Efficient Indexing of Spatiotemporal Objects

Marios Hadjieleftheriou
University of California, Riverside
marioh@cs.ucr.edu

George Kollios
Boston University
gkollios@cs.bu.edu

Vassilis J. Tsotras
University of California, Riverside
tsotras@cs.ucr.edu

Dimitrios Gunopulos
University of California, Riverside
dg@cs.ucr.edu

Abstract

Spatiotemporal objects, i.e., objects which change their position and/or extent over time appear in many applications. In this paper we examine the problem of indexing large volumes of such data. Important in this environment is how the spatiotemporal objects move and/or change. We consider a rather general case where object movements/changes are defined by combinations of polynomial functions. We further concentrate on "snapshot" as well as small "interval" queries as these are quite common when examining the history of the gathered data. The obvious approach that approximates each spatiotemporal object by an MBR 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 ways 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 give an optimal algorithm and a heuristic for splitting a given spatiotemporal object in a predefined number of pieces. Then, given an upper bound on the total number of possible splits, we present three algorithms that decide how the splits are distributed among all the objects so that the total empty space is minimized. The number of splits cannot be increased indefinitely since the extra objects will eventually affect query performance. Using a query cost model, our algorithms can identify a good number of splits, as well. Finally, extensive experiments are presented showing the performance gains of our methods against straightforward indexing approaches.

1 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 [36, 15, 14, 28, 38, 3, 24, 25, 16, 32, 36]. Two variations of the problem are examined: approaches that optimize queries about the future positions of spatiotemporal objects [15, 28, 3, 25, 27] and those that optimize historical queries [36, 38, 14, 21, 24, 25, 16, 32] (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 non-decreasing 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 $alive$.

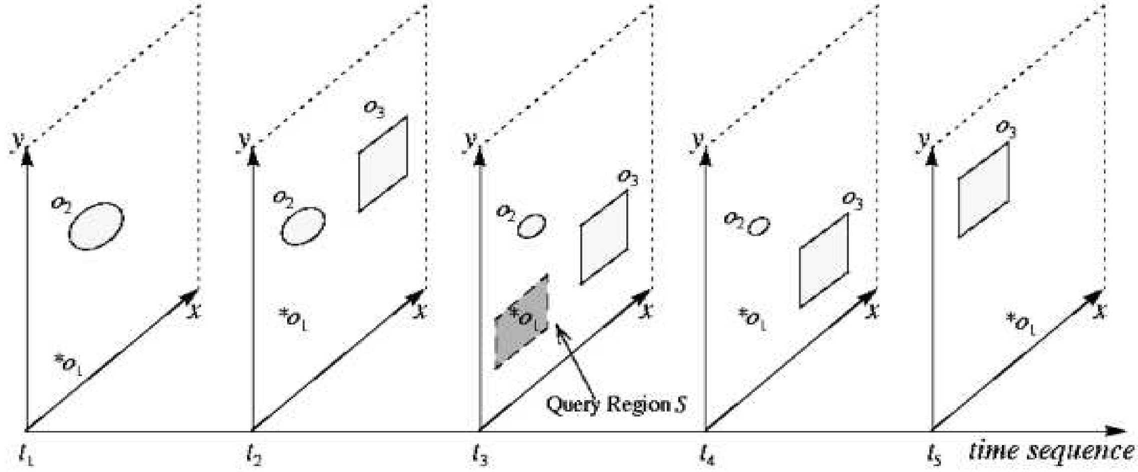

Figure 1: An example of spatiotemporal object evolution.

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 [11] or its variants [5]). 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 rectangle (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 [17, 18, 29, 32, 36, 16]. Better interval clustering can be achieved by using "packed" R-Trees (like the Hilbert R-Tree [13] or the STR-Tree [19]); another idea is to perform interval fragmentation using the Segment R-Tree [17]. However, the query performance is not greatly improved [16].

Another approach is to exploit the monotonicity of the temporal dimension, and transform a 2-dimensional spatial access method to become partially persistent [38, 21, 14, 16, 32]. A partially persistent structure "logically" stores all its past states and allows updates only to its most current state [9, 20, 4, 39, 18, 29]. 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 [6] and multi-version approaches [9]. In the overlapping approach [21, 38], 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 [29]. Conceptually, the multi-version approach [20, 4, 39, 18, 32] 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) [18, 32] (a short description of a PPR-Tree appears in the next section).

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 an 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 presented).

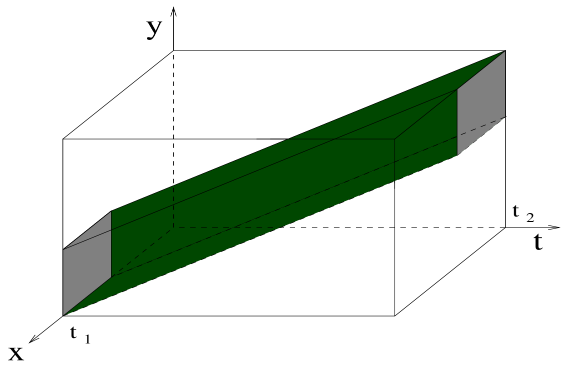

Figure 2: A spatiotemporal object in linear motion.

