Q+Rtree: Efficient Indexing for Moving Object Databases[*]

Yuni Xia Sunil Prabhakar
Department of Computer Science
Purdue University, West Lafayette, IN 47906, USA
{xia, sunil}@cs.purdue.edu

Abstract

Moving object environments contain large numbers of queries and continuously moving objects. Traditional spatial index structures do not work well in this environment because of the need to frequently update the index which results in very poor performance. In this paper, we present a novel indexing structure, namely the Q+Rtree, based on the observation that i) most moving objects are in quasi-static state most of time, and ii) the moving patterns of objects are strongly related to the topography of the space. The Q+Rtree is a hybrid tree structure which consists of both an R-tree and a QuadTree. The Rtree component indexes quasi-static *objects – those that are currently moving slowly and are often crowded together in buildings or houses. The Quadtree component indexes fast moving objects which are dispersed over wider regions. We also present the experimental evaluation of our approach.*

1 Introduction

The advances of wireless communications technologies, personal locator technology and global positioning systems enable a wide range of location-aware services, including location and mobile commerce(L- and M-commerce). Current location-aware services support proximity-based queries including map viewing and navigation, driving directions, and searching for restaurants and hotels. The demand for storing, updating and processing continuously moving data arises in a large number of applications such as the digital battlefield, mobile e-commerce and traffic control and monitoring.

In this paper, we address the problem of indexing continuously moving objects, which could be critical for evaluating queries in response to the movement of objects with near real-time responses. Traditional spatial index structures such as Rtree [4] are not appropriate for indexing moving objects because the constantly changing locations of objects require constant updates to the index structure which greatly degrades its performance. To reduce the number of index updates, many previous schemes [1][6][12][14] use a simple linear function to describe the movements of the objects, where the index and the database are updated only when the parameters of the linear function change. However, in reality, the movements of objects are far too complicated to be accurately represented as a linear function that changes infrequently.

We develop a novel technique, the Q+Rtree, to efficiently index the positions of the moving objects and reduce the update cost to a great extent. The basic idea is to differentiate fast moving objects from quasi-static objects, which account for the majority of all moving objects. Although fast moving objects constitute only a very small fraction of all moving objects, their movements are the main reason for the huge index updating overhead and degradation of index performance. In Q+R tree, quasi-static objects are stored in an Rtree and fast-moving objects are stored in a Quadtree. Objects may switch between two trees when they change their moving status, e.g., if a person moves out of a home and travels on a freeway, it will change from quasi-static state into the fast-moving state.

In this paper, we investigate several index structures for efficient index updating and query evaluation. Our results show that Rtree alone gives good query performance but poor index update performance. On the other hand, Quadtree does pretty well when updating the index, but its query performance is worse than that of the Rtree. By combining them together, Q+Rtree achieves a better performance for both index updating and query evaluation.

Our work distinguishes itself from related work in that it makes use of the topography and the patterns of object movement. By handling different types of moving objects separately, our index structure more accurately reflects the reality and results in better performance. In our work, no assumption is made about the future positions of objects. It is not necessary for objects to move according to well-behaved patterns and there are no restrictions, like the max-

[*]Portions of this work were supported by NSF CAREER grant IIS-9985019, NSF grant 0010044-CCR and NSF grant 992883.

imum velocity, placed on objects either.

The rest of the paper proceeds as follows. Related work is discussed in Section 2. In Section 3, we describe the problem being addressed and present the novel index structure, Q+Rtree, which reduces index updating cost and improves query performance. Experimental evaluation of the proposed approach is presented in Section 4 and Section 5 concludes the paper.

2 Related Work

Developing efficient index structures is an important research issue for moving object databases. As a naive approach, multi-dimensional spatial index structures can be used for indexing the positions of moving objects. Numerous index structures have been proposed for indexing multi-dimensional data. An excellent survey of these indexing schemes can be found in [3]. Recently, in [7] Kothuri et al. argue that R-trees are generally better than Quadtrees and Oracle now recommends the use of only the R-tree. Although traditional spatial index structures can be used, they are not efficient for indexing the positions of moving objects because of frequent and numerous update operations in moving environments.

