Hashing Moving Objects

Zhexuan Song
Department of Computer Science
University of Maryland
College Park, Maryland 20742
zsong@cs.umd.edu

Nick Roussopoulos
Department of Computer Science &
Institute For Advanced Computer Studies
University of Maryland
College Park, Maryland 20742
nick@cs.umd.edu

May 19, 2000

Abstract

In real-life applications, the objects are both spatial and temporal referenced. The objects which continuously change their location are called moving objects. With the development of wireless communication and positioning technology, it becomes necessary to store and index those objects in database. Due to the complexity of the problem, many pure spatial index structures are unable to index large volume of moving objects in database.

In this paper, we propose a whole new idea based on hashing technique. Since it is impossible to re-index all the objects after each time period, we store the objects in buckets. When an object moves within a bucket, the database does not make any change. By using this technique, the number of database update is greatly reduced which makes the index procedure feasible. Then, we extend the previous system structure by introducing a filter layer between the position information collectors and the database. Also four different methods based on the new system structure are presented. Performance experiments were performed to evaluate different aspects of our indexing techniques, and the conclusions are included in the paper.

1 Introduction

Traditionally, database management systems adopt a "static" model which assumes that data stored in the database remain stable until explicitly changed through an update operation. This model serves well if the properties of the objects never or discretely change. However in real life, many objects change their properties continuously. One such application is to maintain a database in air-traffic control. In this application, the objects are flying airplanes. The location of the objects changes continuously. One possible solution in "static" model which we called "naive solution" is to update the location information of the objects in database after each time period. Considering to large update overhead, the solution seems to be very inefficient.

With the rapid development of positioning system, e.g. GPS, wireless communication technologies, and electronics, it is technically feasible and necessary to track and record the positions of large amount of mobile objects. According to [SJL+99], the mobile phone market expect more than 500 million mobile phone users by year 2002 and 1 billion by year 2004, and mobile phones are evolving into wireless Internet terminals. Keeping track of these terminal location may substantially improve the quality of the services. The requirement for more sophisticated database system becomes urgent.

The new database which deals with *geometries changing over time* is called spatiotemporal database. The problems in this new field recently attracted interest of both the academic and the industrial community. [WCD+98] presents the Moving Object Spatio-Temporal (MOST) model and a language (FTL) for querying the current and future location of moving objects; [TJ98] proposes a component-based conceptual model for spatiotemporal application design; Nascimento et al. [NST99] present the GSTD algorithm ("Generate Spatio-Temporal Data"), which generates sets of moving point or rectangular data that follow an extended set of distributions. The synthetic data sets generated by GSTD have been used to evaluate different index designs. ArcView GIS system [ArcV98] has already supported tracking and querying mobile objects.

Our main focus in this paper is how to index large number of moving objects without generating high volume of database updates. We mostly discuss how to quickly respond to range queries over the objects' current location because range queries serve as a basic operation for other queries such as nearest neighbor queries [CG99, SK98]. Since the answer is based on the current knowledge stored in the database, we want that knowledge to be as precise as possible. The main challenge here is how to avoid the prohibitively large update overhead.

The "naive solution" fails when the number of moving objects goes large. Suppose a database management system can finish up to N_t transactions per second, (in most cases, N_t is less than 1000.) The number of objects is N_o. After each time period, in "naive solution", the location information of all the objects need to be updated. Thus, there are N_o update operations per update cycle. It costs at least N_o/N_t seconds to finish. When N_o is small, for example several thousand in the air-traffic control application, the "naive solution" is still fine. However, when N_o is big, for example several million in traffic monitoring, or even more in mobile communication, each update cycle costs tens of minutes or even longer. That means, the location information of an object stored in database may be the location of that objects ten minutes or even longer ago! The query result which based on these data seems unacceptable.

An alternate approach [KGT99, SJL+99] is to model the positions of moving objects as a function of time $f(t)$, and only update the database when the parameters of f changes. In most cases, linear functions are used because of its simplicity. At any given time t_0, we can find the location of each object by calculation $f(t_0)$. This approach can also predict the future position of objects. It may work well on some scientific database when the trace of each object is known beforehand. However, in real life, it is very hard to find a function to describe the objects' activity, or the parameters of f changes too often. For example, if we want to use a linear function to describe mobile phone user's activity. Whenever he changes the direction or speed, a database update request is generated. These changes could happen all the time. Thus, there are still too many database update here. Another drawback is after many parameter updates, we need a very complicate calculation to find the current location of each object. This will greatly damage the efficiency of the query procedure.

Our contributions in this paper include:

- We propose a new idea based on hashing technique. Each object is placed in a bucket. Only when an object goes to a new bucket, does the database make an update. This technique greatly reduce the number of database update which allow the system to store and index large number of moving objects.

- We present a new system structure. Between position collector and the database, we add a new layer called "Location pre-processing part". This layer can filter most of the database update request based on the rules we defined at the beginning.

- We give out four different methods. The first one partition the space into small buckets. The second allows some overlap between buckets which decrease the database update generated by zigzag movement of objects. The third method allows dynamic bucket update which increases the storage efficiency when the object distribution is skewed. And the last one combines the benefit of the second and the third methods. In experiment section, we also give some method selection suggestion.

In part 2, we discuss the related work in spatiotemporal database area. Then in part 3, we propose the basic idea of our methods and a new system structure to implement it. In the next part, we present four specific methods. The experiments results are given in part 5 and the last part includes the conclusions and research directions.

2 Related Work

Recently, many works has been done on the indexing of the location of moving objects. They mostly concentrate on point data. Related work can be conducted in two categories depends on the information stored in database.

The first approach stores the location information of moving objects which is obtained by periodically sampling. The movement of objects between two sample positions is described by using interpolate. The interpolate can be either linear which is the simplest one or polynomial splines [BBB87]. Then the movement of one object in d-dimensional space is described as a trajectory in a $(d+1)$-dimensional space which combined time into the same space [TUW98]. The methods which adopt this approach mainly focus on how to index the trajectories.

In [PTJ99], the authors define an R-tree based method called STR-tree. They use linear interpolate method, so the trajectory of an object is a set of line segments. In STR-tree, when do an insertion operation, the line segments within the same trajectory are more likely to be stored together. Later, in [PTJ00], they propose another structure called TB-tree which totally preserves the trajectories. They claimed that these two novel tree structure works better than traditional R-tree family when indexing moving objects.

Since this approach use interpolate to describe the object movement between two sampling position, some uncertainty factors may happened. Wolfson, et. al. [WCD+98] discuss the imprecision problem and how the DBMS can provide a bound for queries. Pfoser and Jensen pointed out in [PJ99] that given a speed limit, the possible location of an object between two sampling position should be an eclipse instead of a simple line segment. And the queries should consider this factor.

