Efficient Indexing of Spatiotemporal Objects

Marios Hadjieleftheriou*, George Kollios†, Dimitrios Gunopulos*, Vassilis J. Tsotras*
* Computer Science Department
University of California, Riverside
Email: marioh, dg, tsotras@cs.ucr.edu
† Computer Science Department
Boston University
Email: gkollios@cs.bu.edu

***Abstract*— Spatiotemporal objects i.e., objects which change their position and/or extent over time, appear in many applications. This paper addresses the problem of indexing large volumes of such data. We consider general object movements and extent changes. We further concentrate on "snapshot" as well as small "interval" historical queries on the gathered data. The obvious approach that approximates spatiotemporal objects with MBRs and uses a traditional multidimensional access method to index them is inefficient. Objects that "live" for long time intervals have large MBRs which introduce a lot of empty space. Clustering long intervals has been dealt in temporal databases by the use of partially persistent indices. What differentiates this problem from traditional temporal indexing is that objects are allowed to move/change during their lifetime. Better methods are thus needed to approximate general spatiotemporal objects. One obvious solution is to introduce artificial splits: the lifetime of a long-lived object is split into smaller consecutive pieces. This decreases the empty space but increases the number of indexed MBRs. We first introduce two algorithms for splitting a given spatiotemporal object. Then, given an upper bound on the total number of possible splits, we present three algorithms that decide how the splits should be distributed among the objects so that the total empty space is minimized.**

I. INTRODUCTION

There are many applications that create spatiotemporal data. Examples include transportation (cars moving in the highway system), satellite and earth change data (evolution of forest boundaries), planetary movements, etc. The common characteristic is that spatiotemporal objects move and/or change their extent over time.

Recent works that address indexing problems in a spatiotemporal environment include [28], [11], [10], [23], [29], [1], [20], [21], [12], [25]. Two variations of the problem are examined: approaches that optimize queries about the future positions of spatiotemporal objects ([11], [23], [1], [21], [22]) and those that optimize historical queries ([28], [29], [10], [17], [20], [21], [12], [25]), i.e., queries about past states of the spatiotemporal evolution. Here we concentrate on historical queries, so for brevity the term "historical" is omitted. Furthermore, we assume the "off-line" version of the problem, that is, all data from the spatiotemporal evolution has already been gathered and the purpose is to index it efficiently.

For simplicity we assume that objects move/change on a 2-dimensional space that evolves over time; the extension to a 3-dimensional space is straightforward. An example of such a spatiotemporal evolution appears in figure 1. The x and y axes represent the 2-dimensional space while the t axis corresponds to the time dimension. For the rest of this discussion *time is assumed to be discrete*, described by a succession of increasing integers. At time t_1 objects o_1 (which is a point) and o_2 (which is a 2D region) are inserted. At time t_2, object o_3 is inserted while o_1 moves to a new position and o_2 shrinks. Object o_1 moves again at time t_5; o_2 continues to shrink and disappears at time t_5. Based on its behavior in the spatiotemporal evolution, each object is assigned a record with a "lifetime" interval $[t_i, t_j)$ created by the time instants when the object was inserted and deleted (if ever). For example, the lifetime of o_2 is $[t_1, t_5)$. During its lifetime, an object is termed $alive$.

An important decision for the index design is the class of queries that the index optimizes. In this paper we are interested in optimizing topological snapshot queries of the form: "find all objects that appear in area S during time t". That is, the user is typically interested on what happened at a given time instant (or even for small time periods around it). An example snapshot query is illustrated in figure 1: "find all objects inside area S at time t_3"; only object o_1 satisfies this query.

One approach for indexing spatiotemporal objects is to consider time as another (spatial) dimension and use a 3-dimensional spatial access method (like an R-Tree [8] or its variants [3]). Each object is represented as a 3-dimensional rectangle whose "height" corresponds to the object's lifetime interval, while the rectangle "base" corresponds to the largest 2-dimensional minimum bounding region (MBR) that the object obtained during its lifetime. While simple to implement, this approach does not take advantage of the specific properties of the time dimension. First, it introduces a lot of empty space. Second, objects that remain unchanged for many time instants will have long lifetimes and thus, they will be stored as long rectangles. A long-lived rectangle determines the length of the time range associated with the index node (page) in which it resides. This creates node overlapping and leads to decreased query performance ([13], [14], [24], [25], [28], [12]). Better interval clustering can be achieved by using "packed" R-Trees (like the Hilbert R-Tree [9] or the STR-Tree [15]); another idea is to perform interval fragmentation using the Segment R-Tree [13]. However, the query performance is not greatly improved [12].

This work was partially supported by NSF grants IIS-9907477, EIA-9983445, and the Department of Defense.

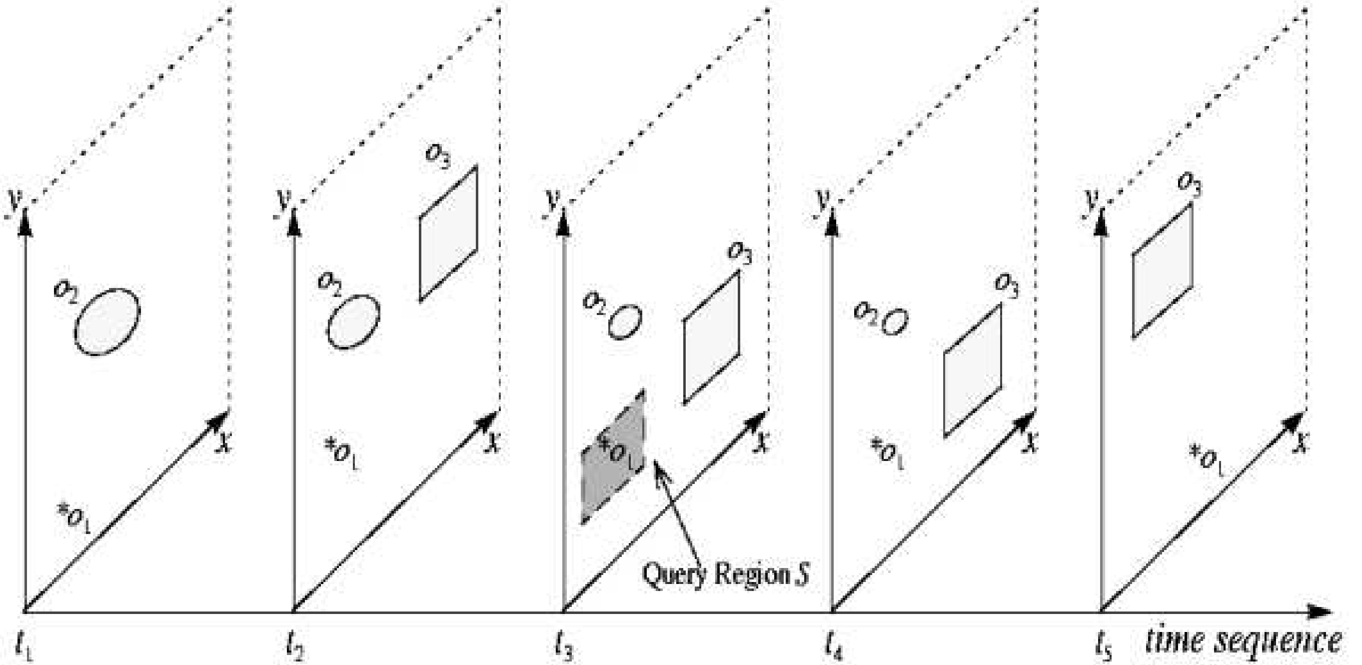

Fig. 1. An example of spatiotemporal object evolution.