Some new index structures have been proposed for indexing moving objects recently. These index structures can be classified into the two categories. Those that index: (1) the trajectories (histories) and (2) the current positions of objects. Our approach belongs to the latter category.

In the first category, object movement in a d-dimensional space is converted into a trajectory in a (d+1) dimensional space when time is treated as a dimension. Examples of this approach are the Spatio-Temporal R-tree (STR-tree) and Trajectory-Bundle tree (TB-tree) proposed in [9]. The authors showed that these two structures work better than traditional spatial index structures for queries related to trajectories. In [13], Tao and Papadias have proposed the Multi- version 3D R-tree(MV3R-tree), which combines multi-version B-trees and 3D-Rtrees.

In the second category, most approaches describe a moving object's location by a linear function, and only when the parameters of the function change, for example, when the moving object changes its speed or direction, is the database updated. The time-parameterized R-tree (TPR-tree) has been proposed in by Saltenis et al. [12]. In this scheme, the position of a moving point is represented by a reference position and a corresponding velocity vector. When splitting nodes, the TPRtree considers both the positions of the moving points and their velocities. Kollios et al. [6] proposed an efficient indexing scheme using partition trees. Tayeb et al. [14] introduced the issue of indexing moving objects to query the present and future positions. They proposed PMR-Quadtree for indexing moving objects. Agarwal et al.[1] proposed various schemes based on the duality and they developed an efficient indexing scheme to answer approximate nearest-neighbor queries.

All these techniques rely upon a good representation of the future movement of objects. They suffer from the problem objects in reality do not follow predictable paths. In many applications, the movement of objects is complicated and non-linear. In such situations, the approaches based on a linear function cannot work efficiently – the function changes too often. Song and Roussopoulos [11] proposed a new idea based on hashing to solve this problem. Their approach is simple and intuitive. However, since it is based on a simple hashing, it might cause problems such as long chains of overflow pages [8]. In [10], we propose two novel approaches, namely Query Indexing and Velocity-Constrained Indexing (VCI), for indexing moving objects. Both approaches achieve significant improvements over traditional approaches. However, Query Indexing cannot efficiently handle the arrival of new queries, while the VCI index does not have good performance when the number of concurrent queries is large.

It should be noted that our approach of using two separate index structures – one R-tree and one QuadTree – is quite different from the hybrid tree [2]. In [2], Chakrabarti et al. presented the hybrid tree, which is basically a space partitioning based data structure that allows the index subspace to overlap. The overlap is allowed only when trying to achieve an overlap-free split would cause downward cascading splits and hence a possible violation of utilization constraints. It combines positive aspects of both space partitioning and data partitioning based index structures to achieve better search performance. The hybrid tree they proposed is for indexing high-dimensional feature spaces.

3 Q+Rtree

In this section, we present our proposed index structure, called the Q+Rtree, to efficiently index and query moving objects. We first introduce the basic idea and motivation of building the Q+Rtree, followed by the details of its construction, update, and query processing.

3.1 Observations

Our Approach is based on two important observations about moving object environments. Firstly, most moving objects do not move at high speeds most of the time. In fact, most objects (especially human beings) are in a *, quasi-static* state most of the time. The term *quasi-static* is used to describe a state where the object is not perfectly static (such as a parked park) but rather is moving within a small region of space (e.g. an office, home or building). The majority of people spend most of their time at home or in an office, where their movements are small and slow. There are objects, of course, that move almost all the time, like taxis or city buses, but the proportion of these constantly moving objects is very small. We can safely estimate that normally, at any time, more than 80% of the objects are in a quasi-static state, which we currently define as moving less than 30 meters per minute. Based upon experiments with the City Simulator developed at IBM Almaden [5], which is designed to generate realistic motion data for cities, we found that the results are consistent with this estimation.

Secondly, the movements of objects are generally related to topography. For instance, huge buildings often contain hundreds or even thousands of people, who are in quasi-static states. On the other hand, fast moving objects are usually on roads or freeways (and not likely to be in buildings). Furthermore because the topography does not change over a short time, we can exploit the topography, the relatively stable elements (compared to the moving objects), to build the index.

