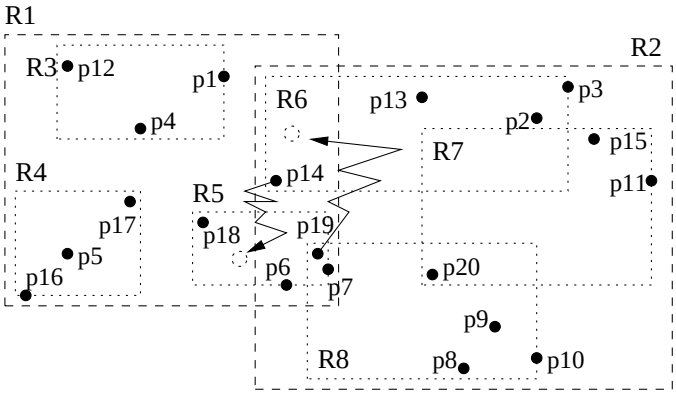


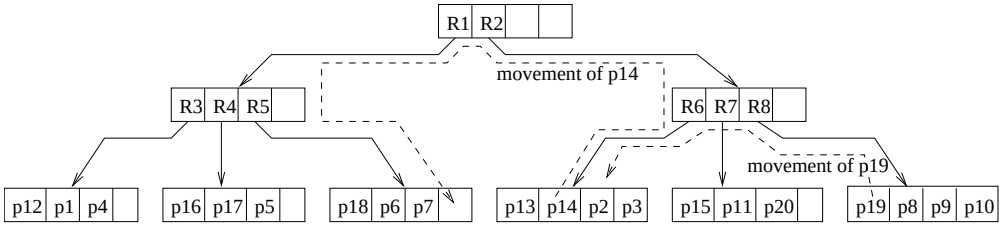
Figure 2. Moving object indexing

data of interest quickly and efficiently. Many indexes have been proposed for multi-dimensional databases and, in particular, for spatial databases. However, these indexes have been designed mainly to speed up the retrieval, in applications where queries are relatively much more frequent than updates. This is being invalidated by new location-based applications in which the number of spatial objects are typically points that are movable. These moving objects pose new challenges to spatial data management and the design of indexes. The workload in such a system is characterized by high index update loads and frequent queries.

Figure 2 shows the various directions in indexing mobile objects and some of the proposals made. Indexing mobile objects can be divided into 2 main categories: indexing current locations of spatial objects and indexing historical locations of spatial objects. The first category can be further divided into two sub-categories: (1) Using functions to approximate movement and (2) indexing locations of moving objects.



a. The planar representation for an R-tree



b. The directory of an R-tree

Figure 3. Updates of nodes due to moving objects

3.1 The R-tree as the Base Index

In the last two decades, many multi-dimensional index structures have been proposed [25]. Among them, the R-trees are the most popular which is a hierarchical, height-balanced index structure. The R-trees have leaf nodes and internal nodes with entries in leaf node pointing to objects and entries in internal nodes pointing to other internal or leaf nodes (see Figure 3). The information contained in an R-tree is thus hierarchically organized and every level in the tree provides more detail than its ancestor level. The search process in an R-tree is very different from that in a B-tree due to the lack of ordering and the possible overlap among keys. To find all rectangles intersecting a given range the search process has to descend all subtrees that intersect or fully contain the range specification. When a new key has to be inserted in an R-tree, one attempts to descend to the geometrically optimal leaf by picking at each level the subtree with the optimal bounding rectangle. In contrast to the B-trees, the R-trees have to recursively update the ancestor MBRs if a leaf's MBR changes. Splitting a node also deviates noticeably from the B-tree pattern. The R-tree will partition the key sequence according to its layout strategy and it is impossible to completely avoid any overlap after split.

If we simply apply the R-tree technique to index the locations of moving objects, readjusting the entire index is inevitable. For example, the movement of $p19$ in Figure 3 from one leaf node to another leaf node will cause the split of the destination leaf node. That is, a split may cause a split of a leaf node which may propagate all the way up to the root. Likewise, it may cause an underflow in a source leaf node, and re-insertions may be necessary. Even if there are no node splitting and merging, frequent movement of objects can retard the performance of the index, as the nodes have to locked when the nodes are accessed and MBRs adjusted. Take object $p14$ for example, its short movement will cause the nodes and MBRs along two paths to be adjusted. Such adjustments are expensive in view of the large numbers of updates that are continuously issued. As a result, the original R-tree technique is not directly suitable for such purposes. However, due to its robustness in handling spatial objects, the R-tree and its variants provide a good basis for extension for supporting moving objects. We shall use the R-tree as the basis for discussion; the ideas are however suitable for most hierarchical indexes.