Another approach is to exploit the monotonicity of the temporal dimension, and transform a 2-dimensional spatial access method to become partially persistent ([29], [17], [10], [12], [25]). A partially persistent structure "logically" stores all its past states and allows updates only to its most current state ([6], [16], [2], [30], [14], [24]). A historical query about time t is directed to the state the structure had at time t. Hence, answering such a query is proportional to the number of alive objects the structure contains at time t. That is, it behaves as if an "ephemeral" structure was present for time t, indexing the alive objects at t. Two ways have been proposed to achieve partial persistence: the overlapping [4] and multi-version approaches [6]. In the overlapping approach ([17], [29]), a 2-dimensional index is conceptually maintained for each time instant. Since consecutive trees do not differ much, common (overlapping) branches are shared between the trees. While easy to implement, overlapping creates a logarithmic overhead on the index storage requirements [24]. Conceptually, the multi-version approach ([16], [2], [30], [14], [25]) also maintains a 2-dimensional index per time instant, but the overall storage used is linear to the number of changes in the evolution. In the rest we use a partially persistent R-Tree (PPR-Tree [14], [25]). A short description of a PPR-Tree appears in section II-B.

Our approach for improving query performance is to reduce the empty space introduced by approximating spatiotemporal objects by their MBRs. This can be accomplished by introducing artificial object updates. Such an update issued at time t, artificially "deletes" an alive object at t and reinserts it at the same time. The net effect is that the original object is represented by two records, one with lifetime that ends at t and one with lifetime that starts at t. Consider for example a spatiotemporal object created by the linear movement shown in figure 2. Here, the 2-dimensional rectangle moved linearly, starting at t_1 from the lower left part of the (x, y) plane and reaching the upper right part at t_2. The original MBR is shown, as well. However, if this object is split (say, at the middle of its lifetime) the empty space is reduced since two smaller MBRs are now used (see figure 3 where the (x, t) plane is represented).

Clearly, an artificial split reduces empty space and thus, we would expect that query performance improves. However, it is not clear if the 3D R-Tree query performance will improve by these splits. An intuitive explanation is based on Pagel's query cost formula [19]. This formula states that the query performance of any bounding box based index structure depends on the total (spatial) volume, the total surface and the total number of data nodes. Using the artificial splits, we try to decrease the total volume of the data nodes (by decreasing the size of the objects themselves). On the other hand, the total number of indexed objects increases. In contrast, for the PPR-Tree the number of alive records (i.e., the number of indexed records) at any time instant remains the same while the empty space and the total volume (i.e., on average the total surface of all "ephemeral" 2-dimensional R-Trees) is reduced. Therefore, it is expected that the PPR-Tree performance for snapshot and small interval queries will be improved.

In [12] we addressed the problem of indexing spatiotemporal objects that move or change extent using *linear* functions of time. Assuming we are given a number of possible splits that are proportional to the number of spatiotemporal objects (i.e., the overall storage used remains linear) a greedy algorithm was presented that minimizes the overall empty space. In particular, the algorithm decides (i) which objects to split and (ii) how the splits are distributed among objects. The algorithm's optimality is based on a special *monotonicity*

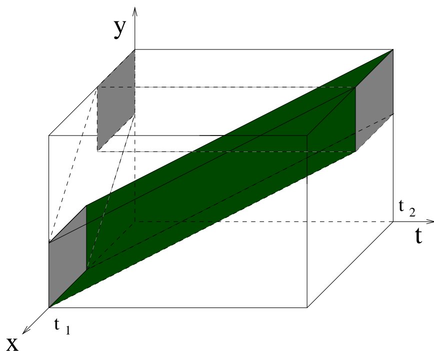

Fig. 2. A spatiotemporal object in linear motion.

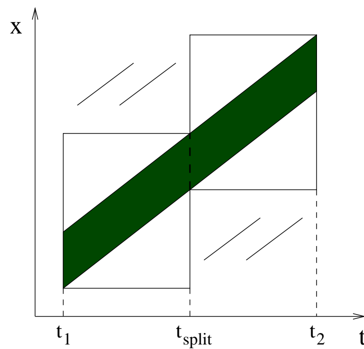

Fig. 3. An example of splitting an object once.

property which holds for linearly moving/changing objects:

Claim 1: Given a spatiotemporal object that follows a linear trajectory and a number of splits, the gain in empty space decreases as the number of splits applied to the object increases. Equivalently, the first few splits will yield big gain in empty space, while the more we split the less gain is obtained.

In this paper we address the more difficult problem where objects are allowed to move/change with general motions over time. Unfortunately, in this case the monotonicity property does not always hold. An example is shown in figure 4. One split will give much less gain in empty space than two.

Hence, new approaches are needed. We first present a dynamic programming algorithm and a heuristic for deciding how to apply a given number of splits on a general spatiotemporal object to maximize the gain in empty space. Furthermore, assuming that there is a predetermined total number of splits, we provide a dynamic programming algorithm for optimally distributing those splits among a collection of general spatiotemporal objects, while minimizing the total volume. The difficulty lies in the fact that the number of splits might not be enough to split every object in the collection, so an optimization criterion has to be considered. Finally, we describe two greedy algorithms that give close to optimal results, with very large gain in running time. While by using more splits the gain is increased, eventually the increase will not be substantial. A related problem is how to decide on the appropriate number of artificial splits. Assuming that a model is available for predicting the query cost of the index method used ([27], [26]), the number of splits can be easily decided.

To show the merits of our approach the collection of objects (including objects created by the artificial splits) are indexed using a PPR-Tree and a 3-dimensional R*-Tree [3]. Our experimental results show that the PPR-Tree consistently outperforms the R*-Tree for snapshot as well as small interval queries.