Based upon these observations about moving objects we propose the hybrid Q+Rtree.

3.2 Q+Rtree

Considering the distribution and moving patterns of objects, we separate the quasi-static objects and fast-moving objects and build separate indexes for each type.

For quasi-static objects, the update frequency is expected to be lower since their velocities are smaller. The range of movement of quasi-static objects is normally small too, such as within an office building or a house. Therefore, if we build an Rtree over the quasi-static objects, the chances that they move out of their current MBRs are small, which can reduce a large amount of index updating overhead by using the Lazy Update approach [8]. This approach updates the structure of the index only when an object moves out of the corresponding MBR. If the new position is still within the MBR, only the position of the object in the leaf node is updated. Furthermore, if we take a look at the distribution of objects, quasi-static objects are often crowded together (e.g. in buildings, parks, or schools), which makes the low-level MBR of the Rtree small and packed. This will increase the precision of the index and speed up searching (in terms of efficient pruning during query evaluation).

For fast-moving objects, we build a Quadtree index. It is not appropriate to use an Rtree index for these fast-moving objects since they are likely to often move out of their current MBRs and result in significant index updating overhead. Moreover, fast-moving objects are more likely to be widely distributed over the space instead of being crowded together, which would result in large (in terms of coverage) nodes and less efficient searching. A Quadtree, on the other hand, works well in this scenario. Since each quadrant can accommodate a certain number of objects, when the density of the objects is low, the area of the quadrant can be very large. Therefore, even if objects are moving fast, there is a small chance that the object will move out of its current quadrant, which makes it easy to update the index (if the object remains in its current quadrant, no update to the index structure is needed).

Any index for moving objects suffers from two conflicting requirements. On the one hand, a large internal node is desirable so that objects will not constantly move out of the range of the nodes and result in changes to the index structure. On the other hand, a small and tight internal node is desirable so that the index can efficiently support queries. By separating slow and fast moving objects, building a loose index for fast objects and a tight index for slow objects, we achieve both goals.

3.3 Q+Rtree Construction

The process of building a Q+Rtree consists of three steps:

1. *Build a Topography-Based Rtree for quasi-static objects.* A simple approach is to build an R-tree over the slow-moving objects in the usual fashion – i.e. either insert the objects one by one into the R-tree, or perform a bulk loading operation. However, our approach takes a different alternative. Instead of building the index using the quasi-static objects, we build an R-tree index over topographical regions. These regions can be determined from an analysis of maps if they are available, or from the past behavior of users. The motivation for building an index over the topographical regions is that these regions represent reasonable bounds that quasi-static objects will remain within for large periods of time. Once this index is created, the quasi-static objects are inserted into the index while treating the topographical objects from the previous step simply as MBR one level above the new leaf level – at which the objects are stored.

   This Rtree, as shown in Figure 1, has the following features:

   (a) The level above the leaves store topographical regions. Therefore, at this level, there is no overlap between the nodes. This is different from a traditional Rtree, which might have overlap between MBRs at each level.

   (b) The leaves of the tree store moving objects. The leaf level is a special level where there is no Maximum/Minimum Entries per node restriction. All objects that belong to a given topographical region are inserted below that region. This also

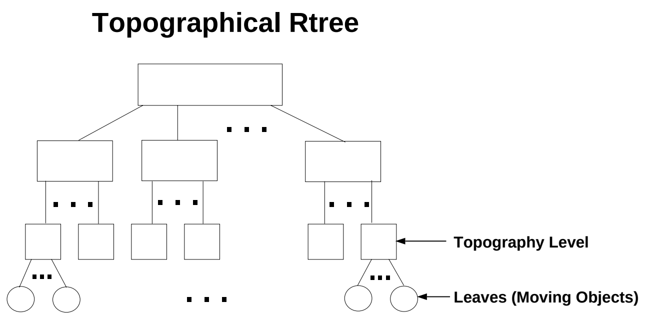


Figure 1. Example of Rtree over the Quasi-Static Objects

makes insertion an easier procedure since there is only one place to insert an object, while in traditional Rtree, an object could be inserted into any node and large amounts of computation needs to be performed to determine which node it should be inserted into, (for example, finding a node that needs least enlargement to accommodate this object or results in least overlap of MBRs). Since there are no Maximum/Minimum Entries per node restriction for the leaf nodes, we use a dynamic array to increase the space utilization.

