A Quadtree Based Dynamic Attribute Indexing Method*

Jamel Tayeb and Özgür Ulusoy
Department of Computer Engineering and Information Science
Bilkent University
Bilkent, Ankara 06533, TURKEY

Ouri Wolfson
Electrical Engineering and Computer Science Department
University of Illinois
Chicago, Illinois 60680

Abstract

Dynamic attributes are attributes that change continuously over time making it impractical to issue explicit updates for every change. In this paper, we adapt a variant of the quadtree structure to solve the problem of indexing dynamic attributes. The approach is based on the key idea of using a linear function of time for each dynamic attribute that allows us to predict its value in the future. We contribute an algorithm for regenerating the quadtree-based index periodically that minimizes CPU and disk access cost. We also provide an experimental study of performance focusing on query processing and index update overheads.

1 Introduction

The so-called dynamic attributes [19] arise primarily in mobile data management [10]. This field was sparked by the recent technological progress in portable laptop computers and the smaller palmtop computers (also called *Personal Digital Assistants* or PDAs). Moreover, their ability to communicate with fixed hosts via the wireless medium leads to the new paradigm of nomadic computing and its associated implications on data management issues [1]. It is speculated that in the near future such devices will be ubiquitous thus challenging

*This research is supported by the Research Council of Turkey (TÜBİTAK) under grant number EEEAG-246 and the NATO Collaborative Research Grant CRG 960648.

the database community to deal effectively with two of their most intrinsic characteristics, namely, scale and mobility.

Given that mobility is the most distinguishing feature of the mobile computing paradigm, it is only natural that location becomes a central piece of information. It gives rise for example to a new kind of queries called *location-dependent queries* [9] for which the computed answer depends on the location of the user or object which issued them. Consider a person driving a car who occasionally wants to be informed about motels that are within five miles of his location in order to select a reasonably priced hotel. It is clear that the set of motels computed as an answer to his query would be different each time his car moves by a reasonable distance. It is also clear that we cannot afford the cost of updating the driver's location 'continuously' to be able to answer his queries. In a typical mobile architecture, mobile users will be registered into a special set of servers called *Mobile Support Stations* which manage their connection session and interface them to the fixed network that contains useful data. Each user or object will have a (temporary) record in a support station and the location attribute will be one field of that record. In the driver example above, the driver's position is the archetype of a dynamic attribute.

In this paper, we propose an indexing technique for dynamic attributes based on a variant of the quadtree data structure in which the indexing directory is in primary memory and the indexed data resides in secondary storage. The method is useful in any application which involves data items whose value varies continuously according to a given function of time (temperature is another example). The general approach proposed in [19] is to have every moving object supply its position and motion equation upon registering to the system. Thereafter, the object may occasionally send a request to update its position or its motion equation or both. It can do that relatively infrequently and is constrained only by the amount of error introduced as a result of using an approximate motion function over a long period of time [22]. We then plot the value of every dynamic attribute as a function of time in the two-dimensional time-attribute space. In this way, we have reduced or transformed the problem of indexing dynamic attributes into a spatial indexing problem albeit with a different flavor.

Our focus is on supporting range queries where the range can be an attribute or time range. Furthermore, we support queries on the moving objects themselves rather than queries issued by the moving objects. A typical example is: Give me all the objects whose attribute value falls in the range $[a_{begin} \ldots a_{end}]$. Such a query would be useful in vehicle monitoring systems and possibly in intelligent vehicle navigation systems [6].

Starting from the equations of motion and their corresponding attribute trajectories plotted in the time-attribute space, we generate periodically our quadtree-based index to support queries about the future. The idea is to destroy and reconstruct the index at the end of every period. In more abstract terms, given the infinite time dimension, we partition it into equal-sized time slices and create an index for each time slice. Theoretically, the union of these indices is the master index of the whole time-attribute space being indexed. Practically, when the period of an index is over it is disposed of and the next one is generated since storage space is finite. We will denote this period by ΔT. Since our index is reconstructed every ΔT time units, we will also call ΔT the *Index Reconstruction Period.* The time span covered by any single index reconstruction period is called a *session.* Conceptually, a session is just a projection of the system state over a finite time interval due to the limitations of disk space. We note that our index is a dynamic one allowing for insertions, deletions, and updates inside periods. Updating the position or the equation of motion of an object requires deleting the records relating to its previous state and inserting new index elements according to its new record. Furthermore, inside a session, an object's equation of motion is assumed to remain valid until an explicit update request is issued by the corresponding user.

For the proposed method, we contribute an index reconstruction algorithm that is optimal in CPU and disk access overheads. We have also conducted a simulation study which shows good query processing performance. The outline of the paper is as follows. In Section 2, we introduce the background information necessary for the rest of the paper. We then describe the indexing method in Section 3. Section 4 presents the experimental setup of the conducted performance study. Section 5 presents the storage requirements of our quadtree based index and Section 6 provides a mathematical analysis of storage utilization. We describe an optimal index reconstruction algorithm in Section 7 after presenting the naive algorithm which motivated its conception. In Section 8, we present query processing performance for two types of supported queries. Section 9 includes a brief digression on related work. Finally, we outline possible areas of future work and conclude the paper in Section 10.

2 Background

Our work is largely based on the ideas introduced by Sistla et. al. in [19]. The authors present a new data model suitable for representing moving objects in database systems. The model is called the *Moving Objects Spatio-Temporal* (MOST) data model and relies on the key idea of representing the position as a function of time. We thus start with N linear equations or functions of time $f_i(t) = a_i \times t + b_i$ $(0 \leq i < N)$ where N is the total number of objects in the system hereafter denoted by *system size.* For the two dimensions of motion, $2N$ equations will be needed, two for each object. Sistla et. al. propose to represent a dynamic attribute A by three subattributes $A.value$, $A.updatetime$, and $A.function$. The dynamic attribute would then take the value $A.value$ at time $A.updatetime$ and the value $A.value + A.function(t)$ at time $A.updatetime + t$. The value of a dynamic attribute at any point in the past is thus being used in conjunction with the supplied function to predict its value at any point in the future.

The slope a in the approximate linear equation $f(t) = at + b$ corresponds to the rate of change of the dynamic attribute. When this attribute is the position of a moving object with respect to a predefined coordinate system, this slope is simply the speed of the object along one of the axes. We remark that for objects moving in two-dimensional space, we will have to index two distinct dynamic attributes; the abscissa and the ordinate attributes. This is necessary to be able to support two-dimensional range queries and leads to two different indices. Answers to two dimensional queries are then taken as the intersection of the two answer sets corresponding to the two unidimensional range queries issued separately over the abscissa and the ordinate spaces. In the rest of the paper, we shall use the word speed to mean the slope a of the motion equation. We define the average speed $\overline{v}$ to be the average of a large set of speed values as computed over a reasonably reliable number of sessions. The quantity $\overline{v}$ turns out to have an impact on the nature of our indexing problem. Let ΔA denote the total length of our indexed attribute space. We assume that ΔA is finite[1]. The relative value of $\overline{v}$ compared to ΔA is more important than its absolute value. We capture this observation in a new parameter which we call *speed ratio* defined as follows.

Definition 1 *: The* **speed ratio** α *of the set of* N *dynamic attributes being indexed is the attribute distance an object moves on the average in a single time unit relative to the total length* ΔA *of the attribute space. It is given by the formula*

$$\alpha = \frac{\overline{v}}{\Delta A}$$

The speed ratio is an intrinsic property of the system of objects. Conceptually, α is an indication of the dynamism of our system; the higher it is the more agitated the objects are while the lower it is the more sluggish the overall system becomes. We then introduce in more detail the types of queries supported.

In [19], three types of queries are discerned for the MOST data model. These are the *instantaneous*, the *continuous*, and the *persistent* queries. An instantaneous query submitted at time t_i is processed against the database state at t_i. The following example is given in [19]: 'Display the motels within 5 miles of my position.' A continuous query submitted at time t_i is processed against all database states starting from t_i (i.e., $[t_i \ldots \infty)$). It is described as 'an instantaneous query being continuously reissued at each clock tick.' In the motels example, the user will just require to be 'continuously' informed about which motels are coming within 5 miles of his position. If we let S_j denote the state of the database at time t_j, then at each $t_j > t_i$, the continuous query is reevaluated against S_j. The persistent query is a bit more demanding. Like the continuous query, it has to be evaluated at each clock tick after its time of submission t_i. However, unlike the continuous query, at t_j it has to be evaluated against the set of states $S_i, S_{i+1}, \ldots, S_{j-1}, S_j$ rather than

[1] Even if ΔA was not finite, in practice, a finite representation or approximation of the attribute space being indexed imposes itself.

against S_j alone. The example query provided in [19] is the following: 'let me know when the speed of object o in the direction of the x-axis doubles within 10 minutes'. Persistent queries arise in the expression of temporal triggers in active database applications [18]. In our research, the focus is on supporting instantaneous and continuous queries. Adapting the indexing method proposed here to handle persistent queries is left for future research. As mentioned above, our focus in this paper is on range queries which ask about the moving objects themselves. The generic and generalized form of our queries is the following:

> Give me all the objects whose value for attribute A falls in the attribute range $[a_i \ldots a_j]$ at some time between time instances t_b and t_e.

Since we have both an attribute and a time range, we obtain two-dimensional range queries. Let t_{now} denote the time at which the query was submitted. Depending upon the values for t_b and t_e, we may have any of the following four types of queries:

1. If $t_b = t_e = t_{now}$, we have an instantaneous range query asking about the present.
2. If $t_b = t_e = t_i$ and $t_i > t_{now}$, we have an instantaneous range query asking about the future.
3. If $t_b = t_{now}$ and $t_e = \infty$, we have a continuous range query.
4. If $t_b = t_i$ and $t_e = t_j$ ($t_{now} \leq t_i < t_j$), we have a general two-dimensional range query over the attribute and time dimensions.