3.2 Indexing historical movement

The historical movement of objects can be represented by their trajectories, i.e., line segments of their positions over time. Figure 4 shows an example. Clearly, an R-tree can be used to index the trajectories. While a single MBR suffices to bound the entire trajectory of an object, it is more efficient to split the trajectories into segments. However, because the end points are not bounded, and large MBRs contain dead space as well as lead to significant overlaps, R-tree has been shown to be very inefficient.

Several methods have been proposed to index historical locations of spatial objects [10, 19, 9]. Specifications and framework for efficient indexing in spatio- temporal databases can be found in [24] and [9] respectively. In particular, the work in [19] proposes a B-tree based scheme, TB-tree, that strictly preserves trajectories, i.e., leaf nodes in the index contain segments belonging to one trajectory. This can improve the performance of trajectory-based queries that require segments of the same trajectories to be retrieved.

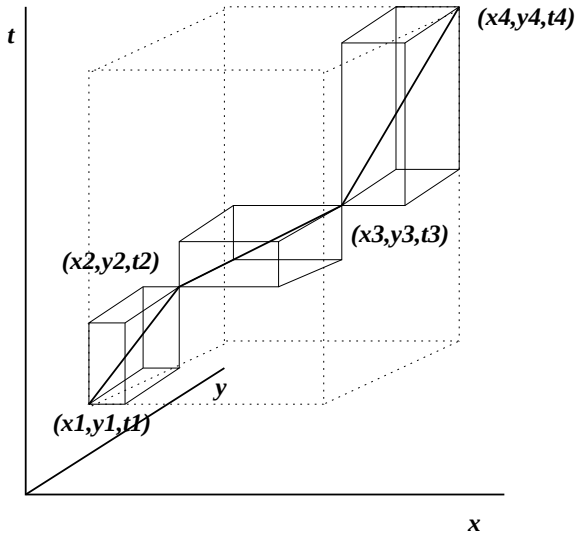


Figure 4. Approximating trajectories using MBRs

3.3 Indexing current and future movement

More recent works have focused on indexing techniques that can facilitate queries on future positions of points [7, 21]. Chon [7] stores the projected trajectories of the positions of points and stores them using a regular grid structure. While this speeds up query processing, it requires duplicating an object across all grids that it intersects. As a result, any update becomes costly.

The time-parametric tree (TPR-tree) [21] is an R-tree based index where the location of a moving point is represented by a reference position and a corresponding velocity vector. In TPR-tree, the coordinates of the bounding rectangles are functions of time. Thus, they are capable of following the objects as they move, and updates are kept to a minimum. Figure 5 illustrates the time-parametrized rectangles in TPR tree. We note that the velocities associated with each edge corresponds to the maximum velocity of points in that direction. The TPR-tree considers the locations of moving points as well as their velocity vector when splitting nodes. In particular, since the bounding rectangles may become very large leading to significant overlaps, there is a need to "tighten" them during updates or search.

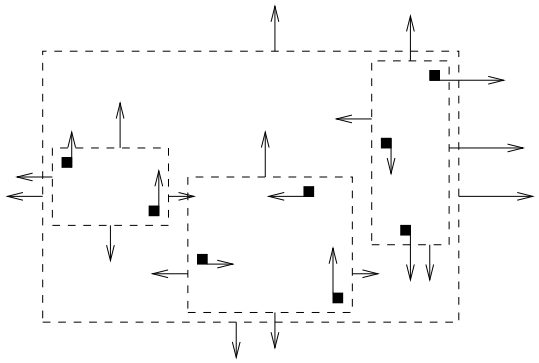


Figure 5. Time-parametric rectangles in a TPR-tree

Kollios et al. [13] an indexing scheme based on partition trees, in which a linear function is used to represent the movement of objects. A line is mapped into a point in the dual plane which facilitates indexing in spatial databases. However, this duality transformation causes a typical rectangle query to become a polygon, making queries difficult to execute. Agarwal et al. [1] also proposed various efficient indexing schemes based on duality and with ways of answering approximate KNN queries.