2. *Build a Quadtree.* The second step is to build a Quadtree over the entire space by insert all objects not in the slow-moving cells. This process is identical to the regular Quadtree operations.

3. *Combine the Quadtree and Rtree structures.* Since the Quadtree and the Rtree overlap in space, at the leaves of the QuadTree, we build a link list for each quadrant with pointers to each of the Rtree nodes contained by or intersecting the quadrant. The Rtree nodes pointed to by the link list can be at different levels. An example of this combined tree is shown in Figure 3 corresponding to the city shown in Figure 2.

3.4 Updating the Q+Rtree

When an update arrives with an object's new location, it could be one of the following cases:

1. The object is currently in the Rtree, its new position is still in the Rtree, corresponding to the scenario that the quasi-static object stays in the slow-moving area. We need to check if its current MBR contains the new location or not.

    (a) If the new position is still in its current MBR,

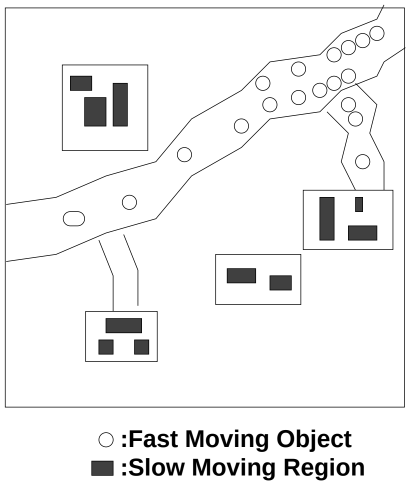


Figure 2. Example of a city

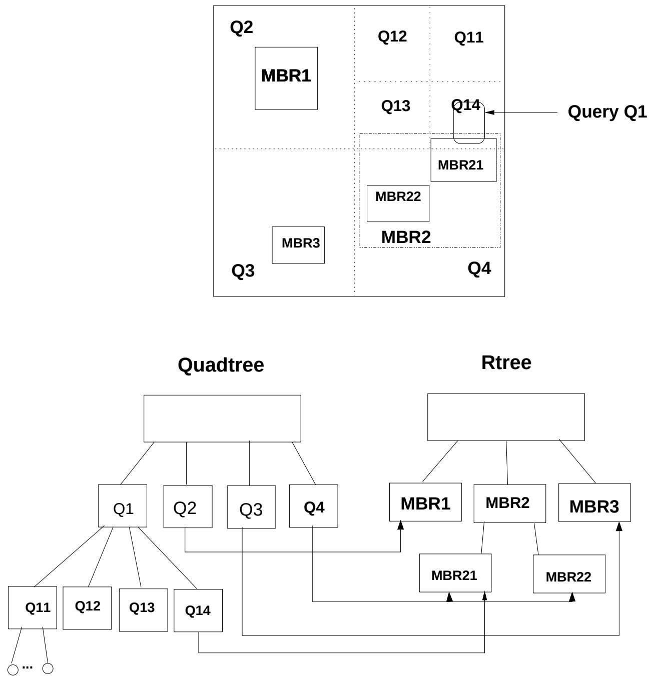


Figure 3. Example of the Combined Q+R Tree

just modify the position of that object in the leaf node.

   (b) If the new position is not in its current MBR, delete it from current node and insert it with its new position.

2. The object is currently in the Rtree, its new position is not in the Rtree: since the Rtree, as Figure 1 shows, contains all the slow moving regions, this corresponds to the scenario that the quasi-static object leaves the slow-moving area and moves into a fast-moving area, such as a freeway. The object should be deleted from the Rtree and inserted into the Quadtree.

3. The object is currently in the Quadtree, its new position is in the Rtree: corresponding to the scenario that a fast moving object leaves the fast-moving area and moves into a slow-moving area The object should be deleted from the Quadtree and inserted into the Rtree.