In practice, we map the first three types of queries to the fourth one which is more general. In instantaneous range queries (cases 1 and 2), we approximate the time point t_i by the time range $[t_i - \delta t \ldots t_i + \delta t]$ where δt is a small time lapse. In continuous queries, the theoretically infinite range $[t_b \ldots \infty)$ is usually decomposed into the set of contiguous intervals

$$[t_i \ldots t_i + \Delta T], \ldots, [t_i + n\Delta T \ldots t_i + (n+1)\Delta T], \ldots \tag{1}$$

This again reduces it to the fourth case. If a continuous query arrives in the middle of a session, its answer is computed over the remaining time until the end of the current session. At the beginning of each session, the answers to continuous queries are computed in batch mode and sent to their corresponding recipients either incrementally (according to when objects enter the queried range) or in one packet (in which case it is the responsibility of the mobile computer to present it properly to the human user). We then describe the indexing method.

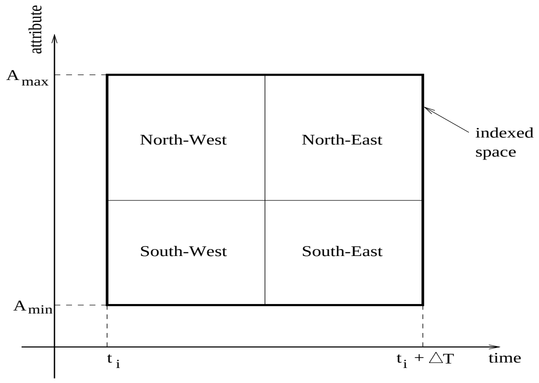

Figure 1: Partitioning of the indexed space in the region quadtree.

3 The Index

As we mentioned above, since we can plot the objects' trajectories in two-dimensional space, the problem of dynamic attribute indexing is transformed into a spatial indexing problem. For this, we could draw upon the literature for spatial access methods [15] and adapt one access method to our specific problem. We have selected the quadtree indexing structure.

The quadtree is treated thoroughly in [14] with several of its variants. The idea common to all quadtree variations is the recursive decomposition of indexed space. However, we are interested in the so called *region quadtree* which is based on the successive subdivision of space into four equal-sized quadrants. This is shown in Figure 1. Among the region quadtree variants, we are particularly interested in the *PMR quadtree* which is the quadtree-based indexing structure for line segments [15]. The idea of the PMR quadtree is to store information about a line segment in every quadrant of the underlying space that it crosses (fully or partially). The data space is partitioned until no more than B lines cross a single quadrant; B is called the *bucket size*. Typically, B will be equal to the number of data records that fit in a single disk page. In our work, we will be using an adaptation of the PMR quadtree in which vertex information is embodied in the object's equations of motion. On the other hand, this makes it similar to the bucket PR quadtree for indexing points [15]. The bucket PR quadtree indexes points by recursively partitioning the underlying space until no more than B points fall in a single quadrant. The difference in our case is in the semantics of a data point (it codes the information that a line crosses a quadrant) which makes the split involve more than a simple distribution of points over the four subquadrants.

At the level of indexed data, index records consist of the object ID, the intercept b, and slope a of the corresponding equation of motion $f(t) = at + b$. The slope a can be both positive or negative. Its sign corresponds to the direction of motion when its magnitude corresponds to the speed of a moving object.

Intercept b may or may not take negative values depending on the application.

When a data page overflows (at the $B + 1^{th}$ insertion), a page or *bucket split* takes place. Bucket splits involve the following operations. We take the (a, b) pair of each object in the bucket and use it in conjunction with the quadrant boundaries X_{min}, Y_{min}, X_{max}, and Y_{max} to find out which of the four subquadrants the trajectory crosses. We then insert the corresponding $< ID, a, b >$ record in every crossed subquadrant (there can be at most three out of four). A bucket split requires that we allocate four new disk pages, one for each subquadrant. Upon completion of bucket split computations, the parent data page is disposed of. The leaf quadtree node that was pointing to it must itself allocate memory for four leaf nodes which will point to the newly allocated disk pages. It then becomes an internal node. Furthermore, the boundaries of the four subquadrants have to be computed from the boundaries of the parent quadrant. The example at the end of this section will further illuminate the operation of the quadtree index in our application.

Given a constructed index, an object ID, and the associated function, we would execute a search for all index elements belonging to the object's trajectory in the following manner. The linear equation of motion $f(t) = at + b$ is our search key and the object ID is used once we reach the relevant data pages. Starting at the root of the quadtree, we use the pair (a, b) and the boundary fields X_{min}, Y_{min}, X_{max}, and Y_{max} found in the root to decide which subquadrants are crossed by the object sought. We then descend the next level and repeat the same thing until we reach the leaves. The disk page pointer field is then used to bring the relevant data pages into the buffer (if they are not there). The search procedure is the basis of insertion, deletion, and update operations. Traversal of the quadtree to answer range queries is done in an analogous manner except that the search key consists of the boundaries of the queried range. Moreover, when this range overlaps more than one subquadrant, the recursive step of the search must decompose it by computing the boundaries of its associated subranges. We are now ready to describe the performance of the index. We start by describing the experimental setup of the study we conducted.

It is possible in theory to have an infinite sequence of bucket splits. This happens when all the $B + 1$ trajectories intersect at a grid point. It is very unlikely that they intersect at any point (in our application $B = 340$) and even less likely that this point happens to be a grid point. Nevertheless, when it happens we propose to make use of overflow buckets and thereby split the overflowing bucket only once. This is done as follows. Because the grid point at which the $B + 1$ trajectories meet belongs to exactly one of the four subquadrants, the other three will not have this problem. for the fourth quadrant, allocating an overflow page is enough to solve the problem[2].

[2] The use of overflow pages also proves useful in index reconstruction as we shall see later.

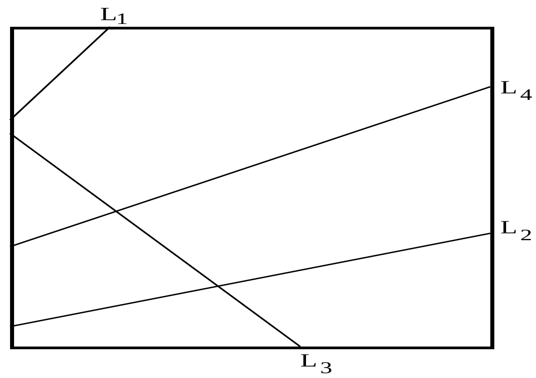

Figure 2: Four object trajectories crossing the indexed space.

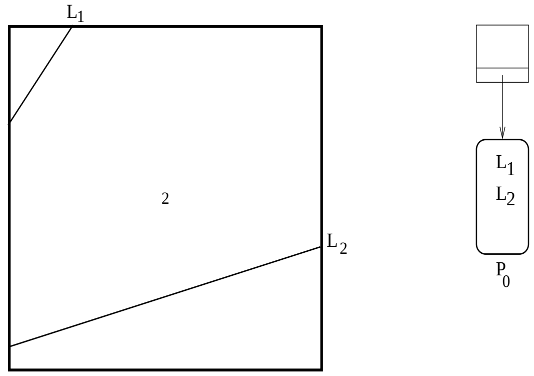

Figure 3: The indexed space and the index tree after insertion of object trajectories L_1 and L_2.

3.1 An Example

We illustrate the operation of the quadtree index by means of an example depicting the successive insertion of four object trajectories into an initially empty tree. The trajectories are denoted by L_1, L_2, L_3, and L_4. The position and orientation of these trajectories relative to the indexed space is shown in Figure 2. trajectories L_1, L_2, and L_4 have a positive slope while L_3 has a negative slope. To be able to illustrate splits and space partitioning, we chose a small bucket size $B = 2$; actual bucket size is much bigger than this (a few hundreds).

We begin by inserting L_1 and L_2. The situation after the insertion is shown in Figure 3. A leaf node is depicted by a square shape with a rectangular slot at the bottom containing a pointer to the data page on the disk. The quadtree index is then simply a single leaf node pointing to the full data page containing index records for L_1 and L_2. Data pages are pictorially depicted by a vertical rectangle with smooth corners. The number two inside the quadrant in Figure 3 indicates the number of trajectories crossing it or alternatively the number of index points stored in its corresponding data page on the disk.

Next, we insert trajectory L_3 into the quadtree. Since page P_0 was full before insertion, a bucket split takes place accompanied by partitioning of the underlying indexed space. The situation after this insertion is shown in Figures 4 and 5. The root now becomes an internal node pointing to four leaf nodes corresponding to the south-west, north-west, north-east, and south-east respectively. The north-east subquadrant remains empty as it is not crossed by any of the three trajectories. The other three are full and the index now consumes three data pages P_0, P_1, and P_3 as shown in Figure 5.

Finally, we insert the fourth trajectory L_4. Since L_4 crosses two previously full quadrants (south-west and north-west), two bucket splits take place and the indexed space becomes partitioned as shown in Figure 6.

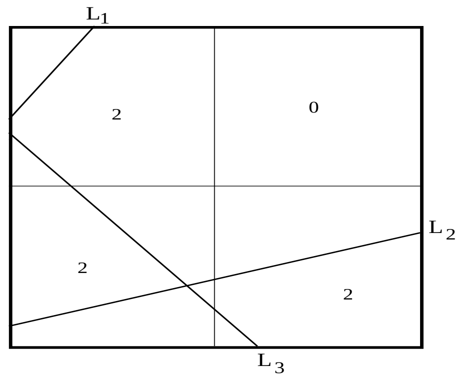

Figure 4: The indexed space partitioned into four quadrants upon insertion of the third trajectory L_3.

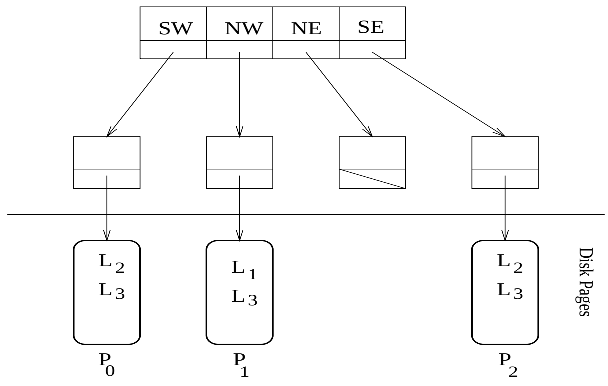

Figure 5: The quadtree grows by one level after bucket split. The root points to four leaf nodes.