We note that some special cases of indexing general spatiotemporal objects have also been considered in the literature: (i) when the objects have no spatial extents (moving points) [20], [21], and (ii) when the motion of each object can be represented as a set of linear functions (piecewise linear trajectories [7], [12]). For the case that points move with linear functions of time, extensions to the R-Tree have been proposed (Parametric R-Tree [5] and the PSI approach in [21]). The problem examined here is however more complex as objects are allowed to move/change with a general motion over time.

The rest of the paper is organized as follows. Section II formalizes the notion of general movements/changes and provides background on the PPR-Tree. Section III presents the proposed algorithms. Section IV discusses how to use analytical models to find a good number of splits for a given dataset. Section V contains experimental results. Related work is given in section VI. Finally, section VII concludes the paper.

II. Preliminaries

A. Formal Notion of General Movements

Consider a set of N spatiotemporal objects that move independently on a plane. Suppose that the objects move/change with linear functions of time: $x = F_x(t)$, $y = F_y(t)$, $t \in [t_i, t_j)$. Then the representation of a spatiotemporal object O can be defined as a set of tuples: $O = \{([t_s, t_j), F_{x_1}(t), F_{y_1}(t)), \ldots, ([t_k, t_e), F_{x_n}(t), F_{y_n}(t))\}$ where t_s is the object creation time, t_e is the object deletion time, $t_j, \ldots, t_k$ are the intermediate time instants when the movement of the object changes characteristics and $F_{x_1}, \ldots, F_{x_n}$, $F_{y_1}, \ldots, F_{y_n}$ are the corresponding functions. In the general case, objects can move arbitrarily towards any direction. Representing an object's movement by the collection of locations for every time instant is not efficient in terms of space. It cannot be approximated very well with combinations of linear functions either, since the number of segments required cannot be bounded. A better approach is to use combinations of polynomial functions. An example of a point moving on the (x, t) plane with the corresponding functions describing its movement is shown in figure 5. For two dimensional movements every tuple would contain two functions, the first giving a movement on the x-axis and the second on the y-axis. This results to an object following a trajectory which is a combination of both functions. An alteration in the object's shape could be described in the same way. An example is shown in figure 6 where the object follows a general movement, keeps constant extent along the x-axis and changes extent along the y-axis.

By restricting the degree of the polynomials up to a maximal value, most common movements can be approximated or even represented exactly by using only a few tuples. As the number of tuples increases, more complicated movements may be represented and better approximations can be obtained. This

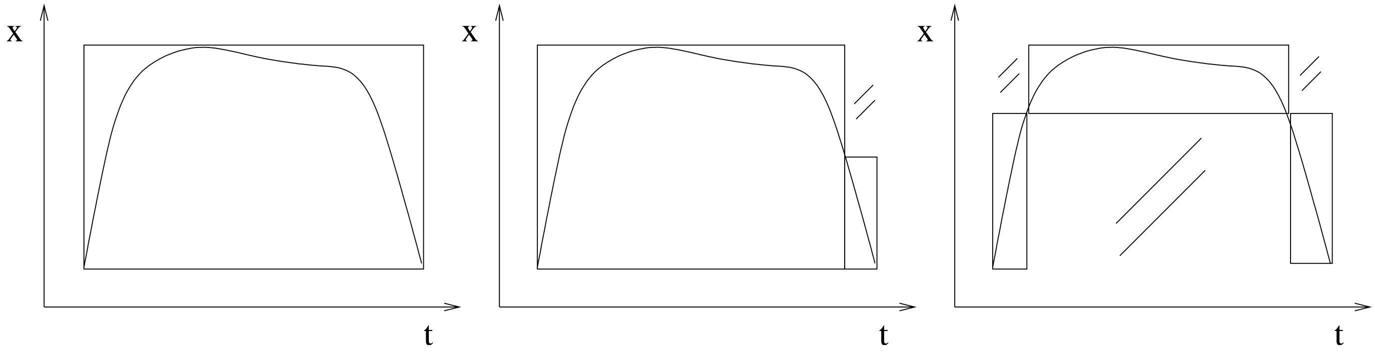

Fig. 4. An example where the monotonicity property does not hold.

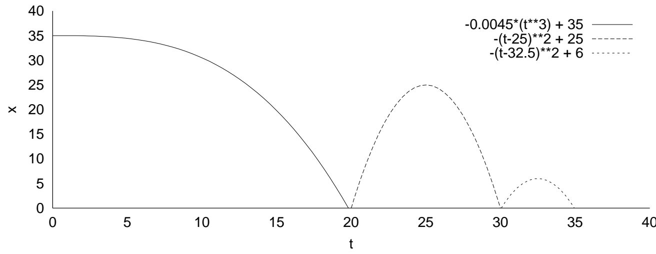

Fig. 5. A moving point and a corresponding set of polynomial functions representing the movement.

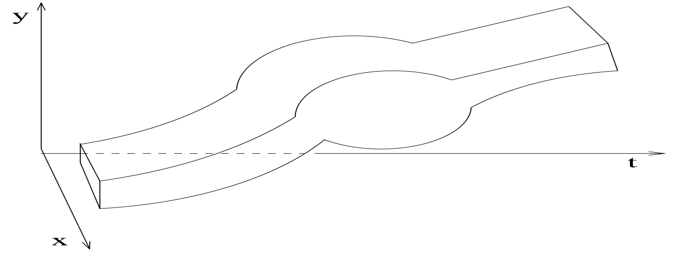

Fig. 6. A moving object that follows a general trajectory while changing shape.

approach is storage efficient, since few tuples are required for the representation of general movements/changes. It further guarantees that the MBR of any movement can be found easily by computing the minimum and maximum of the appropriate functions for all instants in a time interval. In this paper we focus in general movements/changes of objects and our algorithms are designed accordingly. We use, though, polynomial functions for generating moving objects for our experiments.

B. The Partially Persistent R-Tree

Consider a spatiotemporal evolution that starts at time t_1 and assume that a 2-dimensional R-Tree indexes the object shape and locations at time t_1. As the evolution advances, the 2-dimensional R-Tree also evolves, by applying the evolution updates (object additions/deletions) as they occur. Storing this R-Tree evolution corresponds to making the R-Tree partially persistent. The following discussion is based on [14]. While conceptually the partially persistent R-Tree (PPR-Tree) records the evolution of an ephemeral R-Tree, it does not physically store snapshots of all the states in the ephemeral R-Tree evolution. Instead, it records the evolution updates efficiently so that the storage remains linear to the number of changes, while still providing fast query time.

The PPR-Tree is actually a directed acyclic graph of nodes (a node corresponds to a disk page). Moreover, it has a number of root nodes, each of which is responsible for recording a consecutive part of the ephemeral R-Tree evolution. Data records in the leaf nodes of a PPR-Tree maintain the temporal evolution of the ephemeral R-Tree data objects. Each data record is thus extended to include the two lifetime fields: *insertion-time* and *deletion-time*. Similarly, index records in the directory nodes of a PPR-Tree maintain the evolution of the corresponding index records of the ephemeral R-Tree and are also augmented with *insertion-time* and *deletion-time* fields.