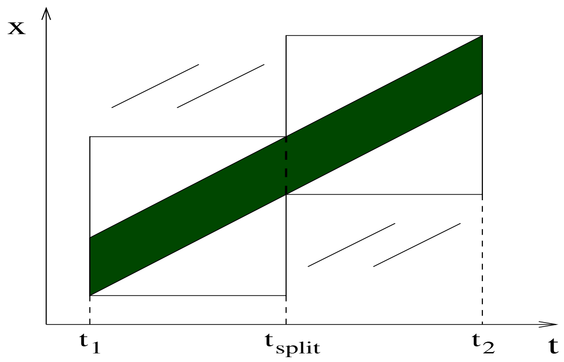

Figure 3: An example of splitting an object.

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 [23]. 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 is reduced. Therefore, it is expected that the PPR-Tree performance for snapshot and small interval queries will be improved.

In [16] 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* property which holds for linearly moving/changing objects. The monotonicity property states that given a spatiotemporal object 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 using combinations of polynomial functions. Using such functions we can describe quite general classes of object movements/changes. Unfortunately, the previous monotonicity property does not hold. An example is shown in figure 4. One split will give much less gain in empty space than two.

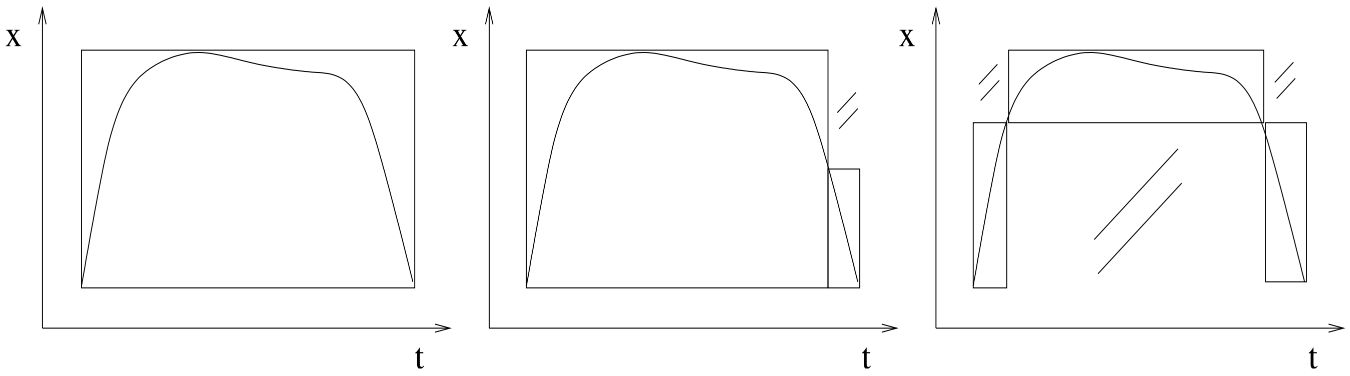

Figure 4: An example where the monotonicity property does not hold.

Hence, new approaches are needed. We first present a dynamic programming algorithm and a heuristic for deciding how to apply splits on a given, general spatiotemporal object. Furthermore, we provide a dynamic programming algorithm for optimally distributing a total number of splits among a collection of general spatiotemporal objects. Finally, we implement two greedy algorithms that give results close to the optimal algorithm but with huge gain in terms of running time. The first algorithm assigns each split to the object that yields the biggest gain in empty space at any step. The second algorithm takes into account the fact that some objects might not follow the monotonicity property. It looks ahead for the biggest gain in empty space given k splits per object at every step, thus, producing a nearly optimal result very fast. While by using more splits the gain is increased, after a while the increase will be minimal. A related problem is how to decide on the number of artificial splits. Assuming that a model for predicting the query cost of the index method used, is available [35, 33], 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 3D R*-Tree [5]. 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) [24, 25], and (ii) when the motion of each object can be represented as a set of linear functions (piecewise linear trajectories) [10, 16]. For the case that points move with linear functions of time, extensions to the R-Tree have been proposed (Parametric R-Tree [7] and the PSI approach in [25]). 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 2 formalizes the notion of general movements/changes and provides background on the partially persistent R-Tree. Section 3 elaborates on how a general spatiotemporal object should be split given a number of splits and how to distribute splits among a collection of spatiotemporal objects. Section 4 discusses how to use analytical models to find a good number of splits for a given dataset. Section 5 contains experimental results and section 6 reviews related work. Finally, section 7 concludes the paper.