The drawback of this approach is that for large amount of object, too many database operations may occur after each sampling. For example, in STR-tree, each sampling will generate n line segment insertions in database where n is the object number. Due to the database limit, the sampling can not happen too often for a large n. This will greatly increase the uncertainty factor and cause the query imprecise.

The second approach uses a function to describe the movement of an object and store that function in database. For example, in one dimensional space, at time t_0, the position of an object is x_0, and it moves in a constant speed v. Then, at any time t, the position of the object can be described as $f(t) = x_0 + v(t - t_0)$. If we simply store f in database, there is no need to do any database update unless the object changes its speed.

In [KGT99], the authors use linear functions to describe the trajectories of objects. Since it is very hard to index an unlimit line in most spatial database, the authors map a line into a point data in the dual plane. The duality transformation allows to formulate the problem in a more intuitive manner. However, it worth mentioning that the normal range query becomes a polygon in dual space. This makes the query a little harder.

Sistla. et.al. propose a data model called MOST in [SWC+97]. In that model, each object has a special attribute called *function*. This attribute is a function of time. Without explicit update, the position of each object can be found by combining this attribute with other traditional attributes (such as *position* and *time*). The model allows the DBMS to execute instantaneous, continuous and persistent queries.

TPR-tree, an R*-tree bases structure [SJL+99], uses a very similar idea. In TPR-tree, the position of a moving object is represented by a reference position and a corresponding velocity vector. This velocity vector can be viewed as the *function* attribute in MOST. The TPR-tree supports the efficient querying of the current and projected future position of moving objects.

As we discussed in section 1, this approach can partly solve the overwhelming database update problem if the objects' movement follow some rules (such as particles in a scientific experiment). In real life, it is impossible to find a simple function to describe a human's movement. Even for a car in highway, its speed may change very frequently. By definition, each speed change generates a database update. So the number of total database update is still too high.

3 Hashing Techniques

This section presents the basic idea of hashing techniques and the structure of the system. The main consideration of our method is to decrease the number of database update so that the system has the ability to store and index large amount of moving objects.

The main difference between moving objects and static objects is that the location of moving objects varies frequently. In the database, if we want to keep track of the exact location information of objects, it is unavoidable to generate a large volume of database update. Therefore, we introduce the "fuzzy" idea: we do not update the location of objects in database unless it leaves its original position very far away. For example, suppose we are tracking the movement of a traveler in Washington DC. At t_0, he is in the Washington Memorial which is stored in our database. he may move around all the time, however we do not save that in database until he is quite far away from its original position, say at t_1, he is in the Capital Hill. Any actions between t_0 and t_1 is simply not reflected in our database.

Now there is some uncertainty in the queries. For example, if we want to find the location of object o now, the result from the database may look like "object o is currently in an area close to p_0, where p_0 is the position information stored in the database". Also in range queries, given a range R, the result includes two parts: some objects are certainly in the query range and some need further check

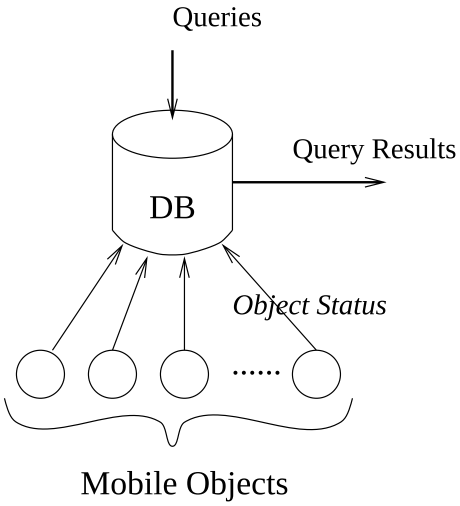

Figure 1: The structure of other methods

In order to solve the problem we design a whole new structure. Before we present our design, we first review what the structure of the other methods looks like. Figure 1 shows a common design. In the traditional structure, moving objects send their latest status directly to the database (such as location, function, velocity etc.). After receiving such information, the database executes the corresponding updates. The database always stores the latest status of each object and answer queries based on that information. The database may use different index structures (STR-tree, TPR-tree) to accelerate the update and query procedures.

The structure (Figure 2) for our method which is called *Hashing Technique*, works in a different way. We first introduce a hash function which uses object's current status as input. From this function, the system is able to find which bucket each object belongs to. The database only store the bucket information: how many objects in each bucket, which bucket each object is currently in. Between database and moving objects, we add a set of filters called "Location Pre-processing parts (LPs)". Each LP monitors a small subset of objects and uses an array to store the latest status of those objects.