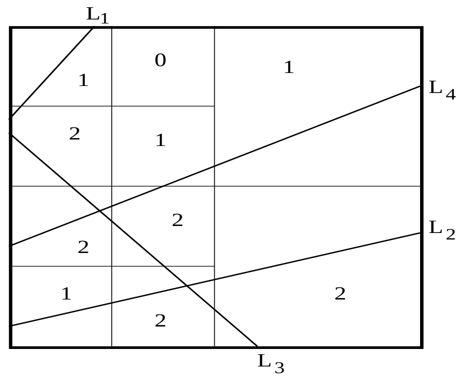

Figure 6: Partitioning of the indexed space and number of index points per quadrant after insertion of L_4.

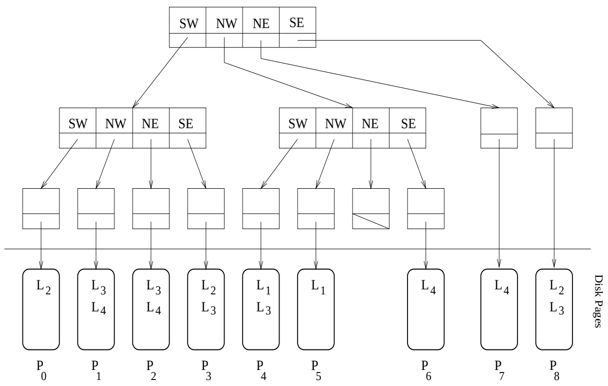

Figure 7: Quadtree index after insertion of L_4.

Numbers inside each quadrant indicate the number of trajectory segments crossing it. L_1 now crosses two quadrants, L_2 crosses three, L_3 crosses five, and L_4 which caused the splits crosses four quadrants. This results in 14 index points distributed over nine data pages. The corresponding quadtree is shown in Figure 7. It is now of height two and contains three internal nodes and 10 leaf nodes one of which is not pointing to any data page (corresponds to an empty quadrant).

An important observation is that a bucket split leads to duplication of index elements. The same trajectory which was represented by a single point before the split becomes represented by one, two, or three points after the split. In the example above, L_3 was represented by a single point in the south-west quadrant before insertion of L_4. After insertion and split, it is represented by three points and is present in the three data pages P_1, P_2, and P_3 corresponding to the crossed subquadrants (see Figure 7). A bucket split is also the mechanism by which the indexed space gets partitioned. Partitioning of indexed space is in turn reflected in the height of the quadtree-based indexing directory. More important however is the impact of splits on the total number of index elements corresponding to a given system size N. These two quantities are used in the definition of a new parameter introduced to characterize duplication in the context of our application.

Definition 2 *: The* **Duplication Ratio** *D of a quadtree index is the average number of copies that a single object has in the index:*

$$D = \frac{\text{Number of Index Points}}{\text{Number of Objects}}$$

Note that duplication of object information is an intrinsic characteristic of our application. In theory, index information about an object's dynamic attribute consists of an infinite number of points in its plotted graph. We are here basically approximating it by a finite number of points. Hoping for a single index element per object is akin to using a single point to approximate the whole graph describing the way the attribute changes over time.

4 Experimental Setup

To study the performance of our indexing technique, we have implemented the bucket PR quadtree as described above together with its associated insertion, deletion, and query processing operations. The programs were written in C and executed on Sun Sparc workstations running the Solaris operating system.

The main parameters in our simulation model are the system size N, the index reconstruction period ΔT, and the speed ratio α. Given N, we generate randomly the corresponding N linear equations describing the way attributes change over time. This is done by generating the intercepts b randomly according to the uniform distribution over the attribute space $[A_{min} \ldots A_{max}]$. We then generate the values of the slopes a in a range determined by the speed ratio α we want to test. Note that this is different from the way line

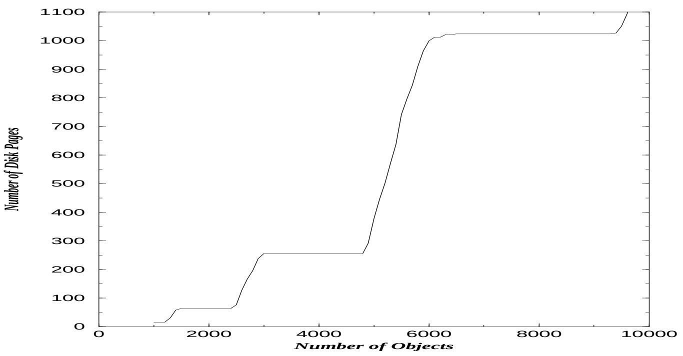

Figure 8: Storage requirements for small system sizes.

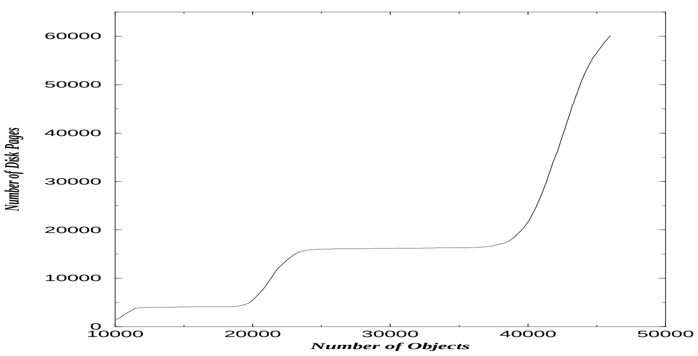

Figure 9: Storage requirements for large system sizes.

segments were generated in the study of PMR quadtree. In [15], it is stated that line segments were obtained by first generating points that are uniformly distributed over a square region and then connecting them. This is not applicable here since we are dealing with trajectories. Deletion requests are generated randomly from the space of active object IDs. Another important component of our experiments is the buffer manager. We manage the buffer using the *Least Recently Used* (LRU) page replacement policy. Buffer size BF is also another model parameter. We experiment with BF values of 8, 16, 32, 64, 128, and 256.

In query processing, the relative size of query ranges is important. Let R denote the absolute range length of a range query. The ratio $R/\Delta A$ gives the relative span of the given query which is traditionally used in percentage form ($\frac{R}{\Delta A} \times 100$). We will simply call this quantity *range size*. In our study, we experiment with range sizes which span 10%, 1%, 0.1%, and 0.01% of the total attribute space ΔA. The centers of the ranges are generated randomly and uniformly over the attribute space $[A_{min} \ldots A_{max}]$ and the ranges' lower and upper boundaries are then determined by the desired range size. The results of some of the conducted experiments are presented in their appropriate sections.

5 Storage Requirements

The storage requirements of the quadtree index depend mainly on system size N and duplication ratio D. Note that their product $(N \times D)$ is the total number of resulting index points. Let R denote the index record size (in bytes) and M the storage consumption of the index. At a perfect bucket utilization of 100%, we have $M = N \times D \times R$. Letting μ denote average bucket utilization ($0 \leq \mu \leq 1$), we obtain the more general equation

$$M = \left(\frac{R}{\mu}\right) N \times D \tag{2}$$

Equation 2 also hints at the importance of keeping the number of duplicates low and utilization high to the extent possible.

Figures 8 and 9 show disk consumption as a function of system size N. We use disk page sizes of 4 Kbytes and our records are 12 bytes long so that bucket size B in our application is about 340. Note that both graphs have the same pattern in which the number of disk pages remains fixed for a while then increases rapidly over a small interval of objects count N until it reaches some new plateau at which it remains fixed for an even longer interval of N. The plateau pattern prevails in other experiments as well; we provide a precise definition below.

Definition 3 *: A* **plateau** *is an interval* $[N_i \ldots N_{i+L})$ *in the number of objects over which the resulting quadtree requires exactly the same number of disk pages. We say* L *is the* **plateau length**.

Plateaus occur at values of 16, 64, 256, 1024, 4096, and 16384 disk pages. The number of disk pages in a plateau is thus four times that of the previous plateau so that in general we have plateaus at 4^i disk pages. The number 4^i comes from the fact that the quadtree starts with one (4^0) disk page and every split wave multiplies the number of disk pages by four.

The question is then why do we have plateaus or why does the number of disk pages stabilize for a while at values of 4^i? The reason is that all subquadrants tend to fill up at the same rate and reach capacity B at the same time. This leads to a wave of splits across all quadrants of the quadtree that is triggered by a relatively small number of newly inserted objects. Furthermore, insertion of a single object typically triggers splits in a big number of quadrants that correspond to a single attribute interval. After these splits are over, it will take many insertions before the new disk pages are filled again. During these insertions, disk pages are getting nearer to full capacity but no new disk pages are being required by the quadtree. For this reason, we have a plateau shape followed by a sharp rise that only ends at the next plateau.

The value 4^i corresponds also to the way the underlying indexed space is partitioned at plateaus. It is equal to the product $2^i \times 2^i$ and comes from the fact that at plateaus, the initial indexed space becomes partitioned into a $2^i \times 2^i$ grid. We embody this observation into the following definition.

Definition 4 *: An* i^{th}**-regular quadtree** *is one which partitions the underlying indexed space into a* $2^i \times 2^i$ *grid of quadrants. We also say that it is at the* i^{th} **plateau**.

A split wave then takes us from an i^{th}-regular quadtree to an $i+1^{th}$-regular one. In the process, the number of points per trajectory (or copies per object) increases, thus increasing the amount of duplication.

The duplication ratio is given in Figure 10 as a function of N. Again the plateau pattern prevails but this time plateaus occur at powers of two ($2^i : i > 0$) rather than four so that we also have plateaus at $D = 8$ and $D = 32$ (not included in the figure). The figure shows a plateau at $D = 64$, then at $D = 128$ (not a power of 4), then at $D = 256$ for $N \approx 50000$. We observe that D corresponds to the number of quadrants on each side of the indexed space. At plateaus, D is a perfect power of two and is exactly equal to the number

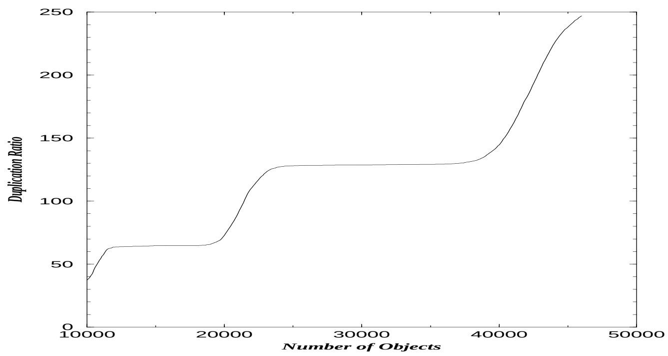

Figure 10: Duplication ratio.

of quadrants on every side of the original quadrant. In summary the duplication ratio increases with N. This might limit the scalability of the method to large system sizes. However, since duplication is caused by bucket splits we could slow its increase by rethinking the bucket splitting event.