All these proposals could not substantially reduce update cost in reality because of the difficulty in finding a sophisticated and robust way of representing moving objects in the real dynamic world. Besides, they may not yield better query performance than the conventional R-tree for current locations.

4 Handling of Fast Updates

Existing database systems have not been designed to handle continuously changing data such as location of moving spatial objects. In order to represent moving objects in

the database, the location of moving objects must be continuously updated. Frequent updates incur both system performance and wireless communication overhead. Hence, locations are represented as function of time to reduce update cost and the imprecise problem caused by the fact that location updates are usually initiated by moving objects themselves. Nevertheless, comparatively, the frequency of updates is still much higher than what we experience in conventional databases. In this section, we outline various strategies that either have been proposed or could be used to alleviate frequent update problems.

4.1 Concurrency Controls

Several concurrency control algorithms have been proposed to support concurrent operation on multi-dimensional index structures. Similar to those of the B-trees, they can be categorized into lock coupling and linking algorithms. The techniques of lock coupling, breadth-first search, lock and scope were supports concurrency control algorithms on the original R-tree structure [6, 18]. They only release the lock on the current node when the lock on the next node to traverse is granted for query operations. For update operations, multiple locks need to be hold simultaneously when node split and MBR change occur, which significantly deteriorates the throughput of the concurrency control algorithm. To solve lock-coupling problem, the linking algorithms were proposed in [14, 15, 12, 22]. These methods lock one node most of the time for search operation and only employs lock-coupling when splitting the node or propagating the MBR change.

The radically different linking approach was originally for B-trees [17]. Instead of avoiding possible conflicts by lock-coupling, the tree structure is modified so that the search process has the opportunity to compensate for a missed split. The crucial addition is the rightlink, a pointer going from every node to its right sibling on the same level. When a node is split and a new right sibling is created, it is inserted into the rightlink chain directly to the right of the old one. The effect is that all nodes at the same level are chained together through the rightlinks. Searching in a B-link tree can therefore be done without lock-coupling. When descending to a node that was split after examining the parent, the search process discovers that the highest key on that node is lower than the key it is looking for and correctly concludes that a split must have taken place. For an insertion process, if the leaf has to be split, it can also avoid lock-coupling when installing a new entry in the parent [20]. As soon as the page has been split and the new right sibling inserted into the rightlink chain, the insertion process can drop the lock on the leaf that was overflowing and then acquire a lock on the parent, possibly moving right to compensate for concurrent splits and splitting up the tree recursively. This linking strategy offers very high concurrency because search and insertion processes only need to lock one node at a time.

The main obstacle to the use of linking mechanism is the lack of linear ordering among the keys in the multi-dimensional index structures. To overcome this problem, the R-Link trees [14] was proposed to provide high concurrency operations on the R-trees through a rightlink-style approach. They assign logical sequence numbers (LSNs) to each node and entry, which are similar to timestamps in that they monotonically increase over time but are not synchronous with any real-time clock. The node entries and the search and insert algorithms are designed so that these LSNs can be used to make correct decisions about how to move through the tree. An R-Link tree is basically a standard R-tree with two key differences. First, all of the nodes on any given level are chained together in a singly-linked list via rightlinks. Second, the main structural addition is an LSN in each node that is unique with the tree. These LSNs give us a mechanism for determining when an operation's understanding of a given node is obsolete. Each entry in a node consists of a key rectangle, a pointer to the child node and the LSN that it expects the child node to have. If a node has to be split, the new right sibling is assigned the old node's LSN and the old node receives a new LSN. A process traversing the tree can detect the split even if it has not been installed in the parent by comparing the expected LSN, as taken from the entry in the parent node, with the actual LSN.