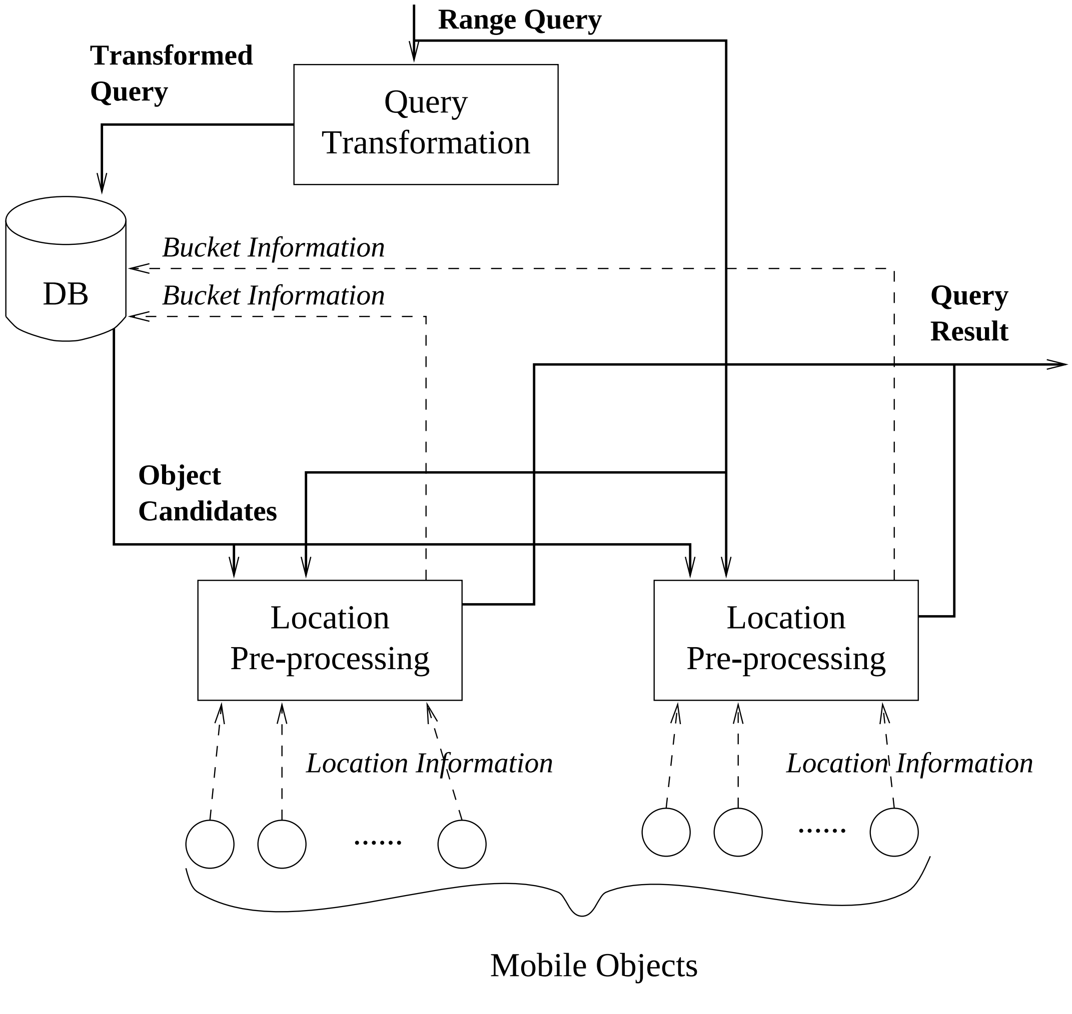

Figure 2: The structure of hashing technique

When an object changes its location and generates an update request, the request first goes to the corresponding LP. The LP updates object's status locally, then it applies the hash function to the object's latest status to see whether the object is still in the same bucket. If so, the request is simply ignored. For those objects which move into a new bucket, their requests are translated into bucket update requests and sent to the database.

In our structure, much database work is done in LPs. There are many benefits in doing so. First, each LP only monitor a small set of objects and LPs are working parallelly. When objects update their status, the system can finish the corresponding change quickly. Therefore, it is possible to do the sampling more frequently. Secondly, the scalability of the system is very good. When the object number increases, we only need to add more LPs and do not need to do big change in database. This design makes it possible to handle large number of objects.

Each bucket in the database contains all the objects which, by using hash function, have the same return value. Sometimes, the bucket can be viewed as a region in working space. The region is the union of all possible locations which, by applying the function, return the same result.

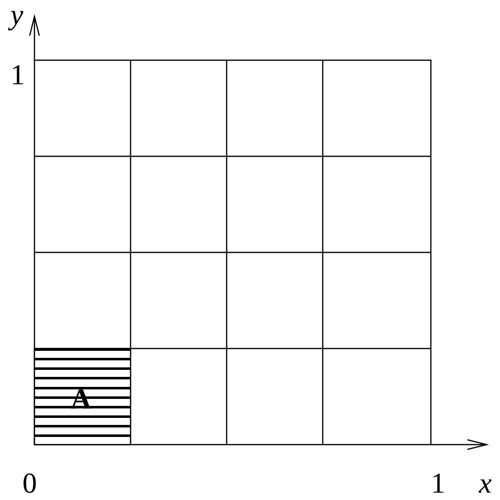

Figure 3: Hash function splits the working space into 16 regions

Example 3.1 *Assume the 2 dimensional working space is* $[0,1]^2$. *The hash function we choose is* $f(p_{x,y}) = (int(y*4))*4 + int(x*4)$ *where* (x,y) *is p's current location. By using this function, the working space is actually split into 16 regions (See Figure 3). In this case, bucket 0 can be viewed as region A because when an object is in region A, by applying this hash function, the object should* *be in bucket 0.*

As we discussed before, there is some uncertainty here if we simply use the database to answer queries. In our structure, the following parts are designed to minimize or eliminate the uncertainty. When a range query arrives, it is first sent to "query transformation part (QT)". QT translates the range query into the bucket query. For example, suppose a query is "Find all the objects in range R". After translation, the query becomes: "Find all the buckets that intersect with R". Each bucket in the database has one of three following statuses:

1. The bucket does not intersect with the query range. In this case, the objects in the bucket can not be in the query range.
2. The query range covers the bucket. Then, the objects in the bucket must be in the query range.
3. The bucket intersects with the query range. This case is somewhat more complicated. The database now has no ability to distinguish which objects in the bucket are in the range and which are not. There are two possible methods. Either, by using statistics, gives an approximated result. Many works have been done in this field [PIH+96, APR99]. Or retrieves all the objects in bucket and sends their id to LPs. Then, LPs check the objects' latest position to see whether or not they are really in the query range and report the result. We adopt the later approach.

In Figure 2, the italic words show the indexing procedure and the bold words show the query procedure.

Example 3.2 (Traditional structure) *Our structure is able to use one LP to simulate the traditional design. Define a hash function $f(p) = 0$ for each object p. Then, the database only have one bucket and this bucket intersects with any query range. According to our design, The LP later rechecks each object and reports the result. In this simulation, our database becomes a dummy and the LP does all the work.*

The structure we designed is very flexible, by choosing different hash functions, we have different methods. In the next section, we present four different methods based on this structure.

4 Hash functions

This section presents four different hash functions and gives detail description of corresponding methods. The first function is based on space split method. After checking its performance, we find two main drawbacks. The next two methods are our solutions to solve them. The last method put these ideas together.

4.1 Overlap-free Space Partition Method

One of the main objectives of using hashing technique is to decrease the number of database updates. The basic idea of the first method is that we partition the space into several parts. Each part maps to a bucket in the database. Only when an object leaves one part and goes to a new one, does the database do the update operation. The details are described below.

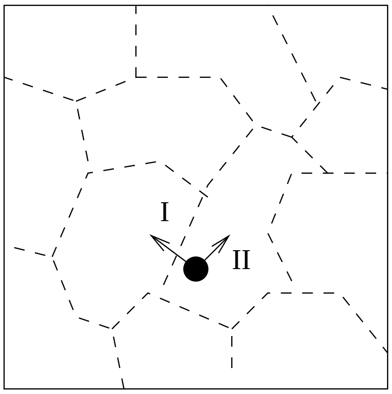

Figure 4: Example of object moving

The first step of this method is to partition the space into many small parts. In Figure 4, the square is working space. The dashed lines divide the space into 12 parts. The region covered by each part can be viewed as a bucket in the database. A perfect partition makes each bucket contains almost the same number of objects all the time. However, the partition work is done beforehand and we have no clues about how the objects move, it is very hard to find such a partition. In some special case, for example, objects are uniformly distributed and move randomly, an equi-size partition is almost perfect. Or if we know objects are moving close to some predefined locations, we could use Vorronoi diagram.

After partitioning the space, we give each part an unique id number. The hash function now is: $f(p) = i$ where p is an object and i is the id of the bucket p is in.

When at time t, an object leaves one part to another one, like path I in Figure 4, the LP which monitors this object sends an update request to database. The request is like $update(part\_id, old\_bucketid, new\_bucketid, t)$. Sometimes, after an object changes its location, it still stays in the same part, like path II in Figure 4, by our design, the database does not know that kind of changes.