In the context of our specific application, a bucket split is associated with the following three events:

1. Each object in the original bucket generates two copies on the average. This assumption is based on a relevant theoretical result[3] [15]. The result is applicable for trajectories uniformly distributed in space and orientation. In our application, intercepts are restricted to belong to a finite interval $[A_{min} \dots A_{max}]$ so that for low values of i, the number of trajectories generating one copy is less than those generating three. However, the effect of this imbalance is diluted for higher values of quadtree order i (above 5) where the border quadrants responsible for the imbalance constitute a diminishing fraction of the total number of leaf quadrants in the quadtree. As such, most quadrants 'see' the trajectories crossing them as uniformly distributed in orientation thereby averaging near to two copies during splits.

2. One page is replaced by four leading to a quadrupling of local space[4].

3. Local utilization drops from 100% to 50% (follows from observations 1 and 2).

It will then be beneficial if we could delay splits to the extent possible. We do this using the following technique. When a bucket of capacity B records overflows, we allocate a second *twin bucket* that will store the next B objects whose trajectory crosses the corresponding region of space. When the two twin buckets overflow, they are replaced by the usual four buckets which correspond to the four subquadrants of the indexed region. This is similar in spirit to Lomet's *partial expansion* technique [13]. We delay quadrupling of local space by increasing the number of buckets per node. Partial expansion delays doubling of local space

[3] "... on the average, if lines are drawn from a sample distributed uniformly in space and orientation, the average number of quadrants intersected by a line passing through a quartered block is two" [15, page 268].

[4] This is hardwired into the quadtree structure and thus independent of the context.

by using the so-called *elastic buckets*; the number of buckets per node remains fixed but the bucket size increases by a factor less than two (e.g., 1.25 or 1.5) before eventually doubling.

Using this simple amendment, we achieve a significant reduction of storage requirements which reaches 50% in theory. In practice, we expect that the average gain will not be much lower than that since 50% corresponds to the gain achieved locally (at the level of a single bucket split/expansion). This gain comes from the resulting decrease in duplication ratio achieved by the technique. By delaying splits we could 'enjoy' lower values of D at higher system sizes. Next, we look into bucket utilization.

6 Storage Utilization

In this section, we are interested in computing the space efficiency of our quadtree in the context of our particular application. In the analysis of utilization provided here, we do not provide the details of all the derivations or proofs; the complete exposition may be found in [20]. We start by defining more precisely the quantity for which we are seeking a formula.

Definition 5 *: The* **Utilization Ratio** *U is the fraction of memory used by the index elements in a given quadtree relative to the total memory capacity of the disk pages consumed by that quadtree.*

For the purposes of our analysis, we need a more specific definition. Let $P(N)$ be equal to the number of index records in the quadtree which indexes N objects and $D(N)$ be the number of disk pages required to store a quadtree which indexes N objects. Let $U(N)$ denote the utilization ratio in a quadtree index corresponding to a system size of N. Then we have

$$U(N) = \frac{P(N)}{D(N)B} \tag{3}$$

We are interested in the *mean utilization ratio* $\overline{U}$ which would theoretically be given by the formula

$$\overline{U} = \lim_{N \to \infty} \frac{\sum_{p=1}^{N} U(p)}{N} \tag{4}$$

However to keep analysis tractable, we need to compute the mean over a finite range of values of N. We choose the range of values between two consecutive plateaus as our finite range. Let N_i denote the number of objects at an i^{th} full plateau of a quadtree. We define the i^{th} *mean utilization ratio* $\overline{U_i}$ as follows.

$$\overline{U_i} = \frac{\sum_{p=N_i+1}^{N_{i+1}} U(p)}{N_{i+1} - N_i} \tag{5}$$

Let us express formula 5 in words. Starting from the point when the i^{th} plateau breaks, we insert objects consecutively until we reach the full $i+1^{th}$ plateau and compute the utilization after every insertion (this is the term $U(p)$ in the summation). We then calculate $\overline{U_i}$ as the arithmetic mean of those intermediate values. Next, we introduce some terminology.

We say that an i^{th} plateau is *full* if all the $2^i \times 2^i = 2^{2i}$ disk pages are full (contain B index elements). Furthermore, when the first quadrant in a full i^{th} plateau splits we say the i^{th} plateau *breaks* and that the i^{th} split wave *begins*. The i^{th} split wave *ends* when the last quadrant of the i^{th}-regular quadtree that has not yet split undergoes a split yielding the $i+1^{th}$-regular quadtree and initiating the $i+1^{th}$ plateau.

Our analysis is based on the following assumptions inspired by the real behavior of splits and trajectories in our application.

1. Upon bucket split, we have an average of two copies per object.

2. A trajectory falls in a single attribute interval $[a_j \ldots a_{j+1})$. As far as utilization is concerned, this assumption does not change the analysis. A trajectory typically spans the whole session inside the attribute space but crosses more than one attribute interval during the session. We are here basically collecting its fragments from the different intervals crossed and 'gluing' them on the original interval from which it started at the beginning of the session.

3. We assume a scenario of a shortest split wave, that is one which ends with the minimum number of insertions. This wave is as follows. We start with a full i^{th} plateau containing 2^{2i} quadrants and 2^i attribute intervals. Based on assumption 2, we need exactly 2^i insertions to begin and end the i^{th} split wave (each insertion falls in a distinct attribute interval). Each such insertion causes 2^i splits along the time axis which transforms a 1×2^i row into a $2 \times 2^{i+1}$ grid. This scenario is also independent of utilization.

In passing from an i^{th} full plateau to the $i+1^{th}$ one, we go through two distinct phases. First, we have a split wave during which all of the 2^{2i} buckets split; call this the $SPLIT$ phase. We call $FILL$ phase the second phase in which insertions increase bucket utilization without adding new buckets to the index. We then need to compute some intermediate values before solving for $\overline{U_i}$.

Let P_i denote the number of index elements at the i^{th} full plateau; then $P_i = 2^{2i}B$ where B is as usual, the bucket size. Let P_i^{sw} denote the number of index elements immediately after the i^{th} split wave ends. The value of P_i^{sw} is directly given in the following lemma.

Lemma 1 *: The number of index elements immediately after the i^{th} split wave ends is given by the formula*

$$P_i^{sw} = 2^{2i+1}(B+1) \tag{6}$$

Proof. A new object splits a row of length 2^i into two making it of length $2 \times 2^i = 2^{i+1}$. Then, each of the 2^i objects which are responsible for the shortest split wave scenario will end up with 2^{i+1} copies for a total of $2^i \times 2^{i+1} = 2^{2i+1}$. Furthermore, at the end of the split wave each of the old P_i index elements will have generated two copies for a total of $2 \times P_i = 2 \times 2^{2i}B = 2^{2i+1}B$. The sum of the old and new index elements is thus $2^{2i+1}(B+1)$.□

We then also need to find a closed form for the difference $N_{i+1} - N_i$ which we denote by ΔN_i. This is the denominator for the expression for $\overline{U_i}$ (Equation 5) and stands for object insertions needed to pass from the i^{th} full plateau to the $i+1^{th}$ full plateau. The expression for ΔN_i is also given directly in the next lemma.

Lemma 2 : *The number of insertions ΔN_i that take a quadtree index from a full i^{th} plateau to a full $i+1^{th}$ plateau is given by the expression*

$$\Delta N_i = 2^i + \frac{P_{i+1} - P_i^{sw}}{2^{i+1}} \tag{7}$$

Proof. In a shortest split wave scenario, 2^i new insertions are needed to split all 2^{2i} quadrants in a full i^{th} plateau. This brings us into a $2^{i+1} \times 2^{i+1}$ grid containing P_i^{sw} index points. To compute the remaining number of insertions necessary, we use the fact that each new object will have 2^{i+1} since we are in a $2^{i+1} \times 2^{i+1}$ grid. When we reach a full $i+1^{th}$ plateau, we have P_{i+1} index elements (by definition of P_i above). The remaining number of insertions can thus be expressed as the difference in number of index elements at the end of the split wave and the end of the FILL phase divided by the number of copies which is 2^{i+1}. This is given by the expression

$$\frac{P_{i+1} - P_i^{sw}}{2^{i+1}}$$

Adding the 2^i objects which induced the split wave gives the result. □

Substituting Equation 6 into the above expression and simplifying yields $\Delta N_i = 2^i B$. Before giving the result for mean utilization $\overline{U_i}$, we need to define two more intermediate variables. Let $U^i_{SPLIT}(m)$ denote the utilization ratio when m objects have been inserted in the $SPLIT$ phase; m satisfies the constraint $0 < m \leq 2^i$. By analogy, we define $U^i_{FILL}(m)$ as the utilization ratio when m objects have been inserted during the $FILL$ phase; that is m satisfies the constraint $2^i < 2^i + m \leq \Delta N_i$ (which simplifies to $0 < m \leq 2^i(B-1)$). These are exactly the component utilizations we talked about earlier that we want to sum and divide by $N_{i+1} - N_i$ to obtain $\overline{U_i}$ (see Equation 5). A more precise expression for $\overline{U_i}$ is then as follows.

$$\overline{U_i} = \frac{\sum_{0<m\leq 2^i} U^i_{SPLIT}(m) + \sum_{0<m\leq 2^i(B-1)} U^i_{FILL}(m)}{\Delta N_i} \tag{8}$$

It now suffices to find formulas for $U^i_{SPLIT}(m)$ and $U^i_{FILL}(m)$ and compute the corresponding summations. The result for both quantities is given below.