4. The object is currently in the Quadtree, its new position is still in the Quadtree: corresponding to the scenario that a fast moving object keeps moving in the fast-moving area We need to check if its current quadrant contains the new location or not.

   (a) If the new position is still in its current quadrant, just modify the position of that object entry

   (b) If the new position is not in its current quadrant, delete it from current node and insert it into the new node.

Since most of the objects are in slow-moving areas and stay in quasi-static states most of the time, The majority of the situations are expected to be case 1.1, which does not result in a large update overhead.

3.5 Query Evaluation with Q+Rtree

In order to process a range or point query, the search begins with the Quadtree. For Example, as show in Figure 3, when range query Q1 arrives, we start searching from the Quadtree and find that quadrant Q14 intersects with the query, consequently we search all objects under quadrant Q14. In addition, since Q14 also links to MBR21 in the Rtree, we need to continue the search for objects indexed under MBR21 in the Rtree as well. Because the Quadtree and the Rtree overlap in space, many queries will result in searching both trees. The link lists from Quadtree nodes pointing to Rtree nodes are very useful in improving the search time for these queries. This link list speed up the search since we do not have to start from the Rtree root for each query.

Parameter	Value
Number of Objects	100,000 - 1,000,000
Number of Queries	100,000

Table 1. Parameter used in the experiments

4 Experimental Evaluation

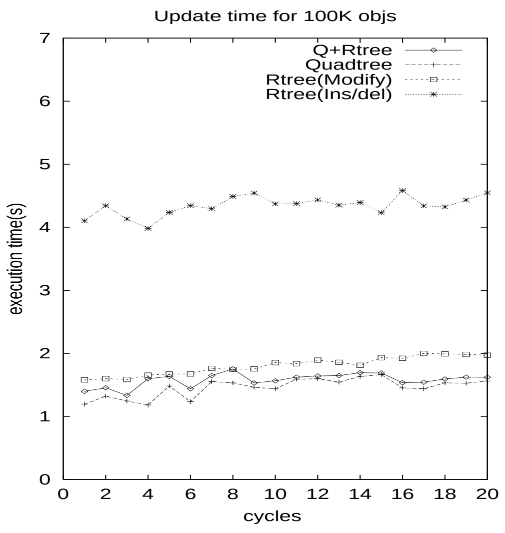
In this section, we present some experimental results for the performance of the Q+Rtree relative to both the Quadtree and R-tree with respect to update and search performance. We compare the following 4 approaches: i) the Q+Rtree, ii) the Quadtree, iii) the R*tree which updates the index by deletion and insertion, i.e., it deletes the old positions and insert the new ones, and iv) the R* tree which updates the index by modifying the MBR, i.e. it extends the MBR to include the new positions if necessary. The results report the actual execution time for the various cases. The experimental settings are described first, followed by the results and discussion.

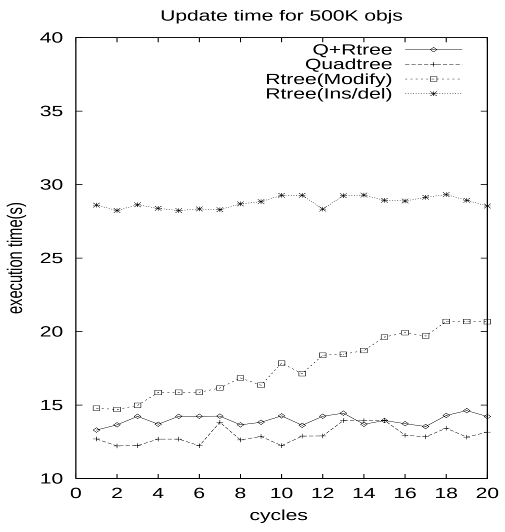
4.1 Experimental Setup

In all our experiments, we used a 1G Hz Pentium III machine with 2GB of main memory. This machine has 32K of level 1 cache, of which 16K is for instructions and 16K for data, and 256K level 2 cache. Due to the unavailability of actual object movement data, we used a synthetic dataset generated by the City Simulator, developed at IBM Almaden. City Simulator is a scalable, three-dimensional model city that enables creation of dynamic spatial data simulating the motion of up to 1 million people. It is designed to generate realistic data for evaluation of database algorithms for indexing and storing dynamic location data.