The query part is very intuitive after f is defined. For *ad hoc* query, the database passes the query to the LP which monitors the object. The LP fetches the object's current location and submit the result. For range query, the procedure is just like what we discussed in the last section. One important thing we want to mention is that the buckets information are static in this case, which means the information (size, location, etc.) of the buckets never changes once the hash function f is given. This allows us using some existing spatial index structure (R*-tree, Quad-tree) to manage the buckets in the database. This will greatly accelerate the database query procedures.

An important issue needed to be considered is the size of the parts. In one hand, if we partition the space into very big parts, objects are more likely to move within a part. That means less update requests reaching database. However, at the same time, when executing a range query, the buckets have more possibility to be hit which causes more objects check and more communication between the database and the LPs. On the other hand, if the bucket size is too small, although the query may be faster (the database is able to finish most of the processing and less objects need to be further checked by the LPs), the cost of managing the buckets in database becomes very expensive. The database will be overflow by enormous update requests which can not be filtered by LPs. So there is a tradeoff here. In later section, we will give a cost mode and discuss more about size selection.

4.2 Augmented Space Partition Method

Comparing to other methods, overlap-free space partition method generates less database update. However there are two drawbacks in this design. This subsection and the next one will discuss them separately.

In overlap-free space partition method, objects zigzag along the bucket border could generate big trouble. Since an area in working space uniquely maps to a bucket in the database, in the rest of paper, without confusion, these two have equal meaning.

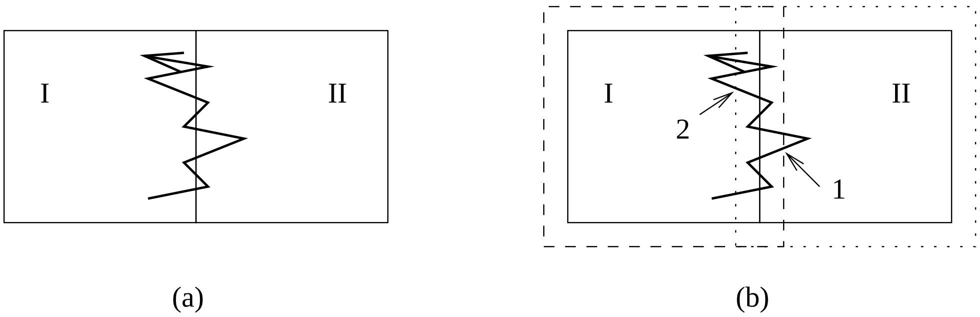

Figure 5: An object moves along the border

Look at Figure 5(a), I and II are two buckets. An object moves along the border. Anytime when it cross the border (from bucket I to bucket II or vice versa), an update request is generated by a LP. The object is first in bucket I. When finishes this path, there are totally eight database update requests.

In order to solve the problem, we increase the size of each bucket a little bit so that there is some overlap area between two buckets. An update request is generated only when an object leaves the augmented area. Look at Figure 5(b). The dashed square and dotted square are two augmented buckets. The object is first in bucket I (the dashed square). At point 1, it leaves bucket I and goes into bucket II (the dotted square), Then it moves in bucket II and at point 2, it goes back to bucket I. Now, following the same path, there are only two database updates.

The augmented space partition method works as following. First we generate a overlap-free space partition. Then we hash each object into buckets according to its original position. After that, each bucket does a δ-expansion. A δ-expansion means the center of the bucket does not change, but the cover area increases a small length δ. For example, if a bucket covers a rectangle $[x_0, y_0][x_1, y_1]$, after δ-expansion, the bucket now covers $[x_0 - \delta, y_0 - \delta][x_1 + \delta, y_1 + \delta]$.

In the next step, we need to find a hash function and store it to LPs. If we merely use the current location information of objects as input, there is a problem: when an object moves to an area covered by two buckets, the function can hardly decide which bucket it should be in. So in this method, we introduce an attribute called *previous_bucketid*. This attribute remembers which bucket an object is previously in. It is also sent to the hash function for decision. After each time period, if an object current stays in an area covered by only one bucket, everything is fine. Otherwise, if an area is covered by more than one bucket, the LPs first check if the object previously stayed in any of these buckets. If so, the LPs send no update request to the database. Otherwise, randomly pick one of these bucket for the object.

The rest part of indexing and the procedure of query is the same as those in overlap-free space partition method. We do not list the details here.

4.3 Quad-tree Hashing Method

The other draw back of overlap-free space partition method is that its buckets have no ability to modify its size or location after the original decision. The following example shows that sometime this may cause some trouble.

Figure 6 shows a skewed case. This time, we use an equi-size partition function and there are 16 buckets in the database. At time t_0, all the objects are in the upper-left corner. Then they start to move following the arrow. At time t_1, they all reach the lower-right corner. During t_0 and t_1, only few buckets may contain objects. (Only one in t_0 and t_1.) The others are empty. In this case, our index structure brings very little benefit for range query.

In order to find a solution for that, we then introduce some dynamic structures. The basic idea is dynamically change the cover area of each bucket. When a bucket contains too many objects, we split it into several smaller ones and reallocation the objects in the old bucket. On the other hand, if several buckets do not have enough objects, we merge them into one big bucket and put the objects together.

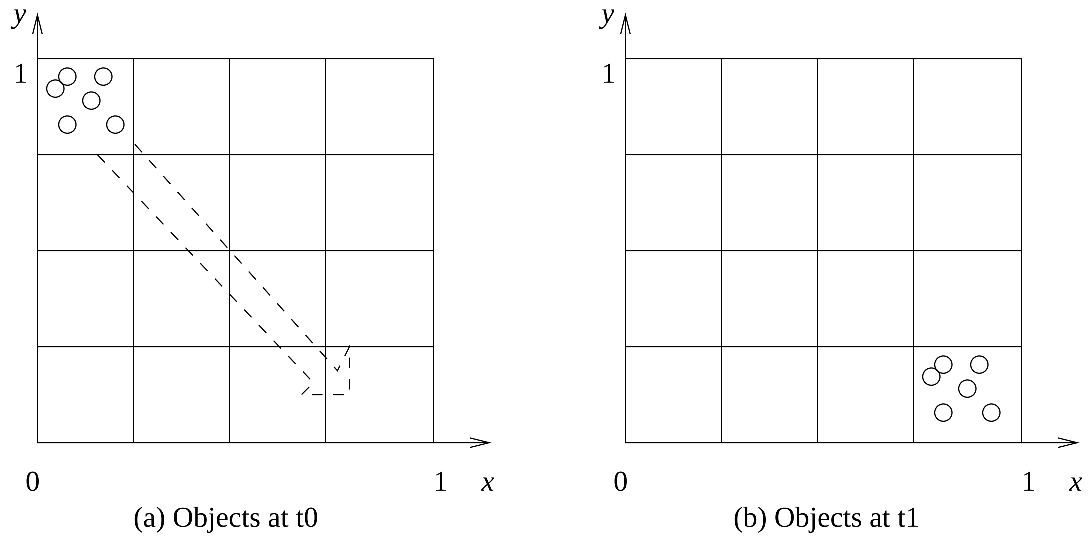

Figure 6: Skewed distribution of objects

Here is some design details:

- In the database, we create a new part called "Bucket Management Part (BM)". BM use a spatial index structure to manage the buckets. Any changes in buckets (such as add or remove an object in a bucket) will trigger an action in BM. BM checks the changing buckets and decide whether or not a split (merge) operation is needed.