2 Preliminaries

2.1 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: $x = F_x(t)$, $y = F_y(t)$, $t \in [t_i, t_j)$. We define a representation of a spatiotemporal object O as:

Definition 1: A spatiotemporal object O is 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 the position of such objects is more demanding and cannot be approximated by linear functions, since the number of segments required cannot be bounded. A direct extension of the above definition could use tuples with polynomial functions of greater degree to approximate general movements.

An example of a point moving on the (x, t) plane with the corresponding functions describing its movement is shown in figure 5. The following three tuples are used to represent it: $O = \{(0, 20, 3, -0.0045, 0, 0, 35)$, $(20, 30, 2, -1, 50, -600)$, $(30, 35, 2, -1, 65, -1050.25)\}$. The last two polynomials have been expanded and the tuple format is the following: $T : (t_s, t_e, d, c_d, \ldots, c_0)$; here t_s is the starting time, t_e is the ending time, d is the degree of the function and $c_d, \ldots, c_0$ are the coefficients. 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 and choosing only monotonically increasing or decreasing functions, 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. The use of monotonic functions allows us to find the MBR of any movement quite easily just by calculating the position of the object at the end points of every time interval and finding the global minimum and maximum.

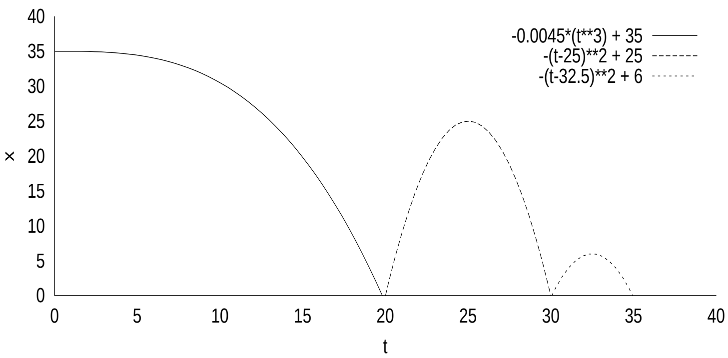

Figure 5: A moving point and a corresponding set of polynomial functions representing the movement.

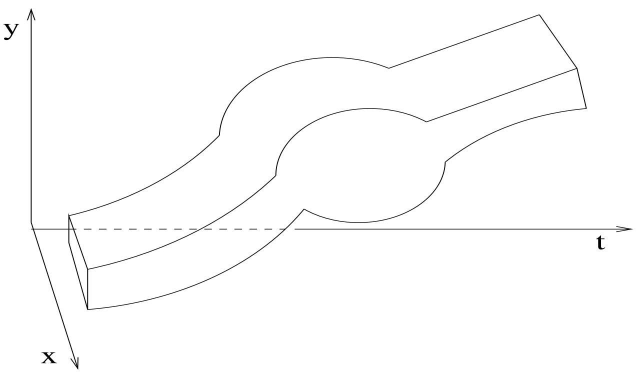

Figure 6: A moving object that follows a general trajectory while changing shape.

The advantage of this approach is two-fold. It is storage efficient, since few tuples are required for the representation of any movement in the general case. It guarantees that for a given object O, between any two consecutive time instants t_k, t_{k+1} the movement of the object is bounded by the MBR defined by the points $[x_k, F_x(t_k)], [x_{k+1}, F_x(t_{k+1})]$. These observations have straightforward extensions to 2-dimensional movements.

2.2 The Partially Persistent R-Tree

Consider the example in figure 1 and assume that the objects at time t_1 are indexed by a 2-dimensional R-Tree. As time advances, this 2D R-Tree evolves, by applying the updates (object additions/deletions) as they occur. Storing this 2D R-Tree evolution corresponds to making a 2D R-Tree partially persistent. The following discussion is based on [18]. While conceptually the partially persistent R-Tree (PPR-Tree) [18] 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, 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. The various roots can be easily accessed through a linear array called the **root***. Each entry in the **root*** contains a time interval and a pointer to the root that is responsible for that interval.

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. A leaf or a directory node is called *alive* if it has not been *split*. With the exception of root nodes, for all times that a node is alive it must have at least D alive records ($D < B$, where B is the maximum node capacity). 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. The PPR-Tree is created incrementally following the update sequence. Consider an update (insertion or deletion) at time t_i. To process this update the PPR-Tree is searched to locate the target leaf node where the update must be applied. This step is carried out by taking into account the lifetime intervals of the index and the data records visited. This implies that the search visits records that are alive at time t_i. After locating the target leaf node, an insertion update adds a data record with an interval $[t_i, now)$ to the target leaf node (now is a variable representing the ever increasing current time). A deletion update will update the deletion-time of a data record from now to t_i.