An index or data record is *alive* for all time instants during its lifetime interval. With the exception of root nodes, a leaf or a directory node is called *alive* for all time instants that it contains at least D alive records ($D < B$, where B is the maximum node capacity). When the number of alive records falls below D the node is split and its remaining alive records are copied to another node. This requirement enables clustering the objects that are alive at a given time instant in a small number of nodes (pages), which in turn will minimize the query I/O. Searching the PPR-Tree takes into account the lifetime intervals of the index and the data records visited. Consider answering a query about region S and time t. First, the root which is alive at t is found. This is equivalent to accessing the ephemeral R-Tree which indexes time t. Second, the objects intersecting S are found by searching this tree in a top-down fashion as in a regular R-Tree. The lifetime interval of every record traversed should contain time t, and its MBR should intersect region S.

III. Representation of Spatiotemporal Objects

Consider a spatiotemporal object O that moved from its initial position at time instant t_0 to a final position at time t_n with a general movement pattern. We can represent this object using its bounding box in space and time. However, this creates large empty space and overlap among the index nodes (we assume that an index like a 3-dimensional R-Tree or a PPR-Tree is used). A better approach is to represent the object using multiple boxes. That way a better approximation is obtained and the empty space is reduced. The object is split

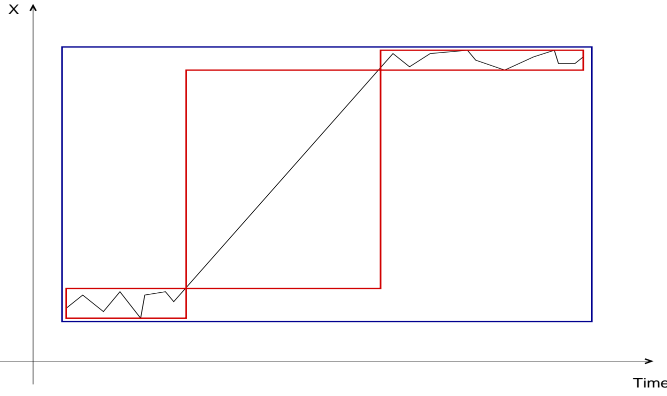

Fig. 7. An 1-dimensional example of representing an object with one and three MBRs.

into smaller consecutive objects and each one is approximated with a smaller bounding box (see figure 7). Note that we consider splitting along *the time axis only*, since the time dimension is the main reason of the increased empty space and overlap.

Next, we present methods for splitting objects in spatiotemporal datasets in order to decrease the overall empty space and increase the query performance. We break the problem into two sub-problems:

A Given an object and an upper limit in the number of splits we have to find how to split the object such that the maximum possible gain in empty space is obtained.

B Given a collection of objects and a predetermined number of splits we try to distribute the splits among all objects in order to optimize the query performance of the index.

A. Splitting One Object

Consider a spatiotemporal object O with lifetime $[t_0, t_n)$. Assume that we want to split the object into k consecutive objects in a way that minimizes the total volume of its representation. As we discussed above, we use splits in the form of artificial updates. It would be logical to assume that we only need to consider splitting an object at the points where the movement changes its characteristics (in case that the movement is represented by a combination of functions for example). This approach does not give optimal gain results in most cases, as could be shown with a counter example (see figure 7 for an intuition behind the problem).

1) An Optimal Algorithm (DPSplit): Let $V_l[0, i]$ be the volume of the MBRs corresponding to the part of the spatiotemporal object between t_0 and t_i after using l optimal splits. Then, the following holds:

$$V_l[0, i] = \min_{0 \le j < i} \{V_{l-1}[0, j] + V[j, i]\}$$

where $V[j, i]$ is the volume of the MBR that contains the part of the spatiotemporal object between t_j and t_i, without any splits. The formula states that in order to find the optimal solution for splitting the object between t_0 and t_i, we have to consider all intermediate time instants and combine the previous solutions. Using the above formula we obtain a dynamic programming algorithm that computes the optimal positions for the k splits and the total volume after these splits. This is achieved by computing the value $V_k[0, n]$.

Theorem 1: Splitting one object optimally using k splits can be done in $O(n^2 k)$, where the lifetime of the object is $[t_0, t_n)$.

Proof: We have to compute the nk values of the array $V_l[0, i]$, $0 \le l < k, 0 \le i < n$. Each value in the array can be found by computing the minimum of n values using the formula above. The volume $V[j, i]$ of the object between positions j and i can be precomputed for every run i, for all values of j, using $O(n)$ space and $O(n)$ time and thus does not affect the time complexity of the algorithm. ■

Input: A spatiotemporal object O as a sequence of n spatial objects, one at each time instant.
Output: A set of MBRs that cover O.

1) For $0 \le i < n$ compute the volume of the MBR for merging O_i with O_{i+1}. Store the results in a priority queue.
2) Repeat n times: Use the priority queue to merge the pair of consecutive MBRs that give the smallest increase in volume. Update the priority queue with the new merged MBR.

Fig. 8. The greedy heuristic (MergeSplit).

2) An Approximate Algorithm (MergeSplit): The dynamic programming algorithm is quadratic to the lifetime of the object. For objects that live for long time periods the above algorithm is not very efficient. A faster algorithm is based on a greedy approach. The idea is to start with n different boxes, one for each time instant and merge the boxes in a greedy way (figure 8). The running time of the algorithm is $O(n \lg n)$. To improve the running time we can merge all consecutive boxes that give a small increase in volume. Then, we can run the greedy algorithm starting with fewer boxes. This greedy algorithm gives in general sub-optimal solutions.

B. Splitting a Collection of Objects

In this subsection we discuss methods for distributing a number of splits K among a collection of N spatiotemporal objects. While using more splits to approximate the objects improves query performance by reducing the empty space, every split corresponds to a new record (the new MBR) and thus increases the storage requirements. Hence, if we are given a total number of splits K (which may correspond to an upper limit on the disk space) and a set of spatiotemporal objects, we want to decide which objects to split and how many splits should be allocated to each one of these objects.

1) An Optimal Algorithm: Assuming an ordering on the spatiotemporal objects (each object gets a number between 1 and N), let $TV_l[i]$ be the minimum total volume occupied by the first i objects with optimal l splits, and $V_j[i]$ be the total volume for approximating the ith object using j splits (e.g., with $j+1$ boxes.) we observe that:

$$TV_l[i] = \min_{0 \le j \le l} \{TV_{l-j}[i-1] + V_j[i]\}$$

Input: A set of spatiotemporal objects with cardinality N.
Output: A near optimal minimum volume required to approximate all objects with K splits.