- The spatial index structure we used is Quad-tree [Sam90] because it has simple structure and easy split and merge algorithm. Does not like R-tree, each inner node in Quad-tree has exact four children and no overlap is allowed between each node.

- The following information is stored in each node: the covered area, number of objects current inside the area, whether or not it is a leaf node, pointers to children if not leaf not, pointer to parent if not root node, etc. Each leaf node corresponds to a bucket in the database.

- When the number of objects in one leaf node is over an upper bound M, the node is split. We define M to be the maximum number of objects which can be stored in one disk page. The split algorithm first creates four children, each covers a quarter of the original cover area. Then it inform the database to generate four buckets. The objects in old buckets are then checked and reinserted into the new generated buckets. After that, the old buckets are deleted in the database, and the number of objects in new buckets are reported to Quad-tree and saved in new leaf nodes. Recursively do the split algorithm if one child still contains more nodes than upper bound.

- The trigger condition for a merge operation is a little more complicated. If a leaf node contains less than m objects, we still need to check how many objects in its sibling nodes. It is very possible that one of its sibling node are still dense. Also, since the merge node is very expensive, we do not want the newly generated node to be split very soon. So, in our algorithm, we define the condition as "a leaf node has less than m objects and the number of objects in its parent node is less than $3M/4$". When a node meets the condition, its parent node now becomes a new leaf node. The database generates a new bucket. The objects in old buckets are moved to the new one and the old buckets are simply deleted.

- Since the bucket structure now is dynamic, the LPs have to know the current structure in order to properly filter the update request. There are two alternations here. The first method is let the BM broadcast the index structure to the LPs after each structure change. This method is feasible only if the bucket structure does not change frequently or we do not have many LPs in the system.

  Another method is to cut the working space into even smaller units. The units can not be further split. Each bucket is a set of units. For example, we cut the two dimensional unit space into $2^{10} \times 2^{10}$ equal-sized units. By quad-tree definition, nodes which has a level less than 10 are squares which cover some units, and no unit is in two different Quad-tree nodes. The LPs know the size of a unit beforehand.When an object stays in the same unit, that means it is impossible to leaf a bucket. The LPs filter this kind of movement. If an object leaves an unit. It may still be in the same bucket or it goes to a new bucket. For this kind of movement, the LPs can not decide whether it should be filtered. So the LPs report them to BM, BM recheck the requests and do a second-time filtering.

  The split algorithm in BM need to be further changed while we adopt the later approach. When split a node, we need to do one more check to see if the node has the same size as an unit. If so, not more split.

The codes of split and merge algorithms are listed in Appendix A.

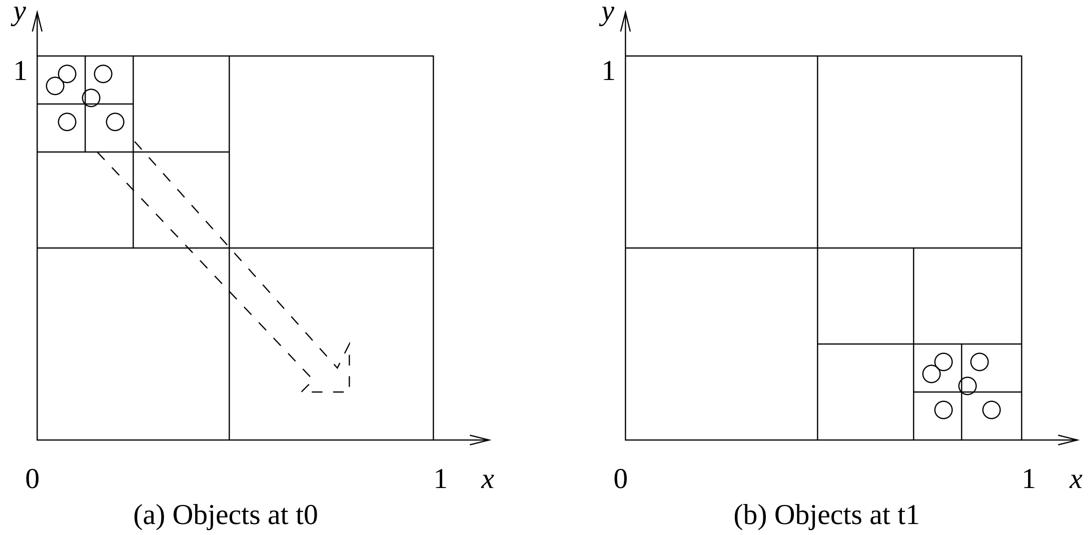

Figure 7: Dynamic buckets structures

Figure 7 shows what the buckets look like after we use the dynamics bucket structure.

When executing a query, the query range goes to the BM first. the BM uses the index structure to decide which buckets need to be further checked. The rest part is almost the same.

4.4 Extended Quad-tree Hashing Method

The last method combines the idea of quad-tree hashing method and augmented space-partition method. This time, we augmented each quad-tree node a little bit.

First we construct a Quad-tree based on the initial distribution of objects. Then like in augmented space-partition method, we execute a δ-expansion on each node (both inner nodes and leaf nodes). Due to the simple structure of quad-tree, after this step, the tree structure is still maintained, *i.e.* the area of each inner node still covers all its children.

As in augmented space-partition method, the EQ-tree uses expanded nodes in indexing. The indexing, insertion and deletion algorithms are almost the same as those in quad-tree hashing method. Therefore, we do not discuss too much about it.

In the last four subsection, we discussed four methods based on our new system structure. In the next experiment part, we will show more details about how to select an appropriate bucket size and compare the performance of these four methods in range query.

5 Experimental result

To access the merit, we wrote a simulation program and performed some experimental evaluation of different methods.

5.1 Experimental Setup and Data Generation

Since there are very few real data available in this field, we write our own data simulator based on a widely used benchmarking environment called "Generate Spatio-Temporal Data (GSTD)" [NST99]. Like in GSTD, our data simulator supports three initial object distribution: *uniform, zipf and Gaussian.* In our study, we only index and query the current status of moving objects. Therefore, we make the following changes:

- In our system, we use equal-length time interval for all objects, and each object reports its latest location after each interval. The global clock runs forever. In GSTD, the interval of each object are within a domain. And there is an upper bound for global clock. After the time passes the upper bound, all the objects are inactive.