For generating the slow-moving topographical regions we employed the same input that was fed as a map into the City Simulator. In the City Simulator, the map is described by an XML file, which specifies detailed information about roads, buildings, parks, etc. We wrote a slow-moving area finder, which scans the XML file and find out all slow-moving regions. Then, we build an Rtree over all of the slow-moving regions, and insert the objects that fall in these cells into the leaf nodes of this Rtree.

The experiments were repeated with datasets of three different sizes. We generated datasets with 100K, 500K and 1M objects respectively. Object movement and query evaluation are carried in cycles. Each cycle consists of two steps: updating the indexes with the new object locations and evaluation of queries. A set of queries is continuously evaluated. We measure the performance of each step separately. In each set of graphs, we present the results for 100k, 500K, and 1M objects. The x-axis gives the number of cycles for





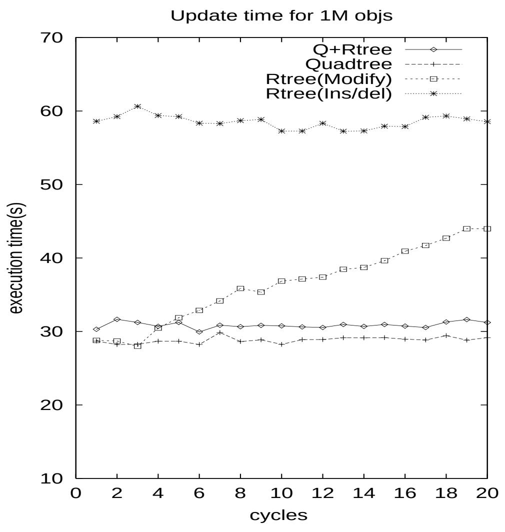


Figure 4. Index Updating time

which the tests were run, and the y-axis gives the actual total execution times observed. We test the performance for 20 cycles. Assume that objects report their locations every 3 minutes, 20 cycles corresponds to an hour.

4.2 Update Performance

In the first experiment, we compared the four approaches in terms of the time to process location update operations for objects in each cycle. Figure 4 shows that both alternatives for the R*tree take more time to update that the Quadtree and the Q+Rtree. Furthermore, the insert/deletion option is particularly poor. The performance of the Quadtree is clearly the best in all cases, although the Q+R-tree does not lag too far behind.

4.3 Search performance

In this experiment, we measured the performance for range queries. The number of queries is the fixed at 100,000. In each dimension, the range of the query windows are approximately 5 percent of the entire range. Query windows are uniformly distributed in the space.

Figure 5 shows that the R*tree, although suffering from the huge indexing updating overhead, performs well in searching. On the contrary, the Quadtree, which is fast for index updating, is not very efficient for searching. We can also see that the modifying scheme makes the R*tree performance deteriorate quickly and thereby result in increased searching time. This is not surprising since the advantage of the modify approach lies in avoiding the cost of moving objects from one node to another. However over time this results in an increase in the average size of MBRs as well as the amount of dead space. Consequently, it is not surprising that the search performance is poorer. The search performance of Q+Rtree is a little bit worse than pure R*tree. However, as showed in the previous section, pure R*tree has a large update overhead, therefore, when considering the overall performance, as we will see next, Q+Rtree outperforms both pure R*tree and pure Quadtree.

4.4 Overall performance

For each cycle, the system needs to first update the index, then evaluate the queries. Figure 6 shows the total cost of the four schemes in each cycle by adding up their updating cost and the query processing cost. Clearly, the Q+Rtree, with good performance in both updating and searching, has the best overall performance.