1) Find volume change for every object using one split. Store in a max priority queue.
2) For K iterations: Remove the top element of the queue. Assign the split to the corresponding object. Calculate the volume change if one more split was used on the same object. Reinsert the object in the queue.

Fig. 9. Greedy Algorithm.

We use the above formula to find the total volume for each number of splits. A dynamic programming algorithm can be used with running time $O(NK^2)$. To compute the optimal solution first we need the optimal splits for each object, which can be found by using the dynamic programming algorithm presented in section III-A.1. Hence, the following theorem holds:

Theorem 2: Optimally distributing K splits among N objects can be done in $O(NK^2)$.

2) The Greedy Algorithm: The dynamic programming algorithm described above is quadratic to the number of splits. That makes the algorithm impractical for many real life applications. Therefore, it is intuitive to look for an approximate solution. The simplest form of such a solution would be to use a greedy strategy: Given the split distribution so far, find the object that if split one more time (or for the first time) it will yield the maximum possible global volume reduction, and assign the split to that object. Continue in the same way until there are no more available splits. The algorithm is shown in figure 9. Assuming the best splits are known in advance for all objects, the complexity of the main loop is $O(K \lg N)$ and the complexity of the algorithm is $O(K \lg N + N \lg N)$.

3) The Look-Ahead Greedy Algorithm (LAGreedy): The result of the previous algorithm will not be optimal in the general case. One reason is the following. Consider an object that if split once gives a very small improvement in empty space but if split twice most of its empty space is removed (see figure 4 for an example). Using the greedy algorithm it is probable that this object will not be given the chance to be allocated any splits, because the first split is poor and other objects with better initial splits, will be chosen before it. However, if we allow the algorithm to consider more than one splits for every object at each step, the possibility for this object to be chosen for splitting is much higher. This observation gives an intuition about how the greedy strategy could be improved to give a better result, closer to the optimal. At every step, instead of finding the object that yields the largest gain by performing one more split, we could look ahead and find objects that result in even larger gain if two, three or more splits are assigned all at once.

For example, the look-ahead-2 algorithm works as follows (figure 10). First, all splits are allocated one by one in a greedy fashion, as before. When the greedy assignment is complete, one new priority queue PQ_{la1} is created, which sorts the objects by the gain offered by the last split allocated to them (if an object is split k times, sort the object according to the volume gain yielded by the kth split). The top of the queue is the minimum gain. A second priority queue PQ_{la2} is also needed, which sorts the objects by the volume that would be gained if two more splits were allocated to each one (if an object is split k times, sort the object according to what the volume gain would be if it was split $k+2$ times). The top of the queue is the maximum gain. If the gain of the top element of PQ_{la2} is bigger than the sum of the gains of the two top elements of PQ_{la1} the splits are reassigned accordingly, the queues are updated and the same procedure continues until there is no more change in the distribution of splits. In essence, the algorithm tries to find two objects for which the combined gain from their last splits is less than the gain obtained if a different, third object, is split two times extra (obviously an object not conforming to the monotonicity property).

Input: A set of spatiotemporal objects with cardinality N.
Output: A near optimal minimum volume required to approximate all objects with K splits.

1) Allocate splits by calling the *Greedy Algorithm*. PQ_{la1} is a min priority queue that sorts objects according to the gain given by their last split. PQ_{la2} is a max priority queue that sorts objects according to the gain given if two extra splits are used per object.
2) Remove top two elements from PQ_{la1}, let O_1, O_2. Remove top element from PQ_{la2}, let O_3. Make sure that $O_1 \neq O_2 \neq O_3$. If the gain for O_3 is larger than the combined gain for O_1 and O_2, redistribute the splits and update the priority queues.
3) Repeat last step until there is no change in the distribution of splits.

Fig. 10. LAGreedy Algorithm.

The algorithm has the same worst case complexity as the greedy approach. However, experimental results show that it achieves much better results for the small time penalty it entails.

IV. Finding the Number of Splits

The real objective of a split distribution algorithm is not to minimize the total volume itself, but to reduce the cost of answering a query if a predefined query distribution model is given. The objective function that should be optimized must represent this cost. Therefore, we need to define a function that is evaluated after each split and gives the average number of I/Os for answering a query. This function will help us find the number of splits that gives the best query results.

The splitting algorithms discussed in the previous section take as input the total number of splits and generate a new

dataset with smaller total volume. However, it is important to choose a number of splits that gives a good trade-off between query time and space overhead. The choice of a good value for this parameter affects the performance of the index structure. In this section, we give an overview of two methods for automatically computing a good value for this parameter.

The first method is based on using analytical models to predict the performance of the index. For a given number of splits, compute a distribution of splits and estimate some statistics about the generated dataset after splitting. Use the statistics as an input to the analytical model of the index that will be used and get a prediction on the number of disk accesses required to answer a random query from a query distribution. Repeat for a different split distribution. Thus, instead of trying to minimize the total volume, try to minimize the average query cost which is the ultimate goal!

Another way to find the best number of splits among a set of possible values is by using sampling. For each number of splits, an index is created and a set of representatives queries are evaluated on each index. The number that gives the best query performance is then chosen. However, instead of using the full dataset, it is possible to use a small sample and create the indices over this sample. The number of splits should be normalized to the full dataset.

V. Experimental Results

To test our algorithms we created four random datasets (uniform) of various sizes with moving rectangles in 2-dimensional space, and another four datasets with trains moving on a railway system (skewed). All object trajectories were approximated with MBRs. First, each object is split with the optimal (DPSplit) algorithm and the merge heuristic (MergeSplit) and the results are stored. Then, the optimal (Optimal), greedy (Greedy) and look-ahead-2 greedy (LAGreedy) algorithms are used to distribute various numbers of splits (from 1% to 150% of the total number of objects) among the objects; again the splitting results are stored. In the rest of the section, $a\%$ splits means that we use $\frac{a}{100}N$ total number of splits on a dataset with N spatiotemporal objects. For comparison purposes we also generated datasets using the simpler approach of splitting the objects in a piecewise manner, i.e., at the points in time where the polynomial representing the movement changes characteristics, which is the same as representing the movements with piecewise linear functions as in [21]. This method resulted in a number of splits about 400% of the total number of objects. Finally, we used the 3-dimensional R*-Tree and the PPR-Tree to index the resulting data. We decided not to use any packing algorithms for the R*-Tree, since from our previous experience, packing does not help substantially with datasets of moving objects. Packing algorithms tend to cluster together objects that might be consecutive in order even though they may correspond to large and small intervals. This leads to more overlapping and empty space [12]. Details about all datasets are presented in Table I.

For the moving rectangles time extents from 0 to 999 time instants. The lifetime of each object is randomly selected between 1 and 100 time instants. The object movement is approximated with a random number of polynomials between 1 and 10. The polynomials have randomly generated coefficients but are either of first or second degree (of course any type of polynomials could easily be generated). All movements are normalized in the unit square $[0, 1]^2$. The extents of the rectangles are randomly selected between 1/1000 and 1/100 of the total space.