In the above algorithm, each entry of internal nodes has extra information to keep the LSNs of child nodes and reduces the storage utilization, which may degrade the query performance. Consequently, an extension for concurrency control was proposed to deal with the extra information problem, called Concurrent GiST (CGiST) [15]. CGiST extends every node with a node sequence number (NSN) and a rightlink and uses these to detect splits. The NSN is taken from a tree-global, monotonically increasing counter variable. During a node split, this counter is incremented and its new value assigned to the original node; the new sibling node receives the original node's prior NSN and rightlink. In general, a traversing operation can now detect a split by memorizing the global counter value when reading the parent entry and comparing it with the NSN of the current node. If the latter is higher, the node must have been split and the operation follows rightlinks until it sees a node with an equal or smaller NSN. When we split a node, we must lock its parent node, split the node, set NSN and increase the tree-global counter. So multiple locks must be hold for split operations, which may delay the concurrent queries. It improves on the R-link tree design by eliminating the space overhead in internal index entries, but this overhead is negligible. For a 4K bytes pages, the fanout of the R-tree can be

more than 160 (2D objects), thus the space overhead is only around 1%. On the contrary, to make CGiST work properly, it must lock the parent before the node split, which reduces the degree of concurrency significantly.

So far, we can see that the linking techniques also need to request multiple locks exclusively for split and MBR change. Some mechanisms were proposed to improve the concurrency based on them. In [12], the authors proposed a new linking scheme, which employed a new approach for MBR modification, called top-down index region modification (TDIM). This scheme performs MBR modification from top down by operating on at most one node along the insertion path for most insertions. TDIM combines MBR modification with tree traversal and avoids locking of nodes from multiple levels of the tree at the same time. Also the MBR modification is done in a piecemeal fashion without excluding query access, queries are not blocked except during node split. Additionally, they proposed a split algorithm, named copy based concurrent update (CCU). The basic idea of CCU is to split the node in a local copy. Queries are free to access the original node while the split is processing. The content of the original node is changed after the split in local copy completes. Thus queries only block during the copy back. The main disadvantage of TDIM is that it does not supports the delete operation. To locate the delete object, we need to traverse multiple path to get the object and we do not sure which node we access is the ancestor of target object. Additionally, even we know the ancestor, we still can not decide whether we can shrink the certain MBR. For CCU, extra spaces are needed for split and each split incurs garbage node, which increases the complexity of the algorithms.

In [22], the authors proposed a concurrency control method to minimize the query delay. To avoid the query delay by MBR updates, they introduced partial lock coupling (PLC) technique. The PLC technique increases concurrency by using lock coupling only in case of MBR shrinking operations that are less frequent than MBR expansion operation. To reduce the query delay by split operation, they optimize exclusive latching time on a split node. The weakness of PLC is that the x-lock is hold during the propagation, and the algorithm did not provide phantom protection. Additionally, this algorithm is based on CGiST, and it must lock the parent of split node before the split; hence multiple locks need to be hold.

Concurrent access to data through a multi-dimensional indexes introduces the problem of protecting query range from phantom update. The CGiST method [15] uses a modified predicate locking mechanism to provide phantom protection over Generalized Search Trees. In [4], the dynamic granular locking (DGL) approach was proposed to phantom protection in R-trees. DGL method dynamically partitions the embedded space into lockable granules that adapt to the distribution of objects. They define the lowest level BRs of the R-tree as the lockable granules. Since the R-tree partitions may not cover the entire embedded space, they present an additional structure that partitions the non-covered space into a set of granules referred to as external granules. Following the principles of granular locking, each operation requests locks on enough granules to guarantee that any two conflicting operations request conflicting locks on at least one granule in common. They also proposed two locking strategies, the "*cover-for-insert and overlap-for-search* policy and the "*overlap-for-insert and cover-for-search*" policy. The DGL approach addressed phantom protection problem in multi-dimensional access methods and granular locks can be implemented more efficiently compared to predicate locks, but DGL may offer lower degree of concurrency because of its complexity. Additionally, DGL must integrate with other methods to form the complete concurrency control algorithm for multi-dimensional access.

Elsewhere [8], we introduce a new concurrency control algorithm for the R-trees, which is designed to support fast and frequent updates. The purpose of the concurrency control algorithms of index is to provide high throughput for frequent update. And the key criteria for concurrency control algorithm is the degree of parallelism. The previous work showed that the main factors which block the other concurrent operation and hence decrease the concurrency of index are MBR modification propagation and node split of update operations. Our algorithm reduces the query blocking overhead to address these two problems. Additionally, we propose an optimistic search algorithm to speed up the processing of query.