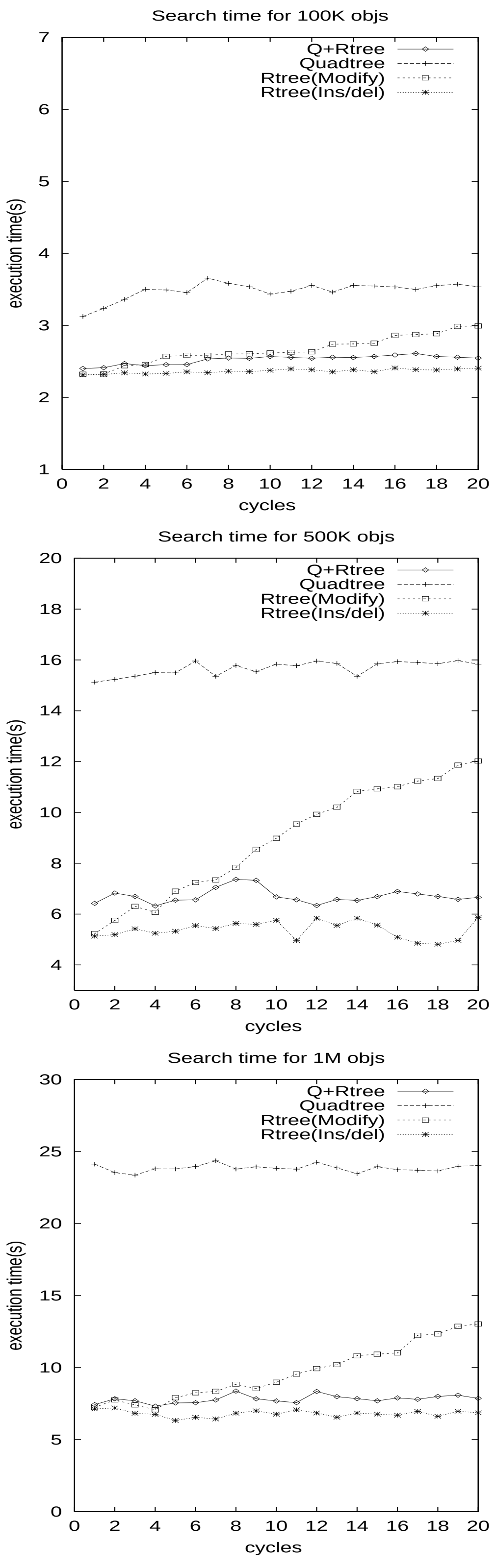


Figure 5. Query Evaluation Time

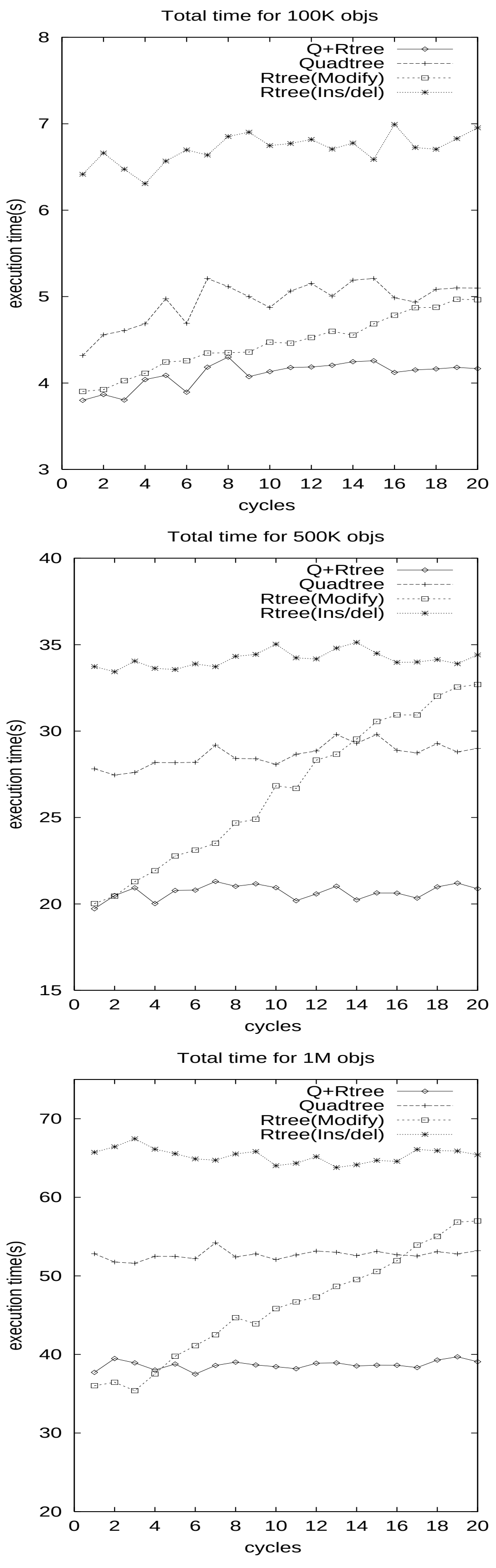


Figure 6. Total Time

5 Conclusion

Traditional spatial index structures do not work well in moving object environments, which are characterized by large numbers of continuously moving objects and concurrent active queries over these objects. The need for frequent index updating results in poor performance. Some early techniques try to reduce the number of updates by approximating the movement of moving objects as a linear function, but the movement of real objects are too complicated to be described as a linear function.

We present a novel indexing technique for scalable execution: Q+Rtree. The Q+Rtree differentiates quasi-static objects, which account for the majority of all moving objects, and fast-moving objects and stores them in different index structures. It also makes use of the topography, which can affect or even determine movement characteristics of the objects. Our experiments demonstrate that Q+Rtree achieves significant improvement over the traditional approaches.

References

[1] P. K. Agarwal, I. Arge, and J.Erickson. Indexing moving points. *Proc. 2000 ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems(PODS)*, May 2000.

[2] K. Chakarabarti and S.Mehrotra. The hybrid tree: An index structure for high dimensional feature spaces. *Proceedings of he Fourteenth International Conference on data engineering(ICDE'99)*, 1999.

[3] V. Gaede and O. Gunher. Multidimensional access methods. *ACM Computing Surveys*, pages 170–231, 1998.

[4] A. Guttman. R-trees: A dynamic index structure for spatial searching. *Proc. of the ACM SIGMOD Int'l. Conf.*, 1984.