An update leads to a *structural* change if at least one new node is created. *Non-structural* are those updates which are handled within an existing node. An insertion update triggers a structural change if the target leaf node already has B records. A deletion update triggers a structural change if the resulting node ends up having less than D alive records as a result of the deletion. The former structural change is a *node overflow*; the latter is a *weak version underflow* [4]. Node overflow and weak version underflow need special handling: a *split* is performed on the target leaf node. This is reminiscent of the time-split operation reported in [20] and the page copying concept proposed in [37]. Splitting a node x at time t is performed by copying to a new node y the records alive in node x at t. Node x is considered *dead* after time t.

To avoid having a structural change on node y soon, when a new node is created the number of alive records must be in the range $D + e$ and $B - e$ (where e is a predetermined constant). This allows at least e non-structural changes on this node before a new structural change occurs. Thus before the new node is incorporated in the structure it may have to be merged with another node (this happens if y has less than $D + e$ alive records and is called a *strong version underflow*), or, "key-split" into two nodes (if y has more than $B - e$ alive nodes, i.e., a *strong version overflow*). For details we refer to [18, 39, 4].

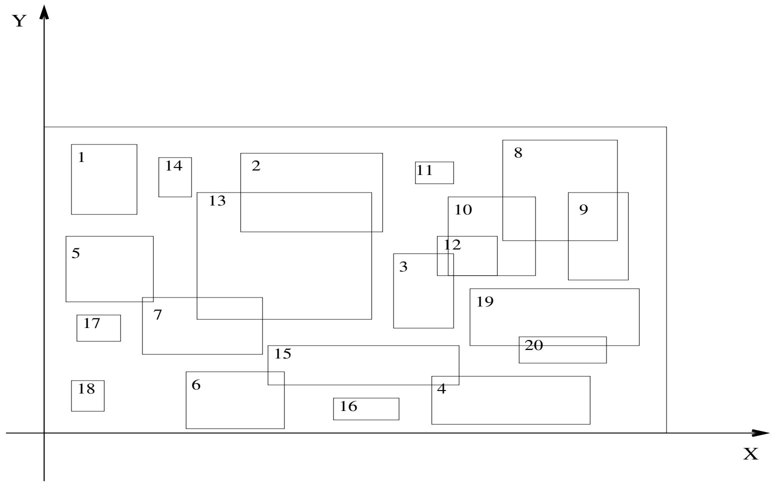

Figure 7: Various object MBRs.

An example of a PPR-Tree is shown in figure 9 using the evolution presented in figure 7 and figure 8. In particular, figure 7 shows the MBRs of 20 objects (numbered from 1 to 20) that appeared in an evolution while figure 8 depicts the lifetimes of these objects. For simplicity, the objects do not change extent neither move during their lifetimes. Here $B = 5$, $D = 2$ and e is set to 1. The **root*** entries show the time intervals associated with each pointer and the pointers to the root nodes of the PPR-Tree. Similarly, index nodes depict the time intervals and the corresponding pointers to the next level of the tree. For clarity, data nodes show only the stored object ids (and not their lifetimes). Note that an object can be stored in more than one data page. For example object 14 is stored in five data pages since it has a long lifetime.

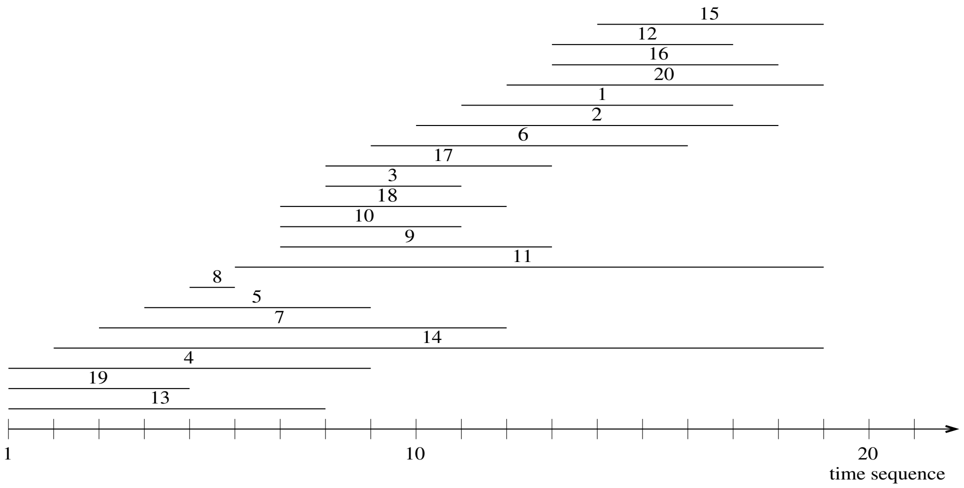

Figure 8: Corresponding object lifetimes.