Lemma 3 : *The utilization ratio $U^i_{SPLIT}(m)$ after m objects are inserted in the SPLIT phase at the i^{th} plateau is given by the formula*

$$U^i_{SPLIT}(m) = 1 - \frac{2(\frac{B-1}{B})m}{3m + 2^i} \tag{9}$$

Proof. Utilization is defined as space used by the index points divided by the space capacity of the data pages consumed. We define $P(i,m)$ to be the number of index points induced after m splits ($m \geq 1$) starting from a full i^{th} plateau. We also define $S(i,m)$ to be the number of buckets induced after m splits starting from a full i^{th} plateau. $U^i_{SPLIT}(m)$ would then be given by the following formula

$$U^i_{SPLIT}(m) = \frac{P(i,m)}{S(i,m) \times B} \tag{10}$$

It then suffices to find expressions for $P(i,m)$ and $S(i,m)$. We start with $S(i,m)$.

Each of the m objects splits one row of length 2^i into a $2 \times 2^{i+1}$ subgrid. There are 2^i such rows to be split. Hence, the non-split part of the grid is composed of $2^i - m$ rows of 2^i quadrants each giving a total of $2^i(2^i - m) = 2^{2i} - 2^i m$. The split part of the grid contains $m \times 2 \times 2^{i+1} = 2^{i+2}m$ quadrants. The total number of pages is then given by the expression

$$2^{i+2}m + 2^{2i} - 2^i m$$

which when simplified yields the following result

$$S(i,m) = 3 \times 2^i m + 2^{2i} \tag{11}$$

We then compute the formula for $P(i,m)$. After m insertions, there are $2^i - m$ rows which have not split yet. These contain a total of $2^i \times (2^i - m)$ quadrants all of which are full since they belong to the i^{th} full plateau. Consequently they contain $2^i \times (2^i - m)B$ index points. The other m rows contained $2^i m$ full quadrants at the onset of the split waves for a total of $2^i mB$ index points. Since upon splitting each point generates two copies, the old elements account for $2^{i+1}mB$ index points in the split segment. The new objects causing the m splits generate 2^{i+1} copies each since they split a row that is 2^i quadrants long. Then, they account for $2^{i+1}m$ index points. Therefore, $P(i,m)$ is given by the expression

$$2^i(2^i - m)B + 2^{i+1}mB + 2^{i+1}m$$

which when simplified yields the following result

$$P(i,m) = 2^{2i}B + 2^i m(B+2) \tag{12}$$

Substituting Equations 11 and 12 into the expression for $U^i_{SPLIT}(m)$ in Equation 10 and simplifying yields the result.□

It could be observed from Equation 9 that utilization is 1 at $m = 0$ before splitting starts. At $m = 2^i$ when the split wave ends, utilization is just above 50% and equals $\frac{B+1}{B}$; in our application it yields a utilization value of 50.147%. This is the lower bound on utilization irrespective of quadtree order i. We next present the formula for $U^i_{FILL}(m)$.

Lemma 4 *: The utilization ratio $U^i_{FILL}(m)$ after m objects are inserted in the FILL phase at the i^{th} plateau is given by the formula*

$$U^i_{FILL}(m) = \frac{B+1}{2B} + \frac{m}{2^{i+1}B} \tag{13}$$

Proof. We start at the end of the minimal split wave with a $2^{i+1} \times 2^{i+1}$ grid containing $P_i^{sw} = 2^{2i+1}(B+1)$ index elements. Every new element generates 2^{i+1} copies when mapped over this grid hence utilization is given by the following expression

$$U^i_{FILL}(m) = \frac{P_i^{sw} + 2^{i+1}m}{2^{2(i+1)}B}$$

Simplifying this expression gives the result.□

We can then present the formula for mean utilization $\overline{U_i}$ at the i^{th} plateau.

Theorem 1 *: The i^{th} mean utilization ratio $\overline{U_i}$ is given by the following formula in which B denotes bucket size and $\epsilon_i = \log_2(2^i + 3) - i$.*

$$\overline{U_i} = \left(\frac{B-1}{B^2}\right)\left[\frac{3B+1}{4} + \frac{1}{2^{i+2}} - \frac{2}{3}\left(1 - \frac{\ln 2(2-\epsilon_i)}{3} - \frac{1}{2^i}\right)\right] + \frac{1}{B} \tag{14}$$

The details of the proof of Theorem 1 are omitted here and may be found in [21]. A few remarks are in place about Equation 14. First, the effect of i is almost negligible. We can see this by looking at Equation 14 and observing that i appears in the fractions $\frac{1}{2^i}$, $\frac{1}{2^{i+2}}$, and in ϵ_i all of which yield negligible values for $i > 5$ (e.g, $\epsilon_6 = 0.066$). Calculations of actual $\overline{U_i}$ confirm this. We have $\overline{U_4} = 0.750472$, $\overline{U_6} = 0.750428$, and $\overline{U_{10}} = 0.750417$. This agrees with our intuition; namely, that the particular choice of plateau will not affect the computation of mean utilization since we are averaging over a large enough number of utilization values (this is $\Delta N_i = 2^i B$). We then remove i from $\overline{U_i}$ and simply use $\overline{U}$. Second, we also remark that the value of $\overline{U_i}$ is dominated by the first multiplication factor; namely the product of $\frac{B-1}{B^2}$ and $\frac{3B+1}{4}$. In fact, we may safely use the following approximation.

$$\overline{U} \approx \frac{(B-1)(3B+1)}{4B^2} + \frac{1}{B} \tag{15}$$

Our value of $B = 340$ yields a utilization of 0.751468, only slightly higher than the $\overline{U_i}$ values given above. Third, we note that utilization is fairly independent of the bucket size B. For a value of $B = 100$, we have a utilization of 0.754975 using the approximation in Equation 15. At $B = 50$, $\overline{U} = 0.7599$ and at $B = 10$, we get $\overline{U} = 0.7975$. Note that we are trying these values to gain better insight into the effect of B. In practice, we do not expect page sizes below 1 Kbyte which yields a value of $B = 85$ in our particular application. It then seems that $\overline{U}$ increases with decreasing B. In the asymptotic case of $B = 1$ (a data page holds only a single index record), we have $\overline{U} = 1$ which again agrees with our intuition (since all buckets will always

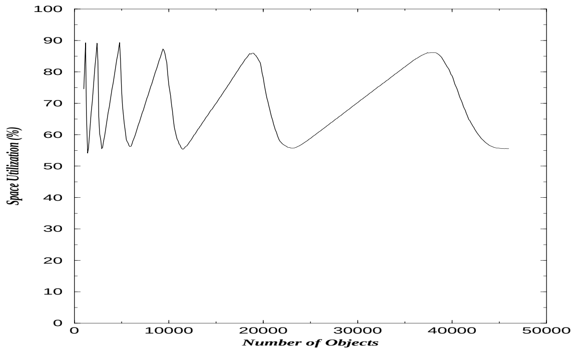

Figure 11: Percentage utilization.

be fully utilized). Since most values of $\overline{U}$ are around 0.75, we may look into Equation 15 to see if this is a general tendency. In fact, writing $\frac{(B-1)(3B+1)}{4B^2}$ as the product

$$\left(\frac{B-1}{B}\right)\left(\frac{3B+1}{4B}\right)$$

and using the approximations $\frac{B-1}{B} \approx 1$, $\frac{3B+1}{4B} \approx \frac{3}{4}$, and $\frac{1}{B} = 0$, we obtain the elegant general value for $\overline{U}$ that approximates it neatly over various values of i and B:

$$\overline{U} \approx \frac{3}{4} \tag{16}$$

The sample values we calculated above for different B and i cases increase our confidence in the validity of this approximation. It also agrees with our expectation since $\frac{3}{4}$ is the middle value between the theoretical maximum utilization of 1 and the minimum utilization which is slightly above $\frac{1}{2}$. The experimental evaluation of the percentage utilization as a function of the number of objects is given in Figure 11. It confirms the fact that minimum utilization does not drop below 50%. Maximum utilization however is always bounded by the 90% barrier. This is because in practice, the $i+1^{th}$ split wave begins before the i^{th} plateau becomes full. In other terms, when the first quadrant split in a $2^i \times 2^i$ partitioned space takes place, there are still quadrants which are not yet full (i.e., contain less than B index elements).

7 Index Reconstruction

Among the distinctive features of our approach to dynamic attribute indexing is the fact that we reconstruct the index periodically. This periodic reconstruction of the index is associated with non-negligible cost which would naturally increase with system size. In this section, we present the details of an index reconstruction algorithm that minimizes CPU time; each object is processed and inserted exactly once and in constant time. Furthermore, each leaf data page involved in the result incurs only one disk page transfer; I/O overhead is then also kept minimal.

The key idea of the algorithm is to find a way to predict the final plateau of the quadtree index to be constructed given system size N. In other words, knowing that we will end up with a partitioning of indexed space into a $2^i \times 2^i$ grid, we want to find i beforehand. We then precompute the contents of the $2^i \times 2^i$ buckets in memory and transfer them to the disk. Let us call i in an i^{th}-regular quadtree the *order* of that quadtree. In Section 7.1 we explain the derivation of quadtree order. Section 7.2 presents the reconstruction algorithm and Section 7.3 suggests ways to cope with non-uniform data distributions.

7.1 Finding Quadtree Order

Suppose we start with a single empty data bucket and we have the N insertion operations to execute. Then the condition $N > B$ tells us that we will have to split the bucket anyway. We then obtain $2N$ index points and 4 buckets of capacity $4B$. Notice that if $2N \leq 4B$ then the 2×2 quadtree is enough to store the N objects. The condition $2N > 4B$ similarly tells us that we will have to go through the second split wave anyway after which we have $4N$ index points (2 copies per object as assumed before) and 16 buckets of capacity $16B$. The next condition to evaluate is then $4N > 16B$. Generalizing this we find that the top-down insertion of N objects will reach the i^{th}-split wave if the condition $2^i N > 2^{2i} B$ is satisfied. By the same token, the $i+1^{th}$ split wave will not be reached if $2^{i+1} N \leq 2^{2(i+1)} B$. Combining these two constraints, we obtain the following characterization of when N objects produce an i^{th}-order quadtree.

$$2^i B < N \leq 2^{i+1} B$$

The following formula for i is then readily computed.

$$i = \left\lceil \log_2 \left(\frac{N}{B} \right) \right\rceil - 1 \tag{17}$$

Remember that we assumed the i^{th} split wave breaks the i^{th} plateau only when it is full. This is not the case in practice where one or more of the $2^i \times 2^i$ buckets of the i^{th}-regular quadtree might become full and split before all 2^{2i} buckets become full. To remedy this we could simply add 1 to i hence constructing a larger grid than the theoretical prediction to cater for premature splitting (which is always the case in real applications). This would be a waste of memory capacity if N were only slightly greater than $2^i B$ where we expect it is too early for splits of the next wave to begin. On the other hand it is reasonable to do so if N were too near to the value $2^{i+1} B$. In the former case we propose to use a $2^i \times 2^i$ grid and add overflow buckets for the few quadrants which are found to have overflowed.