[5] James Kaufman, Jussi Myllymaki, and Jared Jackson. City simulator http://www.alphaworks.ibm.com/tech/citysimulator.

[6] G. Kollios, D. Gunopulos, and V. J. Tsotras. On indexing mobile objects. *Proc. 1999 ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems(PODS)*, June 1999.

[7] Ravi Kanth V Kothuri, Siva Ravada, and Daniel Abugov. Quadtree and r-tree indexes in oracle spatial: a comparison using gis data. *Proceedings of ACM SIGMOD Conference*, 2002.

[8] D. Kwon, S.J. Lee, and S. Lee. Indexing the current positions of moving objects using the lazy update r-tree. *3rd International Conference on Mobile Data Management*, Jan 2002.

[9] D. Pfoser, C. S .Jensen, and Y. Theodoridis. Novel approaches in query processing for moving objects. *Proceedings of the 26th International Conference on Very Large Databases(VLDB)*, September 2000.

[10] S. Prabhakar, Y. Xia, D. Kalashnikov, W. Aref, and S. Hambrusch. Query indexing and velocity constrained indexing:scalable techniques for continuous queries on moving objects. *IEEE Transaction on Computers*, 51(10):1124–1140, Oct, 2002.

[11] Z. Song and N.Roussopoulos. Hashing moving objects. *Proc. of the 2nd Int'l Conf. on Mobile Data Management*, pages 161–172, 2001.

[12] S.Saltenis, C.Jensen, S.Leutenegger, and M.Lopez. Indexing the position of continuously moving objects. *Proceedings of ACM SIGMOD Conference*, 2000.

[13] Y. Tao and D.Papadias. Mv3r-tree: a spatio-temporal access method for timestamp and interval queries. *Proc. of 27th Int'l Conf. on Very Large Data Bases*, 2001.

[14] Jamel Tayeb, Ozgur Ulusoy, and Ouri Wolfson. A quadtree-based dynamic ttribute indexing method. *The Computer Journal*, pages 185–200, 1998.