4.2 Function of Time

Recall that object positions are being modeled as functions of time to minimize updates. Modeling object positions as functions of time enables the "tentative near-future" positions be approximated. The database will only need to be updated when the parameters of the linear function change or the objects move beyond a bound, e.g. the vehicle changes direction or stops moving. The objects may report their parameter values when their actual positions deviate from the last reported positions by certain threshold. One main problem is that each individual object has a different function of movement and agility and that function always changes unpredictably. For example, a vehicle can never maintain constant velocity in a traffic-congested city. In fact, the speed can vary from 0 to 55 mph in a matter of seconds. The same goes for tracking enemy aircraft and missiles that can change direction, speed or even destinations while on the move. Either we have a more sophisticated and robust way of representing the movement of mov-

ing objects, e.g. using complex polynomial functions with intelligent predictions or we are faced with an enormous amount of updates. However, the approximation provides additional degree of accuracy since objects may move since the last update and before the next update on the database.

4.3 Lazy Updates

Movement of objects in the database causes deletion and insertion of objects, and consequently, node merging and splitting. For objects that do not move out from the present MBR, no deletion is necessary although a deletion and reinsertion may provide a more efficient structure. Object A in Figure 6 is a case in point, where a deletion and reinsertion will definitely make the coverage of the current node smaller. For applications whose objects do not move far out from the current MBR, such as object B in Figure 6, an enlarged MBR could be used. The level at which MBRs are to be enlarged is a decision between query performance and update cost. While enlarging MBRs reduces updates, the enlarged MBRs may overlap more and cause the subtree to be traversed unnecessarily.

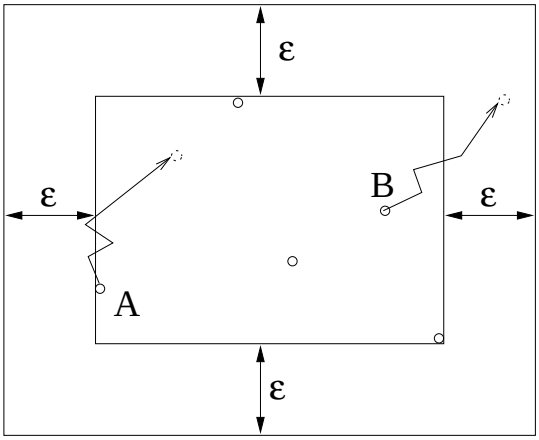


Figure 6. Extended MBR

4.4 Buffering Strategies

Work on buffer management has progressed from the design of replacement policies that are based on stochastic measures to the design of more sophisticated approaches that integrate additional domain knowledge such page reference patterns and external domain hints to obtain more intelligent buffering schemes. The main objective of buffering is to reduce expensive I/O cost. The reduction in I/O operations results in higher throughput as both update and retrieval operations can be executed more efficiently. In [5], a buffer replacement strategy for hierarchical indexes based on index traversal pattern has been shown to improve buffer hit rate and reduce I/O cost. For many applications, the movements of objects are constrained by factors such as area of service and transportation network. Such constraints cause certain patterns of movements to form and hence may cause the objects to be moved around within certain subtree. A replacement strategy can then be designed to exploit such patterns of movements. For a given subtree which has frequent updates, the subtree becomes more likely to be kept in the buffer due to its access pattern, or can be cached on purpose for speedy updates.

Most modern servers come with fast caches to speed up computation, and the recent focus has been on the effective utilization of the L2 cache to minimize cache misses. This is because a L2 cache miss incurs more than 200 processor clocks when data is required to be fetched from the slower but larger capacity RAM or conventional buffer (as compared to only 2 processor clocks for a L1 cache hit). The objective of exploting L2 cache is similar to that of buffer, except now that we have smaller cache lines, and hence to fully exploit L2 cache, indexes have to be tuned with page structures that fit into cache lines whose sizes are usually 32-128 bytes. Such page structure does not map directly into the disk-based structure, therefore mapping is required, which may not be straight forward or even feasible depending on the index structure.