For the railway datasets we generated a map containing 22 cities and 51 railways. The map approximates the states of California and New York with most of the tracks connecting intra state cities with each other. Few cities belong to different states in-between and there is a number of tracks connecting all the states across country. The distances of the cities were approximated to match reality. The trains are allowed to make up to 10 stops and travel for as long as 36 hours with a speed that is randomly selected between 60 and 75 miles per hour. No train is allowed to go back to the city were it originated without stopping somewhere else in-between. After all the parameters of the route have been calculated, a series of linear functions is generated, describing the trajectories in time. The railway tracks are considered to be straight lines. For these datasets also, time extents from 0 to 999 time instants.

For both index structures page capacity was set to 50 entries and we used a 10 page LRU buffer. In addition, for the PPR-Tree we set the minimum alive records per node parameter to $P_{version} = 0.22$, the strong version overflow parameter to $P_{svo} = 0.8$ and the strong version underflow to $P_{svu} = 0.4$. Also, the objects were first sorted by insertion time. For the R*-Tree objects were inserted in random order, but the time dimension was scaled down to the unit range first [25]. For the PPR-Tree the time dimension extent does not matter. To test the resulting structures we randomly generated four snapshot and two range query sets with 1000 queries each. Details about these sets are summarized in Table II. For all experiments the buffer was reset before the execution of every query. All experiments where run on an Intel Pentium III 1GHz personal computer, with 1GB of main memory.

A. *Comparison of Single-Object Splitting Algorithms*

First, we compare the dynamic programming (DPSplit) and the greedy (MergeSplit) algorithms for splitting a single object. In order to test their efficiency, we calculated the best splits of all objects contained in the random datasets, using as many splits as necessary and computed the CPU time needed. In figure 11, time is represented in a logarithmic scale, since for the large datasets the DPSplit algorithm needed almost one day to finish splitting the objects. On the other hand, the MergeSplit algorithm was very fast, requiring from a few minutes to a few hours. In order to show that the MergeSplit algorithm produces good splits, we optimally distributed 50% splits on all random datasets, and calculated the total volume of the resulting MBRs. The results are shown in figure 12. Clearly, MergeSplit gives very similar results to DPSplit.

B. *Comparison of Split Distribution Algorithms*

Next, we evaluate the performance of the Greedy and LA-Greedy algorithms, in comparison with the optimal dynamic

TABLE I
RANDOM AND RAILWAY DATASETS.

Random	10k	30k	50k	80k
Total Objects	10000	30000	50000	80000
Objects Per Instant (Avg.)	545.873	642.25	2749.97	4390.54
Total Segments	37179	111774	186539	297413
Object Lifetime (Avg.)	50	50	50	50
Object Extent (%)	0.1%-1%	0.1%-1%	0.1%-1%	0.1%-1%
Railway	10k	30k	50k	80k
Total Objects	10000	30000	50000	80000
Objects Per Instant (Avg.)	190.605	570.7	948.026	1522.78
Total Segments	27678	82792	137011	220996
Object Lifetime (Avg.)	18	18	18	18

TABLE II
SNAPSHOT AND RANGE QUERY SETS.

Snapshot	Cardinality	Extents (%)	Duration
Tiny	1000	0.01-0.1	1
Small	1000	0.1-1	1
Mixed	1000	0.1-5	1
Large	1000	1-5	1
Range	Cardinality	Extents (%)	Duration
Small	1000	0.1-1	1 - 10
Medium	1000	0.1-1	10 - 50

programming approach. We distributed 50% splits on the random datasets using all three algorithms and calculated the CPU cost of each approach. The results are shown in figure 13. Time is represented again in a logarithmic scale, since the optimal algorithm requires up to a few hours to distribute the splits for the bigger datasets. On the other hand, the two greedy approaches are much faster with the LAGreedy algorithm performing about only 10% slower than the Greedy algorithm, both requiring from a few seconds to a few minutes. To test the efficiency of our algorithms we distributed 150% splits using the LAGreedy algorithm on the random datasets and indexed the resulting MBRs using the PPR-Tree. Finally, we queried the resulting structures with the mixed snapshot query set, recording the average number of disk accesses needed. The results are shown in figure 14. For all the datasets that we tried, the LAGreedy algorithm performed as well as the optimal algorithm, while the Greedy approach was always inferior.

C. Benefits and Drawbacks of Splitting

In order to show that splitting a dataset is beneficial only for the partial persistence indexing approach, we distributed a series of different numbers of splits on all datasets using the LAGreedy algorithm. Then, we indexed the resulting MBRs using a 3-dimensional R*-Tree and a PPR-Tree. We queried the resulting structures using the small range queries and recorded the average number of disk accesses needed. The results for the 50k random dataset are shown in figure 15. Observe that as the number of splits increases the average number of disk accesses needed decreases substantially for the PPR-Tree, while there is a negative effect for the R*-Tree. For completeness, in figure 16 we present the disk space required by the two structures, for an increasing number of splits. We can see that the PPR-Tree requires almost twice as much space as the R*-Tree, which is a reasonable tradeoff considering the gain in query performance. The LAGreedy combined with the PPR-Tree achieves an improvement of 30% in query performance over the best alternative (75 vs 110 I/Os).

D. Comparing Partially Persistent and Straightforward Approaches

Finally, we performed a number of snapshot and range queries in order to test how the partially persistent and the R*-Tree structures react when increasing the number and type of objects. For the small range queries the R*-Tree is somewhat better for unsegmented data and 1% up to 5% splits, while the PPR-Tree becomes much better when the number of splits increases. In figure 17 we plot the average number of disk accesses for small range queries and 150% splits for the PPR-Tree and 1% splits for the R*-Tree, distributed with the LAGreedy algorithm. We also plot the performance of the R*-Tree with the piecewise data. It is obvious that the partial persistence approach is by far superior after splitting. For all datasets and any number of splits we observed that the PPR-Tree is consistently better than the R*-Tree approaches for small, large and mixed snapshot queries. An example is shown in figure 18 for the mixed snapshot queries. We used the 150% PPR-Tree, the 1% R*-Tree and the piecewise R*-Tree. The interesting result here is that the piecewise approach [21] is much worse than the no splits approach. The benefit from

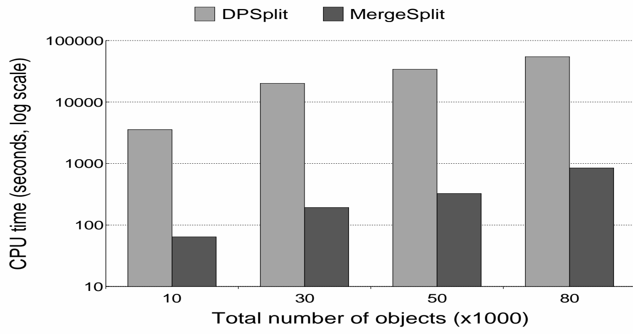

Fig. 11. CPU time for object split algorithms using random datasets.