Answering a query about region S and time t has two parts. First, the root which is alive at t is found. This part is conceptually equivalent to accessing the root of 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. Answering a query that specifies a time interval $[t, t')$ is similar. First, all roots with lifetime intervals intersecting $[t, t')$ are found and the procedure continues in the same manner. Since the PPR-Tree is a graph, some nodes are accessible by multiple

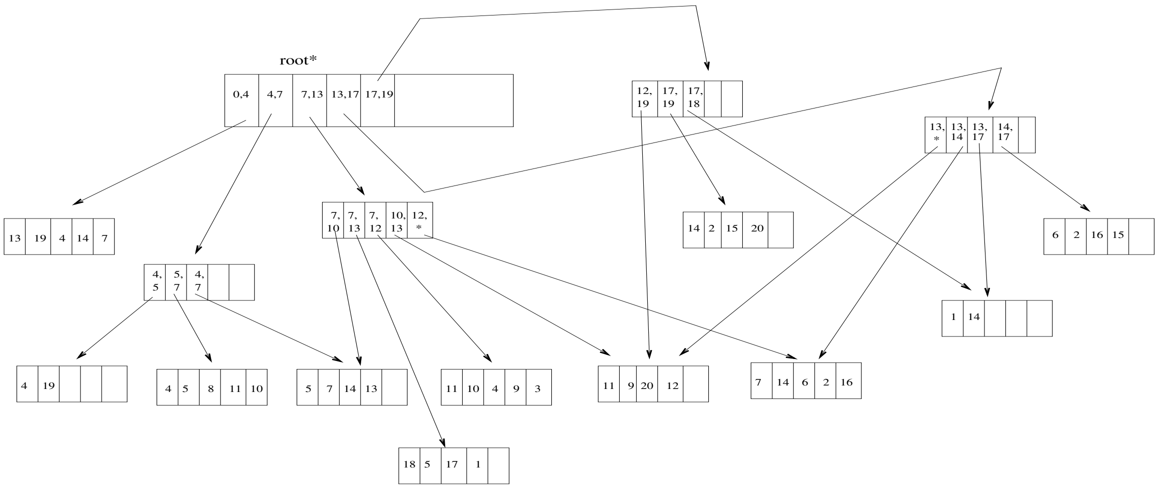

Figure 9: The PPR-Tree created from the above object evolution.

roots. Re-accessing nodes can be avoided by keeping a list of visited nodes. We should mention that another approach is to use the MV3R-tree [32] instead of the PPR-tree, but the results will be qualitatively the same.

An object is represented in the PPR-Tree as 2-dimensional MBR and a time interval. Next, we discuss how we can improve the representation of a general spatiotemporal object such that the query performance of the PPR-Tree is improved.

3 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 the 3D R-Tree or the PPR-Tree is used). A better approach is to represent the object using multiple boxes. An example is shown in figure 10, where only one dimension is shown, for simplicity. Using three boxes a better approximation is obtained and the empty space is reduced. The object is split into three consecutive objects and each one is approximated with a smaller bounding box. 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.

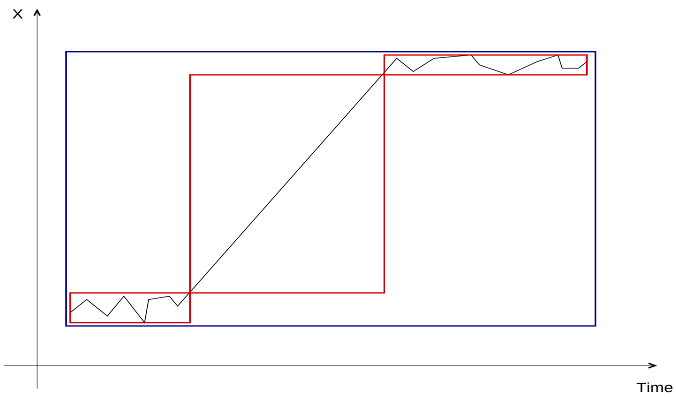

Figure 10: A 1-dimensional example of representing an object with one and three MBRs.

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. First, we are given an object and a number of splits and we have to find how to split the object such that the maximum possible gain in empty space is obtained. For the second problem, we have a collection of objects and a number of available splits and we try to distribute the splits among all objects in order to optimize the query performance of the index.

3.1 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 an way that minimizes the total volume of its representation. As we discussed above, we use splits in the form of artificial updates.

3.1.1 An Optimal Algorithm (DPSplit)

Let $OPTV[i, l]$ 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:

$$OPTV[i, l] = \min_{0 \leq j < i} \{OPTV[j, l-1] + Vol[j+1, i]\}$$

where $Vol[j, i]$ is the volume of the MBR that contains the part of the spatiotemporal object between t_j and t_i. 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 it with 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 $OPTV[n, k]$.

Theorem 1: Splitting one object optimally using k splits can be done in $O(n^2k)$, where the lifetime of the object is $[t_0, t_n)$.
Proof: We have to compute the nk values of the array $OPTV[i, j]$. Each value in the array can be found by computing the minimum of n values using the formula above.

3.1.2 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 11.) The running time of the algorithm is $O(n \log n)$. 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.

3.2 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 each object is allocated.

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 $0 \leq i < n$ 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 MBR.