- In our system, the objects move out of the working space are still considered active. Later they are still possible to move back into the working space. In GSTD, the out-of-bound objects are marked as inactive.

- The GSTD system are providing simulated data for trajectory indexing. So they have a very large log to record every movement of objects. In our system, we only need the current location information of each object, so we do not provide this log file. This allows us to run experiment with large number of objects over a very long period of time.

- Objects in our system can have different initial distribution and movement. For example, in our system, we allow half of the objects in skewed distribution and the other half are uniformed distributed. In GSTD, all the objects must have the same initial distribution.

We use Java language in our experiment due to its strong thread support. After each time period, each MovingObject calculate its new location. There are two LPs in the system. Each monitors half of the objects. After each time period, the LPs check the latest status of the MovingObjects and send the filtered information to the BM. The BM collects the information and reports the experiment result.

The machine we used is a Pentium II 300MHz machine with 128M memory. We use 20 bytes to represent each two dimensional object (2 doubles for location and 1 integer for id). The page size are 4K which allows upper to 204 2-d objects in one page. In both quad-tree methods, we choose M to be 200 and m to be 50.

Most other methods also index the history information for each object. Their setting makes it impossible to index large number of objects. So it is unfair to compare our method with theirs. Besides our four methods, we also use R-tree method. The R-tree method uses traditional static method in moving object problem. It maintains an R-tree in the database. After each time period, the R-tree is updated according to the latest location of objects.

5.2 Notation

The notation used in this section is summarized for easy reference in Table 1. Meanwhile, abbreviations of algorithms used in this part are in Table 2.

5.3 Cost for Index and Query

Before we run the experiments, we first study the factors which have important impact on total performance.

The total cost of index C_{index} mainly includes three parts: database update, bucket structure change, and the communication between LPs and the database. Other factors such as latest position collection, or LP updating do not cost much due to the parallel system structure. Therefore: $C_{index} = DU\# * C_u + C_b + C_c$.

v	Speed of an object
$\bar{v}$	Average speed of all objects
$\sigma(v)$	Standard deviation of speed
S	Bucket size
$DU\#$	Number of database update
$\sigma(D)$	Standard deviation of initial distribution
C_u	Cost for one database update, which includes a delete and an insertion operation
C_b	Cost for bucket structure change
C_c	Cost for communication between LPs and the database
C_q	Cost for database query

Table 1: Notation used in our experiment

RT	R-tree method
SP	Overlap-free space partition method
ASP	Augmented space partition method
QH	Quad-tree hashing method
EQH	Extended Quad-tree hashing method

Table 2: Algorithm abbreviations

In SP and ASP method, because of no bucket structure change, $C_b = 0$. For the other methods, since the bucket structures are maintained in memory, C_b mainly is the cost for moving objects from old buckets to new buckets.

The total cost of query C_{query} includes two parts. The first part is the total cost used in database query and the second part is the communication cost from the database to LPs. So $C_{query} = C_q + C_c$.

5.4 Datasets

The dataset consists of 100,000 objects in working space. We studies the performance of various methods on two initial object distributions and two movement types, which are described below:

5.4.1 Two initial distribution types

Although like in GSTD, we implement three initial distributions, we found only two of them are useful in real-life applications. The first one is uniform distribution. In this case, objects are uniformly distributed in working space. the second one is Gaussian distribution. This time, objects are clustered around one or several central points. This distribution can be viewed as skewed one. Also, in our simulation, we allow combinations of these two methods. (For example, half of the objects are uniformly distributed and the other half are in Gaussian distribution.)

When we use Gaussian distribution, we set the $\sigma(D)$ to be 0.1 in the unit space.

5.4.2 Two movement types

GSTD method are very powerful in describing the objects' movement. In our program, we borrowed their ideas. We totally defined two movement types for our experiment. The first one is random movement. The settings are in Table 5.4.2. The detail meaning of each parameter can be found in [NST99].

The second movement type which we called directed movement is used to simulate the objects moving the lower-left corner to the upper-right corner. The settings are in Table 5.4.2. This

$\bar{v}$	$\sigma(v)$	$max_x(speed)$	$max_y(speed)$	$min_x(speed)$	$min_y(speed)$
0	0.005	0.005	0.005	-0.005	-0.005

Table 3: Settings for randomly moving objects

movement type is useful when we study the car's movement in traffic hour.

$\bar{v}$	$\sigma(v)$	$max_x(speed)$	$max_y(speed)$	$min_x(speed)$	$min_y(speed)$
0.005	0.005	0.01	0.01	0	0

Table 4: Settings for directed moving objects

5.5 Query Sets

The query sets consists of 1000 rectangles lying within the working space. We choose the centers of the rectangles randomly. The size of the query rectangle was 1% of the total area. At the end of each time period, one query is randomly selected form the query set and applied in the current object distribution.

5.6 Experimental Results

In the first experiment, we compare the performance of various techniques for both indexing and querying.

5.6.1 Experiment 1: Impact of Bucket Size

In this section, we want to study the index performance under different S and $\bar{v}$ in SP method.

We define the hash function to be $f(x, y) = i * int(y * i) + int(x * i)$. By changing the value of i, we will have $i \times i$ equal-size square buckets. Then we fix $\bar{v}$ to be 0.005 and record the performance under different i value. The result in Figure 8. shows that the performance is proportional to i. In Figure 9, we fixed S and find that the performance also is proportional to $\bar{v}$. So the main conclusion

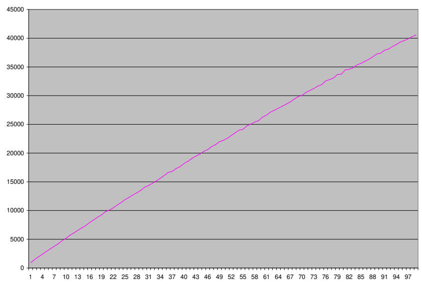

Figure 8: Performance vs. S, $\bar{v} = 0.005$

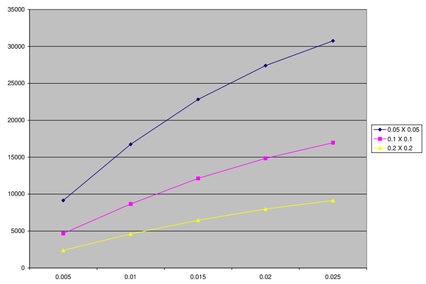

Figure 9: Performance vs. $\bar{v}$

in this experiment is:

$$DU\# \propto \frac{\bar{v}}{\sqrt{S}}$$

.

5.6.2 Experiment 2: Impact of Overlap Size

In this experiment, we look at the impact of overlap size in ASP method. The bucket size is fixed as 0.1×0.1. Figure 10 shows the average number of database update for different overlap size. The x value is the size of expansion on each side, and y value is the total database update number. Two curves show the results under different $\bar{v}$.