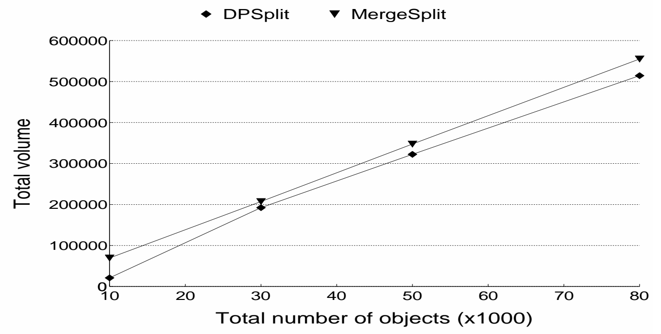

Fig. 12. Total volume for object split algorithms using random datasets.

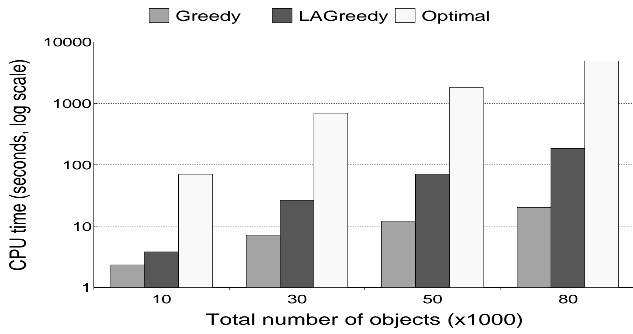

Fig. 13. CPU time for split distribution algorithms using random datasets.

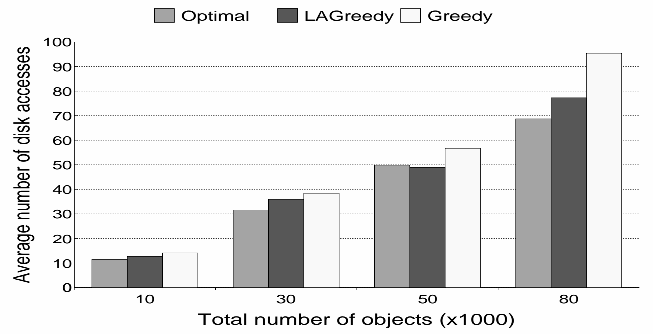

Fig. 14. Mixed snapshot queries using random datasets.

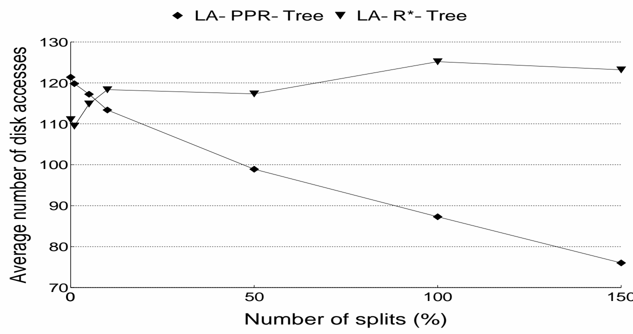

Fig. 15. Small range queries using the 50k random dataset.

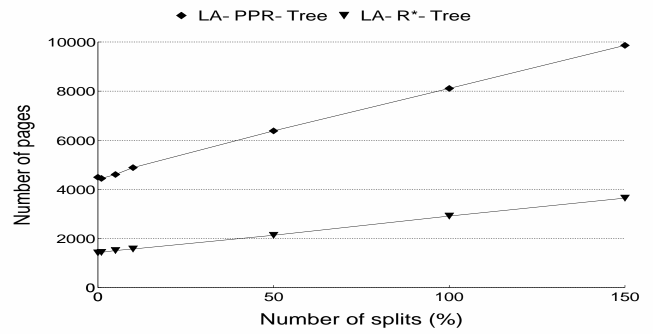

Fig. 16. Total space needed using the 50k random dataset.

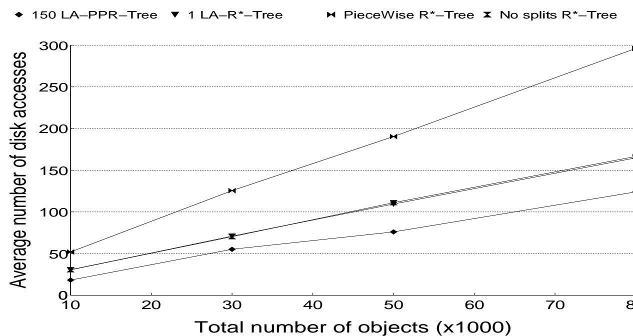

Fig. 17. Small range queries using random datasets.

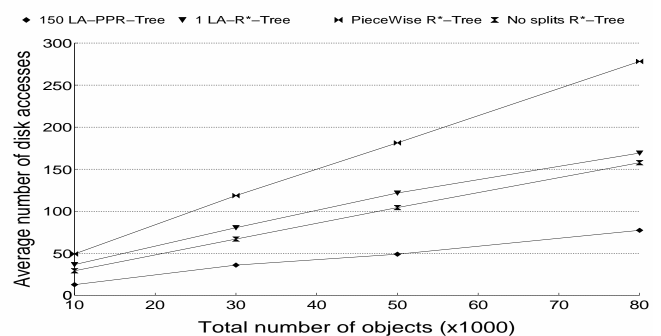

Fig. 18. Mixed snapshot queries using random datasets.

splitting the spatiotemporal objects ranges from 20% for small interval queries to more than 50% for snapshot queries. For the railway datasets we observe that the PPR-Tree is again superior in all cases. Due to lack of space the figures have been omitted.

VI. Related Work

Spatiotemporal data management has received increased interest in the last few years and a number of interesting articles appeared in this area. As a result, a number of new index methods for spatiotemporal data have been developed.

In [12] we discuss methods for indexing the history of spatial objects that move with a linear function of time. [21] examines indexing moving points that have piecewise linear trajectories. Two approaches are used, the Native Space Indexing where a 3-dimensional R-tree is used to index the line segments of object's trajectories and the Parametric Space Indexing. A similar idea is used in [5]. [20] presents methods to answer efficiently navigation and trajectory historical queries. This type of queries, though, are different than the topological queries examined in this paper. [18] addresses the problem of approximating spatial objects with small number of z-values, trying to balance the number of z-values with the extra space of the approximation.