7.2 The Algorithm

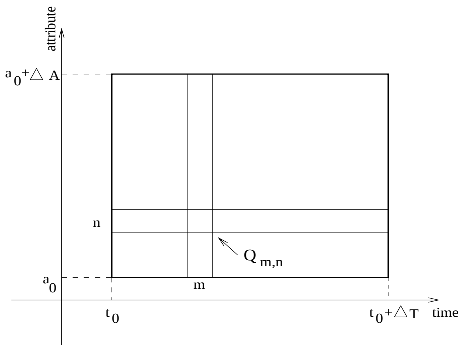

Figure 12: Definition of $Q_{m,n}$.

Before presenting the algorithm we introduce some necessary notation. In what follows we talk about an i^{th}-regular quadtree. We have earlier defined ΔA to be the length of the attribute dimension. Let $\delta t_i = \frac{\Delta T}{2^i}$ denote the length of each quadrant along the time axis and $\delta a_i = \frac{\Delta A}{2^i}$ denote the length of each quadrant along the attribute axis. Let our indexed space be $\mathcal{S} = [t_0 \ldots t_0 + \Delta T], [a_0 \ldots a_0 + \Delta A]$ and let

$$Q_{m,n} = [t_0 + m\delta t_i \ldots t_0 + (m+1)\delta t_i), [a_0 + n\delta a_i \ldots a_0 + (n+1)\delta a_i). \quad (0 \leq m, n < 2^i)$$

designate the subquadrant of our space which lies at the intersection of the m^{th} time interval and n^{th} attribute interval as shown in Figure 12. Finally let $s = 2^i$ be the side length of the i^{th}-regular quadtree measured in number of intervals.

Since there will be 2^{2i} buckets in the final quadtree, we will need to fill and write 2^{2i} disk pages during reconstruction and this is then the minimum disk access cost which we can hope for. It would then be better if we could shift all other auxiliary overhead into the CPU which is what we propose to do. The idea is to construct an in-memory $s \times s$ array (call it Q) which corresponds to the quadrants $Q_{m,n}$ of our indexed space defined above. We then compute for each of the N trajectories the coordinates (m and n: $0 \leq m, n < 2^i$) of quadrants it crosses and add the object information to every such quadrant. The entry $Q[m,n]$ of our array is thus a set defined as follows.

$$Q[m,n] = \{o_p : \text{trajectory of } o_p \text{ crosses } Q_{m,n}\}$$

We call the latter operation *object path computation.* Let us then provide a short description of the object path computation algorithm (hereafter denoted by PCA) and see its complexity.

The algorithm is given in Figure 13. The objects to be inserted are considered one by one in the outer loop (ranging over variable p). Given an object o_p, we examine the s time slices $[t_0 + m\delta t_i \ldots t_0 + (m+1)\delta t_i)$ $(0 \leq m < s)$ one by one in increasing order of m. Using the equation of motion $f_p(t) = a_p t + b_p$, the while loop computes at each time interval the attribute slices in which object o_p's trajectory falls. This is accomplished in the above algorithm using the function $BoundaryIntervals()$ which takes as arguments the

for $p \leftarrow 1$ **to** N **do**

$m \leftarrow 1$

$ObjectFinished \leftarrow FALSE$

while $m \leq s$ **and** $\overline{ObjectFinished}$ **do**

$< n_{p,low}, n_{p,high} > \leftarrow BoundaryIntervals(m, a_p, b_p)$

if $n_{p,low} < n_{p,high}$ **then**

for $n \leftarrow n_{p,low}$ **to** $n_{p,high}$ **do**

$Q[m,n] \leftarrow Q[m,n] \cup \{o_p\}$

else

$ObjectFinished \leftarrow TRUE$

endif

endwhile

endfor

Figure 13: Path Computation Algorithm.

current time slice and the trajectory parameters. It returns the first attribute interval $n_{p,low}$ and the last attribute interval $n_{p,high}$ crossed by the trajectory. In case the interval $[n_{p,low} \ldots n_{p,high}]$ is empty, the while loop is exit and we move to the next object otherwise index points are inserted in their relevant buckets (i.e., $Q[m,n]$).

The PCA algorithm is analogous to Bresenham's algorithm as used in computer graphics and computational geometry although the context here is quite different. Since it constructs the whole index in main memory, the PCA algorithm typically requires a large table in main memory to store the Q array. This translates into a prohibitive cost for deeper indices. To circumvent this problem, we could modify the algorithm to construct the index incrementally in two or more phases thus consuming less of the main memory at any given phase. In general, we could design a P-phase algorithm that consumes only $\frac{1}{P}$ of the total space needed by the grid. In the i^{th} phase, we compute index data for the i^{th} segment of the grid according to some suitable grid partitioning policy. An interesting question would then be whether we could devise the algorithm so that each of the P phases requires sublinear time (i.e., less than $\mathcal{O}(N)$).

The insertion of an object o_p in $Q[m,n]$ expressed in the above algorithm as a set union could be implemented to be $\mathcal{O}(1)$ if we use an array of size B for $Q[m,n]$. The CPU cost of regenerating our quadtree is optimal in the sense that no multiple insertions or recomputations are done for a single object. In summary, no extra overhead is incurred to create and place index points in their correct quadrants other than the strict minimum. Once the quadrants array Q is filled, we just transfer its contents from memory to the disk by allocating one disk page for every $Q[m,n]$ ($0 < m,n \leq 2^i$) and copying the index points in $Q[m,n]$ to it. Since there are $2^i \times 2^i = 2^{2i}$ entries in array Q, this amounts to 2^{2i} disk accesses in the worst case. In practice however we may have the opportunity to produce a *packed index*[5] which ideally requires a single disk seek while the rest of I/O time is spent transferring the buckets to contiguous pages on the disk. Index packing also improves query performance dramatically.

7.3 Handling Skewed Data

The above algorithm assumes uniform distribution of trajectories' intercept values over the attribute dimension. As such it generates a regular quadtree in which all leaf quadrants are of the same dimension ($2^{-i} \times 2^{-i}$). In the case of skewed data (i.e., intercepts b) distributions, there will be big quadrants in sparse areas of the attribute space and smaller quadrants in the densely populated areas. The above algorithm is then not applicable (as is) to skewed data since quadtree order is not defined in this case. We propose two solutions to handle skewed data both of which make use of the path computation algorithm presented above; implementation details are omitted.

In the first solution, we start by running the PCA algorithm using the maximum depth d of the quadtree of the previous session as a value for quadtree order i. Notice that for sparsely populated segments of the attribute space, we will have many quadruples of sibling quadrants occupied by less than B unique trajectories (the rest are duplicates). We call those quadruples *mergeable quadrants* since they ought to be merged and replaced by their parent quadrant. The idea is then to make a few passes over the $2^d \times 2^d$ array constructed by the PCA algorithm and in each pass merge or consolidate any mergeable quadrants. We need $d-1$ passes in the worst case. The disadvantage of this method is that if the difference between the maximum and minimum depths is big, the $2^d \times 2^d$ sized array construction will be an overkill and a big waste of main memory and CPU requirements. The other extreme is to start with a single page (which corresponds to $1 \times 1 = 2^0 \times 2^0$ sized array) and insert the objects one by one incurring all ensuing splits which would result in a prohibitive cost. The idea is then to compute some *average depth* which we denote by $\overline{D}$ and use it in place of the maximum depth d. Given a quadtree partitioning of the attribute space, the average depth $\overline{D}$ may be computed by weighting each possible depth with the fraction of indexed space that is indexed at that depth and summing across all the depths. Assume (without loss of generality) that our indexed space is the unit square (i.e., a 1×1 quadrant). Let n_i denote the number of leaf quadrants that

[5] "... we define an index to be packed if each of its buckets uses a minimal amount of space to store entries (without room for growth), and all its buckets are allocated contiguously on disk" [17].

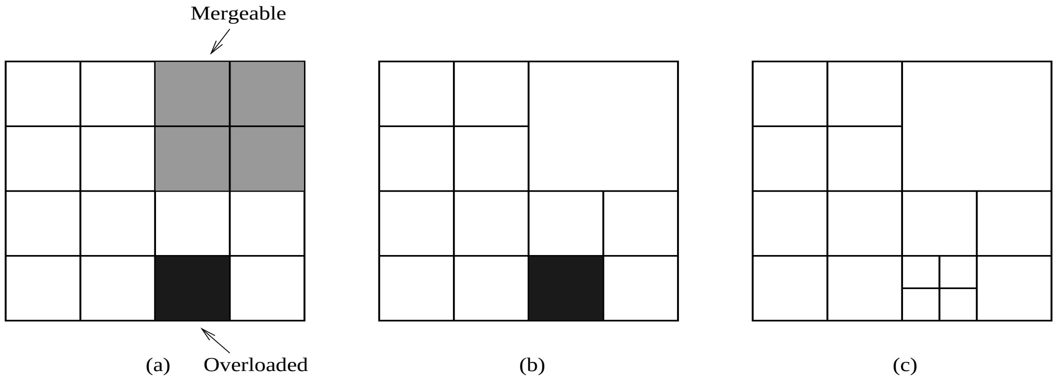

Figure 14: (a) Initial approximation of the space partitioning using average depth of 2 ($2 \leq \overline{D} < 3$). (b) Mergeable quadrants identified in the north-east and consolidated. (c) Overloaded quadrant identified and split yielding the correct configuration.

are at depth i. Then $\overline{D}$ would be given by the following formula

$$\overline{D} = \sum_{i=1}^{d} n_i \times 2^{-i} \times 2^{-i} \times i \tag{18}$$

We run the PCA algorithm over a $2^{\lfloor \overline{D} \rfloor} \times 2^{\lfloor \overline{D} \rfloor}$[6] array. Where the areas are dense, we will have more than B unique objects in a single array entry. Quadrants corresponding to such array entries are thus *overloaded* and need further splitting. We then need a second phase for splitting overloaded quadrants beside the phase for merging mergeable quadrants. In practice, we will have to temporarily allocate one or more extra pages for overloaded subquadrants until the splitting phase begins so that no loss of information occurs by using $\lfloor \overline{D} \rfloor$ as a temporary approximation of grid shape. The $2^{\lfloor \overline{D} \rfloor} \times 2^{\lfloor \overline{D} \rfloor}$ partitioning of the indexed space may be seen as the nearest regular approximation of the irregular partition resulting from skewed data. Alternatively, the average depth $\overline{D}$ may be viewed as answering the following question: At which depth d does a partitioning of the indexed space into a $2^d \times 2^d$ grid yield an average bucket occupancy that is nearest to B? As such, we expect that $\overline{D}$ will minimize the overhead of the splitting and merging phases.