Figure 11: The greedy heuristic.

3.2.1 An Optimal Algorithm

Assuming an ordering on the spatiotemporal objects (each object gets a number between 1 and N), we observe that:

$$OPTK[i,l] = \min_{0 \leq j \leq i} \{OPTK[i-1, l-j] + OVol[i,j]\}$$

where $OPTK[i,l]$ is the minimum total space for the first i objects with optimal l splits, and $OVol[i,j]$ is the total volume for approximating the ith object using j splits (e.g. with $j+1$ boxes.)

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 $O(N^2K)$. To compute the optimal solution first we should know the optimal solutions for each object, which can be found by using the dynamic programming algorithm presented in section 3.1. Hence, the following theorem holds:

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

Note however, that the real objective of the splitting algorithms is not to minimize the total volume *per se*, but to reduce the cost of answering a query from a predefined set of queries. 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. We discuss possible ways to express this function and choose a good value for the total number of splits in section 4.

3.2.2 The Greedy Algorithm

The dynamic programming algorithm described above is quadratic to the number of spatiotemporal objects. 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 a greedy algorithm that allocates splits one at a time to the object that if split once more would yield the maximum possible total volume reduction. The algorithm is shown in figure 12. The complexity of the main loop is $O(K)$ and the complexity of the algorithm is $O(K + N \log N)$.

3.2.3 The Look-Ahead Greedy Algorithm (LAGreedy)

The above algorithm also requires the calculation of the minimum volume for a specific number of splits for some of the objects, depending on the particular dataset. The result of this algorithm will not be optimal in the general case. One problem is the following: consider an object that if split once gives a very small improvement in empty

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

1. Store in a priority queue all objects according to the total volume change if one split is assigned to each one.
2. For K iterations: Remove the top element of the queue. Assign the split to the corresponding object. Calculate the total volume change if an extra split was used. Place the object back in the queue.
3. Remove all elements from the priority queue and calculate the total volume according to the splits assigned to each object.

Figure 12: Greedy Algorithm.

space but if split twice most of its empty space is removed (see figure 4 for an example). Using the above algorithm most probably this object will not be allocated a split, because the first split is poor and other objects will be chosen before it. However, if we allow the algorithm to consider more than one splits for every object at a time, the possibility for this object to be split 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 used all at once (look-ahead-2, look-ahead-3, etc). The algorithm is shown in figure 13.

The look-ahead-2 algorithm works as follows. First, all splits are allocated one by one in a greedy fashion, as before. Then, one new priority queue PQ_{la1} is created, which sorts the objects by the minimum gain offered by the last split allocated to them, and a second priority queue PQ_{la2} which sorts the objects by the maximum gain if two more splits are allocated to each one. 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 two splits are allocated to the object of PQ_{la2} and they are removed from the others. The queues are updated and the same procedure continues until there is no more change in the distribution of splits.

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

1. Allocate splits be calling the *Greedy Algorithm*.
   PQ_{la1} sorts objects by minimum gain given by the last split.
   PQ_{la2} sorts objects by maximum gain given with two more splits.
2. While there is a change: 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 of O_3 with two more splits is bigger than the combined gain of O_1 and O_2 from their last splits, redistribute the splits and update the priority queues.

Figure 13: LAGreedy Algorithm.

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

4 Finding the Number of Splits

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. If the number of splits is small, the query performance may be much worse than the performance that we can get by using the optimal number of splits. On the other hand, if the number of splits is much larger than the optimal one, we create many artificial objects that waste disk space. Also, the query performance may deteriorate if the height of the tree is increased. In this section, we discuss methods for automatically computing a good value for this parameter. This is not always easy, since the optimal number of splits depends on the distribution and shape of the spatiotemporal objects and the distribution of queries. We briefly discuss two general methods to optimize the performance of the splitting algorithms (more details will appear in the full version of the paper). The first method is based on using analytical models to predict the performance of the index. The second method, experimentally evaluates a small sample of the dataset over a representative set of queries and decides the best total number of splits.

4.1 Using Analytical Models

The basic idea here is to use an analytical model and predict the performance of an index using this model. For a given number of splits, we compute the best distribution of splits and estimate some statistics about the datasets generated after the splits. We use the statistics as an input to the analytical model and we get a prediction on the number of disk accesses required to answer a random query from a query distribution. Thus, instead of trying to minimize the total volume, we try to minimize the average query cost which is our ultimate goal.

In particular, for the R-Tree approach, we can use one of the proposed analytical models that estimate the I/O complexity of an R-Tree given a dataset and a query [35, 26]. These papers provide formulas that use statistical information of the datasets to predict the performance of a (uncreated) R-Tree. In our case, we evaluate the formula in each iteration of the splitting algorithms in section 3 and we keep the number of splits that give the smallest access overhead. Depending on the dataset and the query, a new split may result in a higher or lower access overhead. Therefore, we need again to give an upper bound on the number of possible splits, but the splits that we choose at the end can be much smaller. In the extreme case that the dataset is static, the choice will be 0 splits. For the PPR-Tree, a formula proposed recently in [33] can be used. Finally, another approach is to use non-parametric statistical summaries of the datasets (e.g. spatial histograms) [12, 2, 1, 40, 31].