4.5 Use of Auxiliary Structures

Since moving objects have to be identified by identifiers, the update can be done based on the identifier rather than spatial location. To support fast location of moving objects in the spatial index, a secondary indexing structure such as hash-table can be maintained to provide a direct link between the object and the leaf page in spatial index that contains the object [16]. Without a secondary index, location of moving object has to be based on the last location recorded in the database, and the search for the object within the spatial index is much slower than a direct lookup using a secondary index. However, in order to provide fast update, a backward or parent pointer has to be maintained such that the tree can be adjusted should the moving object be removed from the current node. An example is illustrated in Figure 7. While adjusting on the way up, identification of the right subtree for re-insertion due to the new location can be performed concurrently. If the objects are not moving too far away from the current location, re-insertion is likely to be within the same subtree before reaching the root. Together with the use of enlarged MBR, updates could be localized to some subtrees to reduce traversal and locking of nodes.

Song and Roussopoulos [23] proposed a hashing based method to reduce update costs. Though simple and intuitive, it is difficult to support the various types of queries

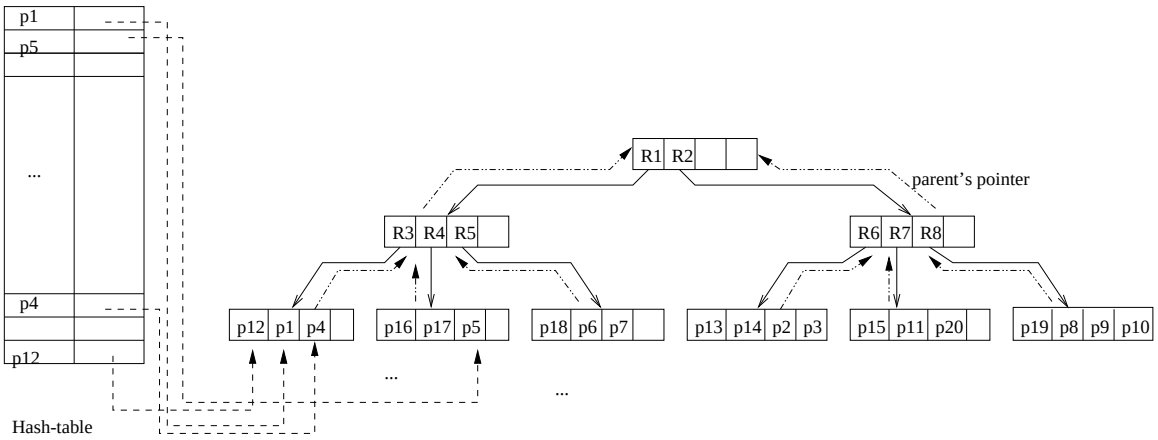


Figure 7. Use of a secondary structure

(KNN, RNN, spatial joins) mentioned earlier on. Furthermore, overflow pages can seriously deteriorate performance as all pages need to be searched, made worse if we have a skewed or Gaussian distributed data.

5 Conclusion

The demand for tracking the locations of moving objects is fueled by the advancement of GPS, wireless technologies and small but powerful digital devices. Spatial indexes that have been typically designed to index static objects and facilitate fast retrieval may not be efficient for indexing moving objects where updates are frequent. Fast update is not an objective contradictory to fast retrieval – it is an added requirement on indexes for moving objects. Assuming that the base index is already providing fast retrieval, which has been the objective all this while, the index now has to minimize the number of nodes being locked and updated so that the overall throughput remains acceptable. In this paper, we examined various design issues and techniques in designing indexes for supporting fast retrieval of moving objects based on spatial location and proximity, and frequent updates. Most techniques discussed are complementary and can be used in parallel to achieve the objective of fast retrieval and update.

References

[1] P. K. Agarwal, L. Arge, and J. Erickson. Indexing moving points. In *Proceedings of the 19th ACM Symposium on PODS*, pages 175–186, 2000.

[2] R. Benetis, C. Jensen, G. Karciauskas, and S. Saltenis. Nearest neighbor and reverse nearest neighbor queries for moving objects. In *Proceedings of the International Database Engineering and Applications Symposium*, pages 44–53, Edmonton, Canada, July 2002.

[3] E. Bertino, B. Ooi, R. Sacks-Davis, K. Tan, J. Zobel, B. Shilovsky, and B. Catania. *Indexing Techniques for Advanced Database Systems*. Kluwer Academic Publishers, 1997.