At a high level, it is clear that allowing overlap between buckets help considerably. This could filter many database update requests generated by objects near the bucket border. Note that the curves decrease very fast at the beginning, then they slow down in both cases. And the phenomenon is more obvious when $\bar{v}$ is slow. Our suggestion is to set the overlap size to be $\bar{v}$. For example, if $\bar{v}$ is 0.005 in one case, in ASP, we should make a 0.005-expansion on each bucket.

5.6.3 Experiment 3: Study of index performance

In this experiment, we want to find out the index performance of different methods in various initial distribution and movement types. We count the number of database update and the disk pages used to store all the data. Here, the number of database update includes two parts: database

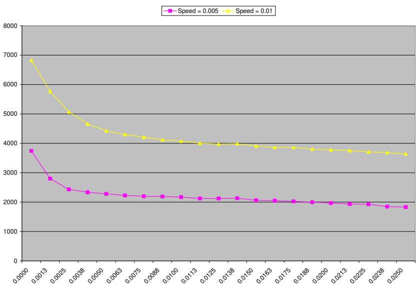

Figure 10: Impact of Overlap Size

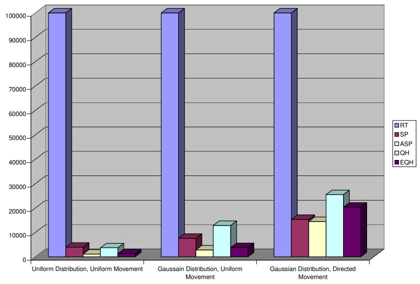

Figure 11: Number of Database Update

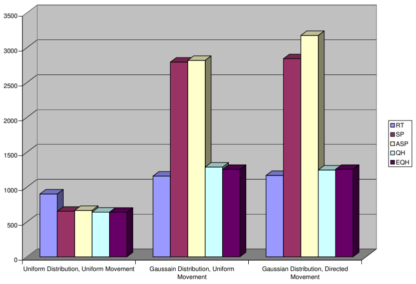

Figure 12: Number of Disk Page Used

update request generated by LPs and the update operation when merge and split buckets. Figure 11 and 12 list the result.

Some facts which can be observed from the figures:

- RT method updates the location of all the objects after each time period. Therefore, the number of database updates is the same as the number of objects. The other four methods are much better than RT method.

- When the initial distribution of objects is uniform. The performance and QH is the same as SP. This is normal because when the objects are uniformly distributed, the quad-tree is very balanced and the leaf nodes have the same height. Therefore, the leaf nodes (buckets) have equal size. The whole structure turns to be a equi-size space-partition in SP method. For the same reason, the performance of ASP and EQH are equal.

- In ASP method, we set the overlap area to be as big as $\bar{v}$, we found that the total database update is about 60% to 70% of that in SP. The improvement is huge.
- When the objects are in Gaussian distribution, The number of database update of QH is bigger than that in SP because the following reason: In SP, each bucket has same size. No matter where the object is, it always has the same possibility to cross the buckets border and generate a database update. However in QH, quad-tree is used. Each leaf node in quad-tree, which is a bucket in the database, can not have more than M objects. Since the objects are near the center of working space, the size of buckets near the center now is much smaller than that of buckets outside. So the objects in these buckets are more likely to cross the bucket border and generate database updates. The other reason is that the bucket update in QH (merge and split operation) generates extra database update (although not too many in this experiment). For the same reason, EQH generates more database update than ASP.
- The benefit of QH and EQH is their storage efficiency. In Figure 12, we can find that in uniform distribution, the disk page used in all four methods are almost the same. However, when the initial distribution is skewed, QH and EQH use about half of disk pages in SP and ASP because in QH and EQH, there is a bucket management part which merges the buckets with small number of objects.
- When the movement type is set to be directed, the system triggers a lot of merge and split operation in QH and EQH. This generates extra database updates. Therefore, in QH and EQH, total number of database update is much bigger than the result in uniform movement. However, the storage efficiency is still kept.

5.6.4 Experiment 4: Comparison on Query Performance

In this part, we want to test the query performance of all our four methods. We collect two sets of data in experiments: disk pages accessed and objects checked in LPs. The first one is normally used to show the performance of a method, and the second one for communication cost.

In uniform distribution, as in index phase, the performance of SP and QH are almost the same. So are ASP and EQH. However, in Gaussian distribution, the QH and EQH methods check about

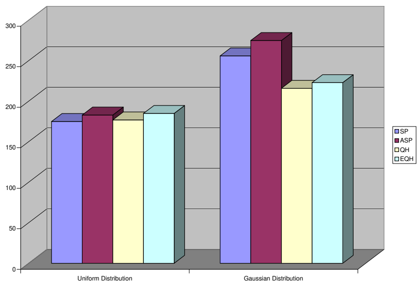

Figure 13: Number of disk pages checked

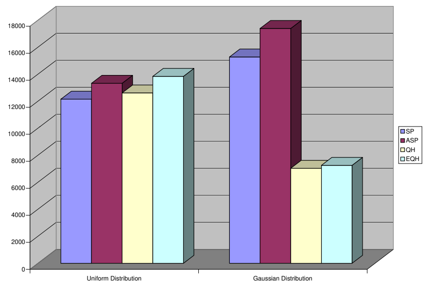

Figure 14: Number of objects checked by the LPs

30% less disk pages, and the communication cost is less than half of that in SP and ASP. This is the original design objective for these two methods. And EQH is a little worse than QH because after expanding the index tree node, the possibility for each node to intersect with the query area increases a little bit. This increases the query cost. However the different is very small.

5.6.5 Discussion and Method Selection

The first discussion is whether we need to expand the buckets, such as use ASP instead of SP or use EQH instead of QH. Our answer is yes. The benefit to doing this is huge: about 30% - 40% of the database update request is filtered. And the cost is small: the query is a little slower after that. The best size of overlap area should be the same as $\bar{v}$, which, in most cases, is very small.

Then, should we dynamically change the bucket structure? It depends. The static bucket structure (in SP and ASP) is easy to implement. Each LP only need to remember a hash function which is given at the beginning. Also, the structure works well in uniform cases. The dynamic structure (in QH and EQH) is a little bit harder to implement. Extra cost of dynamic bucket structures includes: a tree structure maintained in memory, extra communication cost between LPs and the database, etc. However, it answers query efficient.

Our suggestion is to use ASP method if the distribution is not very skewed and the query load is not high. Otherwise choose EQH method.

6 Conclusion and Future Works

In this paper, we studied the moving object problems. The main technique we used is called hashing technique which allows the database simply save the bucket information of each object instead of many details. Meanwhile, we gave four hashing methods built on the new system structure which is designed for the technique. The experiment showed that methods based on hashing technique generated far less database updates than static index method. This makes it possible to index and manage huge number of moving objects later.