As an example, suppose our approximation using average depth yielded $\lfloor \overline{D} \rfloor = 2$ so that we start with a 4×4 grid as shown in Figure 14-(a). Note that this is the situation after running the PCA algorithm. The figure shows that the north-east subquadrant of the parent quadrant are unnecessarily partitioned into four sparse quadrants. These are identified as mergeable quadrants. Figure 14-(b) shows the situation after merging them. Beside this, an overloaded quadrant is detected and split in the split phase into four subquadrants. The final correct partition of the indexed space is shown in Figure 14-(c). The merge phase thus coarsens the space partitioning while the split phase refines it.

[6] We could also take the nearest integer to $\overline{D}$

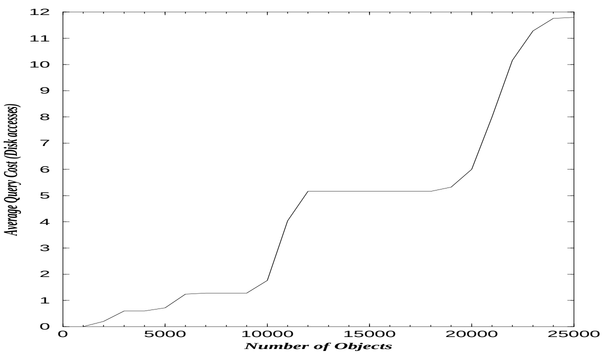

Figure 15: Instantaneous query cost at a high range size of 10%.

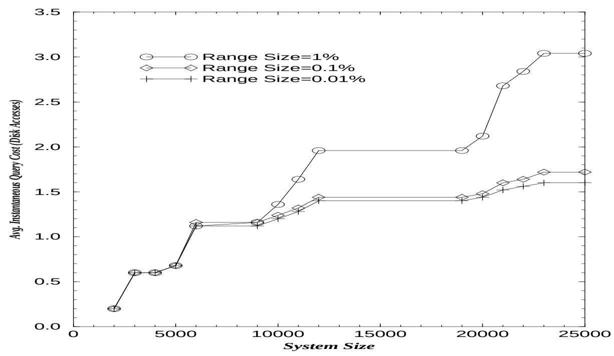

Figure 16: Instantaneous query cost at low range sizes.

The second solution relies on the idea of using the shape of the quadtree generated in the previous session to predict or approximate its shape in the new session. By shape we mean the particular way the indexed space ended up being partitioned in the previous session. We take this information from the leaf nodes of the quadtree directory. It is reasonable to expect that if a segment of the attribute space was sparsely populated at the end of the last session, then it will continue to be so for some (or all of the) time in the next session. In fact, we may even have segments that are consistently dense across the lifetime of the application[7]. We then simply run the PCA algorithm above over the specific partition induced by the previous session which results in big savings in memory cost compared to the first solution. Note that during the span of a session, we expect the space partitioning inducing the quadtree to change through splits and merges governed by insertions and deletions. For this reason, we do not get into a dead cycle in which we repeat always the same quadtree.

The idea of this second solution is similar in spirit to the *seeded trees* of Lo and Ravishankar [12] in which they copy the first k levels of an existing (seeding) R-tree and use them as a seed for a new R-tree to be constructed from a different data set than that which induced the seeding tree. Their work addresses the problem of designing efficient algorithms for processing the spatial join query [2] in the special case where a spatial index is not available for at least one of the participating relations. The difference is that they take their 'seeds' from the top of the tree while (in a sense) we take them from the bottom.

8 Query Processing

Our quadtree based index exhibits good performance for the two popular types of queries described earlier: instantaneous and continuous queries. In-memory overhead consists of recursively descending the quadtree

[7] In the context of vehicle navigation systems, dense segments may correspond to the center of a city where motion is slower and the number of vehicles is higher while long highways between cities correspond to the sparse segments of the indexed space.

directory to reach the leaves pointing to data pages relevant to the queried range. For a quadtree of order i, there will be 4^i leaf nodes and the above operation will thus be $\mathcal{O}(i)$. The rest of this section discusses the more dominant I/O cost for both query types. We assume we are at an i^{th}-regular quadtree.

An instantaneous query submitted at time t_{now} with an attribute range $[R_{low} \ldots R_{high}]$ targets the time range $[t_{now} - \delta t \ldots t_{now} + \delta t]$ (or $[t_{now} \ldots t_{now} + \delta t]$) where δt is a small time lapse to be chosen according to the application domain. Then we may constrain δt to be small compared to a single time interval of the quadtree (i.e $\delta t \ll \frac{\Delta T}{2^i}$). In fact, we want it to be small enough to fit in a single time slice δt_i. Alternatively we may adopt the policy of evaluating an instantaneous query submitted at t_{now} using the time interval in which t_{now} falls since our purpose in using parameter δt was to have a finite approximation to the infinitesimal t_{now}. For continuous queries, the theoretical time range over which they are evaluated is $[t_{now} \ldots \infty)$. In practice, if a continuous query comes in a period $[p\Delta T \ldots (p+1)\Delta T)$, it is first evaluated over the time interval $[t_{now} \ldots (p+1)\Delta T)$ then over all subsequent periods of the application until it is explicitly deleted from the queries list.

The number of data pages required to answer a range query is (intuitively) equal to the number of quadrants covered by the range. Let us characterize this more accurately. Let $A_Q = R_{high} - R_{low}$ denote the length of the attribute range of a query Q and $T_Q = T_{high} - T_{low}$ the length of the time range of query Q. Excluding the effect of buffering, the disk access cost C_Q^{disk} of such a query is

$$C_Q^{disk} = \left(1 + \left\lceil \frac{A_Q}{\delta a_i} \right\rceil\right) \left(1 + \left\lceil \frac{T_Q}{\delta t_i} \right\rceil\right) \tag{19}$$

We have simply multiplied the number of intervals covered by each of the two ranges. For the special case when one range starts exactly at the beginning of an interval, the 1 in the multiplicand in Equation 19 is omitted. As ranges are supplied independently of the current status of the quadtree and its partition, we expect the general case embodied in Equation 19 to hold most of the time. We can then determine the cost of instantaneous and continuous queries using this formula.

Letting C_{inst}^{disk} and C_{cont}^{disk} denote the disk cost of instantaneous and continuous queries (respectively) over an i^{th}-regular quadtree we obtain the following formulas.

$$C_{inst}^{disk} = 1 + \left\lceil \frac{A_Q}{\delta a_i} \right\rceil \tag{20}$$

$$C_{cont}^{disk} = 2^i \left(1 + \left\lceil \frac{A_Q}{\delta a_i} \right\rceil\right) \tag{21}$$

The cost of an instantaneous query is just the number of attribute intervals its attribute range spans while for continuous queries it is that number multiplied by 2^i, the number of time intervals in ΔT.

Figures 15 and 16 show the average cost of instantaneous queries as a function of N across a few typical attribute range percentages (10%, 1%, 0.1%, and 0.01%). On the average we need less than three disk

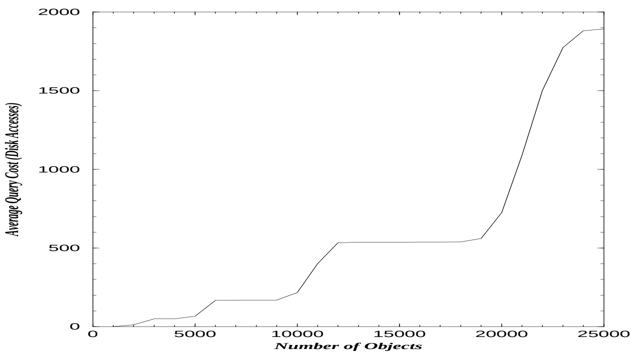

Figure 17: Continuous query cost at a high range size of 10%.

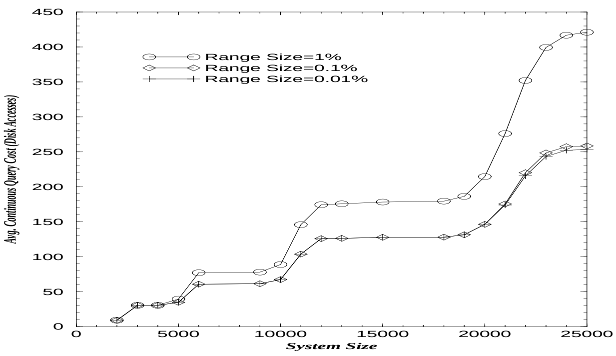

Figure 18: Continuous query cost at low range sizes.

accesses per instantaneous query for system sizes up to 30000. Given that a typical I/O operation costs 10 milliseconds [4, 3], this allows us to process a reasonably large number of instantaneous queries per second. The reason why range sizes of 0.1% and 0.01% have identical costs (up to $N = 20000$) is that such ranges are thin enough to fit in a single attribute interval length δa_i so that in both cases one or two data pages will need to be retrieved. This averages to less than two disk accesses as shown in Figure 16. For the larger range sizes such as 10% of the attribute space, access cost rises to five disk accesses for values of N below 20000 and to ten disk accesses on the next plateau. This still translates into a reasonable I/O cost. Note that we do not consider the improvement technique mentioned earlier where we delay a bucket split by allocating a twin bucket. As this technique delays the rise to the next plateau, we expect to enjoy a small number of disk accesses per instantaneous query especially for small to average range sizes (below 5%).