[4] K. Chakrabarti and S. Mehrotra. Dynamic granular locking approach to phantom protection in r-trees. In *Proceedings of the 14th International Conference on Data Engineering*, pages 446–454, 1998.

[5] C. Y. Chan, B. C. Ooi, and H. Lu. Extensible buffer management of indexes. In *Proc. 18th International Conference on Very Large Data Bases*, pages 444–454. 1992.

[6] J. K. Chen and Y. F. Huang. A study of concurrent operations on r-trees. In *Information Science 98*, pages 263–300, 1997.

[7] H. Chon, D. Agrawal, and A. El Abbadi. Query processing for moving objects with space-time grid storage model. In *Proceedings of 3rd International Conference on Mobile Data Management*, 2002.

[8] B. Cui, Z. Huang, B. Ooi, and K. Tan. Concurrency control mechanisms for supporting fast updates in the r-tree. Technical report, School of Computing, NUS, 2002.

[9] G. Faria, C. B. Medeiros, and M. A. Nascimento. An extensible framework for spatio-temporal database applications. Technical Report TR-27, TIMECENTER Technical Report, 1998.

[10] M. Hadjieleftheriou, G. Kollios, V. J. Tsotras, and D. Gunopulos. Efficient indexing of spatio-temporal objects. In *Proceedings of 8th Conf. EDBT*, 2002.

[11] C. Jensen, editor. *Special Issue on Indexing of Moving Objects*. IEEE CS, 2002.

[12] K. V. R. Kanth, F. D. Serena, and A. K. Singh. Improved concurrency conrol techniques for multi-dimensional index structures. In *12th International Parallel Processing Symposium*, pages 580–586, 1998.

[13] G. Kollios, D. Gunopulos, and V. J. Tsotras. On indexing mobile objects. In *Proceedings of the 18th ACM Symp. on PODS*, pages 261–272, 1999.

[14] M. kornacker and D. Banks. High-concurrency locking in r-trees. In *Proc. 21th VLDB Conference*, pages 134–145, 1995.

[15] M. Kornacker, C. Mohan, and J. M. Hellerstein. Concurrency and recovery in generalized search trees. In *Proc. of the ACM SIGMOD Conference*, pages 62–72, 1997.

[16] D. Kwon, S. Lee, and S. Lee. Indexing the current positions of moving objects using the lazy update. In *Proceedings of 3rd International Conference on Mobile Data Management*, pages 113–120, 2002.

[17] P. Lehman and S. Yao. Efficient locking for concurrenct operations on b-trees. In *ACM Transactions on Database Systems (TODS)*, volume 6, pages 650–670, 1981.

[18] V. Ng and T. Kamada. Concurrenct accesses to t-treestrees. In *Proc. of Symposium on large spatial databases*, pages 142–161, 1993.

[19] D. Pfoser, C. S. Jensen, and Y. Theodoridis. Novel approaches in query processing for moving object trajectories. In *Proceedings of 26th Int'l. Conf. on VLDB*, pages 395–406, 2000.

[20] Y. Sagiv. Concurrenct operations on b*-trees with overtaking. *Journal of computer and system sciences*, 33(2):275–296, 1986.

[21] S. Saltenis, C. Jensen, S. Leutenegger, and M. A. Lopez. Indexing the positions of continuously moving objects. In *Proceedings of ACM SIGMOD Int'l. Conf. on Management of Data*, pages 331–342, 2000.

[22] S. Song, Y. Kim, and J. Yoo. An enhanced concurrency control scheme for multi-dimensional index structures. In *Proc. of the 7th DASFAA Conference*, pages 190–199, 2001.

[23] Z. Song and N. Roussopoulos. Hashing moving objects. In *Proceedings of 2nd International Conference on Mobile Data Management*, pages 161–172, 2001.

[24] Y. Theodoridis, T. Sellis, A. N. Papadopoulos, and Y. Manolopoulos. Specifications for efficient indexing in spatiotemporal databases. In *IEEE SSDBM*, 1998.

[25] C. Yu. *High-dimensional Indexing*. Lecture Notes in Computer Science 2341, Springer-Verlag, 2002.