4.2 Using Cross-Validation

Another way to find the best number of splits among a set of possible values is by cross validation. For each number of splits, an index is created and a set of (the same) 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. For example, if the dataset contains N objects, the sample size is s, and the best number of splits for the sample is l, $\frac{Nl}{s}$ number of splits must be chosen for the full dataset. Note, that if the sample is small, the index can be built in main memory.

5 Experimental Results

To test our algorithms we created four random datasets (uniform) of various sizes with moving rectangles in 2D 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 [25]. This method resulted in a number of splits about 400% of the total number of objects. Finally, we used the 3D 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 [16], 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 [16]. Details about all datasets are presented in Tables 1 and 2.

For the moving rectangles time extents from 0 to 999 timestamps. 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. 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 timestamps.

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 [32]. 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 Tables 3 and 4. For all experiments the buffer was reset before the execution of any query. All experiments where run on an Intel Pentium III 1GHz personal computer, with 1GB of main memory.

5.1 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 14, 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

Dataset	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%

Table 1: Random datasets.

Dataset	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 2: Railway datasets (skewed).

Query Set	Cardinality	Extents (%)	Duration (timestamps)
Tiny	1000	0.01-0.1	1
Small	1000	0.1-1	1
Mixed	1000	0.1-5	1
Large	1000	1-5	1

Table 3: The snapshot query sets.

Query Set	Cardinality	Extents (%)	Duration (timestamps)
Small	1000	0.1-1	1 - 10
Medium	1000	0.1-1	10 - 50

Table 4: The interval query sets.

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 15. Clearly, MergeSplit behaves very closely to DPSplit.

5.2 Comparison of Split Distribution Algorithms

Next, we evaluate the performance of the Greedy and LAGreedy algorithms, in comparison with the optimal dynamic 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 16. 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 17. For all the datasets that we tried, the LAGreedy algorithm performed as well as the optimal algorithm, while the Greedy approach was

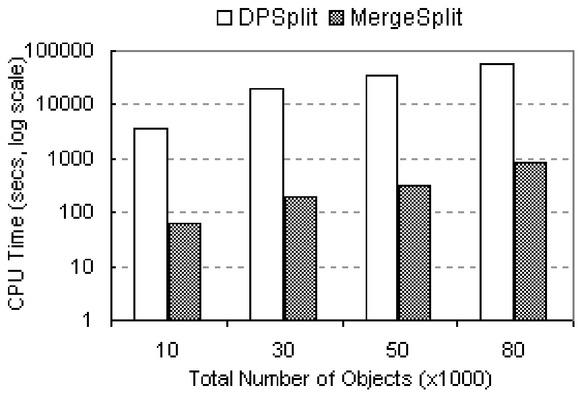

Figure 14: CPU time in seconds (logarithmic scale) for the *one object* split algorithms, using the random datasets.

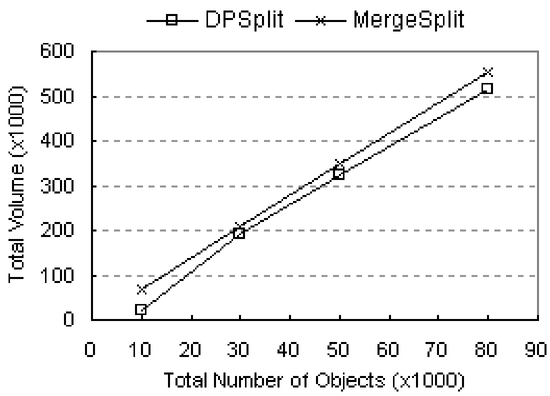

Figure 15: Total volume for the *one object* split algorithms and 50% splits distributed with the LAGreedy algorithm, using the random datasets.

always inferior.

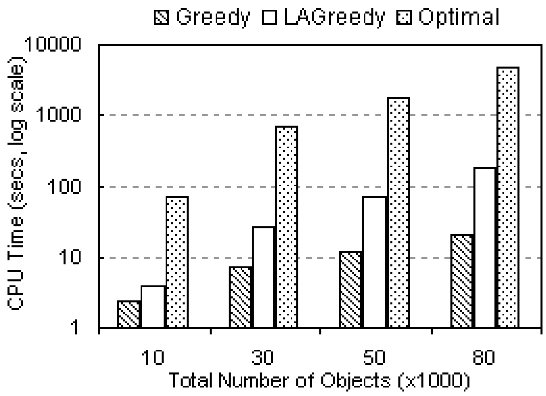

Figure 16: CPU time in seconds (logarithmic scale) for 50% splits distributed with all *split distribution* algorithms, using the random datasets.