The average I/O cost of continuous queries is given in Figure 17 for a query range size that is 10% of the attribute space and in Figure 18 for the smaller range sizes. Remember that for continuous queries, for each attribute interval (of length δa_i at the i^{th} plateau) covered by the range, we have to examine all the corresponding 2^i quadrants which span the length of a single session (ΔT). Figure 18 confirms this by the fact that for the low range sizes of 0.1% and 0.01% (which span no more than a single attribute interval), the plateaus occur at exact values of 2^i (e.g., 32 for $N < 5000$, 64 for $6000 \leq N \leq 9000$, and 128 for $6000 \leq N \leq 9000$). In general, the average cost per continuous query ranges between 100 and 200 disk accesses. For large range sizes, Figure 17 shows that the cost may quickly become prohibitive. However, we expect that in practice it is not sensible to request continuous information about large segments of the attribute space. For system sizes below 10000 (which we consider enough for many application domains), we have an I/O cost in the order of a few tens for small range sizes that is still subject to improvement using the technique of delayed bucket splits described above. Furthermore, if the index is packed then we would incur a single seek time per query. If we cannot achieve total packing of the index, we may still achieve a lower level of packing in which all quadrants of a single attribute interval (2^i for uniform data) are allocated to contiguous pages on the disk. This again reduces the cost of a single continuous query to a few disk seeks. Although continuous queries remain more costly compared to instantaneous queries, they still lend themselves to optimizations in a way which instantaneous queries do not.

Since in any arbitrary session, we (typically) have a fixed minimum number of continuous queries to answer, after a few sessions we can identify *hot spots* of the attribute space which not only continue to be referenced from one session to the next but are also referenced by many queries inside a single session. We then could achieve enormous gains in I/O overhead by storing in memory ready answers for the most heavily referenced intervals. Identification of the hottest attribute intervals could be done using a simple statistical analysis of past query ranges. We would then process those intervals at the beginning of each session by filtering duplicates (from the 2^i quadrants corresponding to a single interval), packing the unique objects that cross every hot interval in the coming session, and storing them into a table in memory. For every continuous query, we first compare its range with the packaged intervals to see if there are intersections. Only the fragments of its range which are not 'hot' will then require to be brought into memory and incur I/O cost. Given the relatively high cost of continuous queries, we expect tangible improvements even for a modest number of hot intervals.

9 Related Work

To the best of our knowledge, no prior work addressed the specific problem of dynamic attribute indexing. Dynamic attributes themselves were defined in [19]. This is not surprising given that the database community has only recently begun to explore the impact of mobility on data management issues (see for example [1], [5], and [10]).

However, our problem is closest to the problem of indexing collections of line segments since we start with lines in two-dimensional space. The way the line indexing problem is approached in the literature is influenced by the types of queries expected. Hoel and Samet [7] identify three classes of queries: (a) those which deal only with the line segments, (b) those which involve the line segments and the space from which they are drawn, and (c) those which involve attributes of the line segments. Jagadish [11] proposes a solution for a family of queries belonging to the first class above. His solution relies on transforming lines into points in a transform space in which slope is plotted in one axis and the intercept is plotted on the other (the so-called *Hough Transform*). In [7], the PMR quadtree is used as an access method for line segment databases and an algorithm for finding the nearest line segment to a given point is presented (a query belonging to the second class above). In [8], the use of the R^*-tree and the R^+-tree for indexing line segments is also studied in conjunction with the PMR quadtree.

In our particular line indexing problem we did not need to care about vertices[8] and line information is summed up in the (slope,intercept) pair associated with every object. Furthermore, more care is given to index reconstruction in our work since it is a periodic overhead.

The problem of dynamic attribute indexing is that of handling data that is (rapidly) evolving over time. As such the work of Shivakumar and Garcia-Molina [17] on indexing evolving databases is of particular importance. The authors present a set of interesting techniques for maintaining indexing information about a (moving) window of days. The resulting indices are called *wave indices* and the techniques apply to almost all classes of index structures. The question addressed is how to efficiently add data of the new day and remove data of the oldest day(s) from the wave index. It is an interesting question whether or not we could adapt their algorithms for our context to replace the periodic index reconstruction approach.

Another work worth mentioning here is that of Shekhar and Yang [16] in which they address the issue of mobility in a geographic system. Their index is called *MoBiLe file* and maps the two-dimensional space of motion to the disk tracks and sectors while attempting to preserve proximity relationships. This map requires a mapping function and knowledge about the population distribution. An object's geographical location is then used as the primary key to locate the disk block where it resides. However, since they do not make use of an equation of motion, the nature of their work is different from ours.

10 Conclusion

In this paper, we have proposed a solution to the problem of indexing dynamic attributes that is based on the quadtree structure. A key idea used in our index is the prediction of the future values of a dynamic attribute from an approximative linear function which describes the way it changes over time. Starting from the plotted graphs of these functions in the two-dimensional time attribute space, we transform the problem into the spatial indexing domain and adapt the bucket PR quadtree to solve it. The aim was to support two types of range queries called instantaneous and continuous queries.

We provide detailed experimental and analytic studies of the main performance parameters of the indexing method. Since the approach requires periodic reconstruction of the index, we contribute an efficient algorithm for index reconstruction that is optimal in CPU and I/O costs. The index is also shown to exhibit very good performance for instantaneous queries that averages two disk accesses per query. It also exhibits good performance for continuous queries where we suggested a minor optimization technique that yields tangible

[8] In fact, most of the line segments start and end at the same abscissa; namely t_i and $t_i + \Delta T$, the beginning and end of each session.

improvement of average cost per query.

We are currently working on another solution still based on the same general approach. Our focus and interest will be on reducing the average number of copies per object in the index (i.e., duplication ratio). We will also study the effect of ignoring the time dimension in the indexing directory.

Another trail of future work would be the design of an indexing method that does not require periodic destruction and reconstruction of the index. In other words, we would want an index that adapts in a more graceful way to the passage of time. One way to do this would be to adapt the techniques presented in [17] for wave indices to our application. Since we need an efficient way to generate indexing data corresponding to a given time interval, the task is not trivial. We will also look into the possibility of transforming the problem of dynamic attribute indexing into the temporal indexing domain.

References

[1] R. Alonso and H. F. Korth. Database System Issues in Nomadic Computing. In *ACM SIGMOD International Conference on the Management of Data*, pages 388–392, Washington, DC, May 1993.

[2] T. Brinkhoff, H. Keigel, R. Shneider, and B. Seeger. Multi-Step Processing of Spatial Joins. In *Proc. of ACM SIGMOD International Conference on the Management of Data*, pages 197–208, Minneapolis, Minnesota, May 1994.

[3] T. Brinkhoff, H. Kriegel, and B. Seeger. Parallel Processing of Spatial Joins Using R-Trees. In *12th International Conference on Data Engineering*, pages 258–265, New Orleans, Louisiana, February 1996.

[4] T. Brinkhoff, H. P. Kriegel, and B. Seeger. Efficient Processing of spatial Joins Using R-Trees. In *ACM SIGMOD International Conference on the Management of Data*, pages 237–246, Washington, DC, May 1993.

[5] M. Dunham and A. Helal. Mobile Computing and Databases: Anything New? *SIGMOD Record*, 24(4):5–9, December 1995.

[6] M. J. Egenhofer. What's Special About Spatial? Database Requirements for Vehicle Navigation in Geographic Space. In *ACM SIGMOD International Conference on the Management of Data*, pages 398–402, Washington, DC, May 1993.

[7] E. G. Hoel and H. Samet. Efficient Processing of Spatial Queries in Line Segment Databases. In *Advances in Spatial Databases - 2nd Symposium, SSD'91*, Zurich, Switzerland, August 1991.

[8] E. G. Hoel and H. Samet. A Qualitative Comparison Study of Data Structures for Large Line Segment Databases. In *ACM SIGMOD International Conference on the Management of Data*, pages 205–214, California, USA, June 1992.

[9] T. Imielinski and B. R. Badrinath. Replication and Mobility. In *Second IEEE Workshop on Management of Replicated Data*, 1992.

[10] T. Imielinski and B. R. Badrinath. Mobile Wireless Computing: Challenges in Data Management. *Communications of the ACM*, 37(10):18–28, October 1994.

[11] H. V. Jagadish. On Indexing Line Segments. In *Proc. of the* 16^{th} *Very Large Databases Conference*, pages 614–625, Brisbane, Australia, 1990.

[12] M. Lo and C. V. Ravishankar. Spatial Joins Using Seeded Trees. In *ACM SIGMOD International Conference on the Management of Data*, pages 209–220, Minneapolis, Minnesota, May 1994.

[13] D. B. Lomet. Partial Expansions for File Organizations with an Index. *ACM Transactions on Database Systems*, 12(1):65–84, March 1987.

[14] H. Samet. The Quadtree and Related Hierarchical Data Structures. *ACM Computing Surveys*, 16(2):187–260, June 1984.

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

[16] S. Shekhar and T. A. Yang. Motion in a Geographical System. In *2nd Symposium on Advances in Spatial Databases*, pages 339–357, Zurich, Switzerland, August 1991.

[17] N. Shivakumar and H. Garcia-Molina. Wave-Indices: Indexing Evolving Databases. In *Proc. of ACM SIGMOD International Conference on the Management of Data*, 1997.

[18] A. P. Sistla and O. Wolfson. Temporal Triggers in Active Databases. *IEEE Transactions on Knowledge and Data Engineering*, 7(3):471–486, June 1995.

[19] A. P. Sistla, O. Wolfson, S. Chamberlain, and S. Dao. Modeling and Querying Moving Objects. In 13^{th} *International Conference on Data Engineering*, pages 422–432, Birmingham, UK, April 1997.

[20] J. Tayeb. Design and Performance Evaluation of Indexing Methods for Dynamic Attributes in Mobile Database Management Systems. Master's thesis, Bilkent University, May 1997.

[21] J. Tayeb, Ö. Ulusoy, and O. Wolfson. A Quadtree Based Dynamic Attribute Indexing Method. Technical Report BU-CEIS-9720, Bilkent University, 1997.

[22] O. Wolfson, S. Chamberlain, S. Dao, L. Jiang, and G. Mendez. Cost and Imprecision in modeling the Position of Moving Vehicles. In 14^{th} *International Conference on Data Engineering.*, Orlando, FL, February 1998.