The future researches include the following. Current, we can only index and query point object. In the next step, we want to find a way to index rectangle data and the object which changes shape (such as in monitoring forest fire, etc). Also, we plan to do some research on spatial join between moving objects and static objects. Lo, et. al. propose hash-join in static spatial environment [LR96]. We will borrow some of the ideas and use them in dynamic environment. The third thing is that we want to find more hash functions based on our system structure.

Acknowledgments: The authors would like to thank Samir Khuller for his helpful advice.

References

[APR99] S. Acharya, V. Poosala, S. Ramaswamy. *Selectivity Estimation in Spatial Databases* Proc. of SIGMOD 1999.

[ArcV98] *ArcView GIS* ArcView Tracking Analyst, 1998.

[BBB87] R. Bartels, J. Beatty, B. Barsky. *An Introduction to Splines for Use in Computer Graphics & Geometric Modeling* Morgan Kaufmann Publishers, Inc., 1987.

[BBK98] Stefan Berchtold, Christian Bhon, Hans-Peter Kriegel. *The Pyramid-Technique: Toward Breaking the Curse of Dimensionality* Proc. of the ACM SIGMOD, 1998.

[BKK96] Stefan Berchtold, Daniel A. Keim, Hans-peter Kriegel. *The X-tree: An Index Structure for High-Dimensional Data* Proc. of VLDB, 1996.

[BKS+90] Bechmann N., Kriegel H.P., Schneider R., Seeger B.. *The R*-tree: An Efficient and Robust Access Method for Points and Rectangles* Proc. of the ACM SIGMOD, 1990.

[CG99] Surajit Chaudhuri, Luis Gravano. *Evaluating Top-k Selection Queries* Proc. of VLDB, 1999.

[Gut84] A. Guttman. *R-Trees, A Dynamic Index Structure for Spatial Searching* Proc. of the ACM SIGMOD, 1984.

[KGT99] G. Kollios, D Gunopulos, V. J. Tsotras. *On Indexing Mobile Objects* In Proc. of PODS, 1999.

[LJF95] Lin K, Jagadish H. V., Faloutsos C. *The TV-Tree: An Index Structure for High-Dimensional Data* Proc. of VLDB, 1995

[LR96] Ming-Ling Lo, Chinya V. Ravishankar. *Spatial Hash-Joins* Proc. of the ACM SIGMOD, 1996.

[NHS84] J. Nievergelt, H. Hinterberger, K.C. Sevik. *The Grid File. An Adpatable, Symmetric Multikey File Structure* ACM Transactions on Database Systems, Vol. 9, 1, 1984.

[NST99] M. A. Nascimento, J. R. O. Silva, Y. Theodoridi. *Evaluation of Access Structures for Discretely Moving Points* Intl. Workshop on Spatio-Temporal Database Management (STDBM'99), Edinburgh, UK, September 1999.

[PIH+96] V. Poosala, Y. E. Ioannidis, P. J. Haas, E. J. Shekita. *Improved Histograms for Selectivity Estimation of Range Predicates* Proc. of SIGMOD 1996.

[PJ99] D. Pfoser, C. S. Jensen. *Capturing the Uncertainty of Moving-Object Representations* Advances in Spatial Databases, 6th International Symposium, SSD'99, Hong Kong, China, July 20-23, 1999.

[PTJ99] D. Pfoser, Y. Theodoridis, C. S. Jensen. *Indexing Trajectories of Moving Point Objects* Chorochronos Technical Report, CH-99-3, October, 1999.

[PTJ00] D. Pfoser, Y. Theodoridis, C. S. Jensen. *Novel Approaches in Query Processing for Moving Objects* Chorochronos Technical Report, CH-00-3, February, 2000.

[Sam90] H. Samet. *The Design and Analysis of Spatial Data Structures* Addison-Wesley, Reading, MA, 1990.

[SK98] Thomas Seidl, Hans-Peter Kriegel. *Optimal Multi-Step k-Nearest Neighbor Search* Proc. of SIGMOD 1998.

[SR99] Zhexuan Song, Nick Roussopoulou. *Hashing Technique: A Framework for Indexing High Dimensional Data* Technical Report, CS-TR-4059, University of Maryland, 1999.

[SJL+99] S. Saltenis, C. S. Jensen, S. T. Leutenegger, M. A. Lopez. *Indexing the positions of Continuously Moving Objects* CHOROCHRONOS Technical Report CH-99-19, 1999.

[SWC+97] A. P. Sistla, O. Wolfson, S. Chamberlain, S. Dao. *Modeling and Querying Moving Objects* Proc. of ICDE 1997.

[TJ98] Nectaria Tryfona, Christian S. Jensen. *A component-Based Conceptual Model for Spatiotemporal Applications Design* CHOROCHRONOS project, technical report CH-98-10, 1998.

[TUW98] J. Tayeb, O. Ulusoy, O. Wolfson. *A Quadtree Based Dynamic Attribute Indexing Method* The Computer Journal, 41(3), 1998.

[WCD+98] O. Wolfson, S. Chamberlain, S. Dao, L. Jiang, G. Mendex. *Cost and Imprecision in Modeling the Position of Moving Objects* In Proc. of ICDE, 1998.

A Algorithms in Quad-tree Hashing Method

In this appendix, we list the code of our algorithms used in quad-tree hashing mehtod.

```
addObject (MovingObject mo) {
  if (! area.isIn (mo.getLocation ()))
  // if mo is not in the area of this node
    return;
  numberOfObjects ++; // update the number of object in this node
  if (isLeaf) { // leaf node
    update the bucket in the database;
    if (numberOfObjects > M && level < MAXLEVEL)
    // level check gurantees that unit can not be split further
      split ();
  }
  else { // inner node
    for (each child c)
      if (c.area.isIn (mo.getLocation ()))
        c.addObject (mo);
  }
}

split () {
  generate four nodes;
  isLeaf = false; // now becomes an inner node
  create four new buckets in database;
  for (each objects mo in this bucket)
    addObject (mo); // the recursive call allows further split
  clear current buckets;
}
```

Figure 15: Add an object in quad-tree hashing method

```
remove (MovingObject mo) {
  if (! area.isIn (mo.getLocation ()))
    return;
  if (isLeaf) {
    remove the object in the bucket
    for (QtreeNode node = this; node != null; node = node.parent)
      node.numberOfObjects --;
      // update the number attribute until to the root
    if (numberOfObjects < m && parent != null) // not root, consider merge
      parent.merge ();
    return;
  }
  else { // inner node
    recusive call the remove for each child;
    stop when one child deletes the object;
  }
}

merge () {
  if (numberObObjects < 3 * M / 4) { // the second condition checks here
    create a bucket in the database;
    removeAllChildren (this);
    // put all the objects in this node into the new bucket
    isLeaf = true;
    if (numberObObjects < m && parent != null)
      parent.merge (); // recursive check the upper level
  }
}
```

Figure 16: Delete an object in quad-tree hashing method