Methods that can be used to index static spatiotemporal objects include [25], [17], [29], [5], [28]. These approaches are based either on the overlapping or on the multi-version approach for transforming a spatial structure into a partially persistent one. Another related paper is [7] where general structures to index spatiotemporal objects are discussed.

VII. Conclusions

In this paper we investigated the problem of indexing spatiotemporal data. We assume that objects move with general motion patterns and we are interested in answering snapshot and small range queries. The obvious approach for indexing spatiotemporal objects is to approximate each object with an MBR and use a spatial access method. However this approach is problematic due to extensive empty space and overlap. In this paper we show how to use artificial splits on a set of spatiotemporal objects in order to reduce overlap and empty space and improve query performance. We present algorithms to find good split positions for a single spatiotemporal object and methods to distribute a given number of splits between a collection of objects. Also, we discuss how to find a value for the number of splits that achieves a good trade-off between query time and space overhead. Experimental results validate the efficiency of the proposed methods. The combination of splitting algorithms and the PPR-Tree can achieve up to 50% better query time than the best previous alternative. An interesting avenue for future work is addressing the on-line version of the problem.

References

[1] P. K. Agarwal, L. Arge, and J. Erickson. Indexing moving points. *In Proc. of the 19th ACM Symp. on Principles of Database Systems (PODS)*, pages 175–186, 2000.

[2] B. Becker, T. Ohler S. Gschwind, B. Seeger, and P. Widmayer. An Asymptotically Optimal Multiversion B-Tree. *VLDB Journal 5(4)*, pages 264–275, 1996.

[3] N. Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. The R* - tree: An Efficient and Robust Access Method for Points and Rectangles. *Proceedings of ACM SIGMOD*, pages 220–231, June 1990.

[4] F. Burton, J. Kollias, V. Kollias, and D. Matsakis. Implementation of overlapping btrees for time and space efficient representation of collection of similar files. *The Computer Journal, Vol.33, No.3*, pages 279–280, 1990.

[5] M. Cai and P. Revesz. Parametric r-tree: An index structure for moving objects. In *Proc. of the COMAD*, 2000.

[6] J. Driscoll, N. Sarnak, D. Sleator, and R.E. Tarjan. Making Data Structures Persistent. *Journal of Computer and System Sciences, Vol. 38, No. 1*, pages 86–124, 1989.

[7] R. Guting, M. Bohlen, M. Erwig, C.Jensen, N. Lorentzos, M. Schneider, and M. Vazirgiannis. A Foundation for Representing and Querying Moving Objects. *In ACM TODS, Vol. 25, No 1*, pages 1–42, 2000.

[8] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proc. of ACM SIGMOD*, pages 47–57, 1984.

[9] I. Kamel and C. Faloutsos. Hilbert R-tree: An Improved R-tree Using Fractals. *Proceedings of VLDB*, pages 500–510, September 1994.

[10] G. Kollios, D. Gunopulos, and V. Tsotras. Indexing Animated Objects. *In Proc. 5th Int. MIS Workshop, Palm Springs Desert, CA*, 1999.

[11] G. Kollios, D. Gunopulos, and V. Tsotras. On Indexing Mobile Objects. *In Proc. of the 18th ACM Symp. on Principles of Database Systems (PODS)*, pages 261–272, June 1999.

[12] G. Kollios, D. Gunopulos, V. Tsotras, A. Delis, and M. Hadjieleftheriou. Indexing Animated Objects Using Spatio-Temporal Access Methods. *IEEE Trans. Knowledge and Data Engineering*, pages 742–777, September 2001.

[13] C. Kolovson and M. Stonebraker. Segment Indexes: Dynamic indexing techniques for multi-dimensional interval data. In *Proc. of ACM SIGMOD*, pages 138–147, 1991.

[14] A. Kumar, V.J. Tsotras, and C. Faloutsos. Designing access methods for bitemporal databases. *IEEE Trans. Knowledge and Data Engineering*, 10(1):1–20, 1998.

[15] S.T. Leutenegger, M.A. Lopez, and J.M. Edgington. STR: A simple and efficient algorithm for r-tree packing. In *Proc. of IEEE ICDE*, 1997.

[16] D. Lomet and B. Salzberg. Access Methods for Multiversion Data. *In Proceedings of ACM SIGMOD Conf.,Portland, Oregon*, pages 315–324, 1989.

[17] M. Nascimento and J. Silva. Towards historical r-trees. *Proc. of SAC*, 1998.

[18] J. A. Orenstein. Redundancy in spatial databases. In *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data*, pages 294–305, Portland, Oregon, 31 May–2 June 1989.

[19] B.-U. Pagel, H.-W. Six, H. Toben, and P. Widmayer. Towards an analysis of range query performance in spatial data structures. In *Proc. of ACM PODS*, pages 214–221, 1993.

[20] D. Pfoser, C. Jensen, and Y. Theodoridis. Novel Approaches in Query Processing for Moving Objects. *In Proceedings of VLDB, Cairo Egypt*, September 2000.

[21] K. Porkaew, I. Lazaridis, and S. Mehrotra. Querying mobile objects in spatio-temporal databases. In *Proc. of 7th SSTD*, July 2001.

[22] S. Saltenis and C. Jensen. Indexing of Moving Objects for Location-Based Services. *To Appear in Proc. of IEEE ICDE*, 2002.

[23] S. Saltenis, C. Jensen, S. Leutenegger, and Mario A. Lopez. Indexing the Positions of Continuously Moving Objects. *In Proceedings of the ACM SIGMOD*, pages 331–342, May 2000.

[24] B. Salzberg and V. Tsotras. Comparison of access methods for time-evolving data. *ACM Computing Surveys*, 31(2):158–221, 1999.

[25] Y. Tao and D. Papadias. Mv3r-tree: a spatio-temporal access method for timestamp and interval queries. In *Proc. of the VLDB*, 2001.

[26] Y. Tao and D. Papadias. Cost models for overlapping and multi-version structures. In *Proc. of IEEE ICDE*, 2002.

[27] Y. Theodoridis and T. Sellis. A model for the prediction of R-tree performance. In *Proc. of ACM PODS*, pages 161–171, 1996.

[28] Y. Theodoridis, T. Sellis, A. Papadopoulos, and Y. Manolopoulos. Specifications for efficient indexing in spatiotemporal databases. *In Proc. of 11th Int. Conf. on SSDBMs*, pages 123–132, 1998.

[29] T. Tzouramanis, M. Vassilakopoulos, and Y. Manolopoulos. Overlapping linear quadtrees and spatio-temporal query processing. *The Computer Journal 43(3)*, pages 325–343, 2000.

[30] P.J. Varman and R.M. Verma. An Efficient Multiversion Access Structure. *IEEE Transactions on Knowledge and Data Engineering, Vol. 9, No 3.*, pages 391–409, 1997.