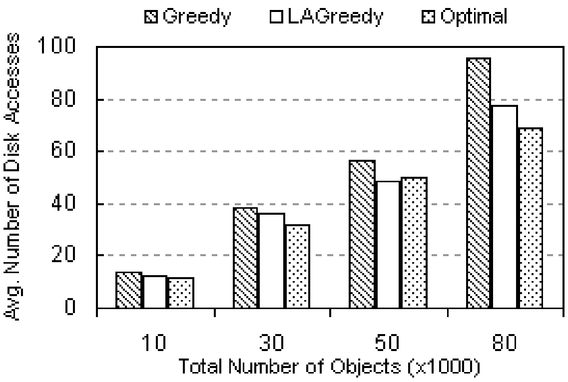

Figure 17: Average number of disk accesses for 150% splits distributed with all *split distribution* algorithms, using a PPR-Tree and mixed snapshot queries with the random datasets.

5.3 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 the 3D R*-Tree and the 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 18. 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 3D R*-Tree. For completeness, in figure 19 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 3D 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).

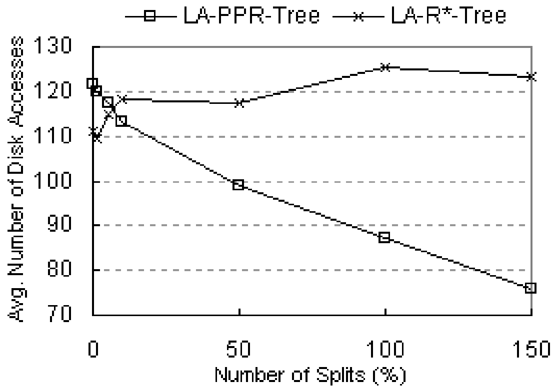

Figure 18: Average number of disk accesses for the PPR-Tree and R*-Tree, with various numbers of splits distributed with the LAGreedy algorithm, using small range queries and the 50k random dataset.

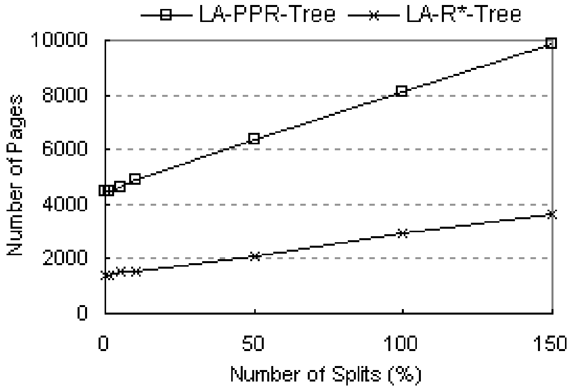

Figure 19: Number of pages for the PPR-Tree and R*-Tree, with various numbers of splits distributed with the LAGreedy algorithm, using the 50k random dataset.

5.4 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 normal R-Tree structures react when increasing the number and type of objects. For the small range queries the 3D R*-Tree is somewhat better for un-cut data and 1% up to 5% splits, while the PPR-Tree becomes much better when the number of splits increases. In figure 20 we plot the average number of disk accesses for small range queries and 150% splits for the PPR-Tree and 1% splits for the 3D R*-Tree, distributed with the LAGreedy algorithm. We also plot the performance of the 3D 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 3D R*-Tree approaches for small, large and mixed snapshot queries. An example is shown in figure 21 for the mixed snapshot queries. We used 150% splits for PPR-Tree, 1% splits for the 3D R*-Tree and the piecewise 3D R*-Tree. The interesting result here is that the piecewise approach [25] is much worse than the no splits approach. The benefit from splitting the spatiotemporal objects ranges from 20% for small interval queries to more than 50% for snapshot queries.

The results for the railway datasets are shown in figures 22 and 23. We observe that the PPR-Tree is again superior for all cases.

6 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.

The most related work with our paper includes [16, 25, 24]. In [16] we discuss methods for indexing the history of spatial objects that move with a linear function of time. In the current paper we address the problem of objects

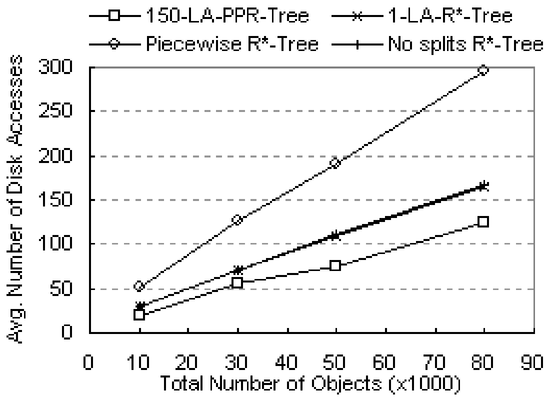

Figure 20: Average number of disk accesses for small range queries, using the random datasets.

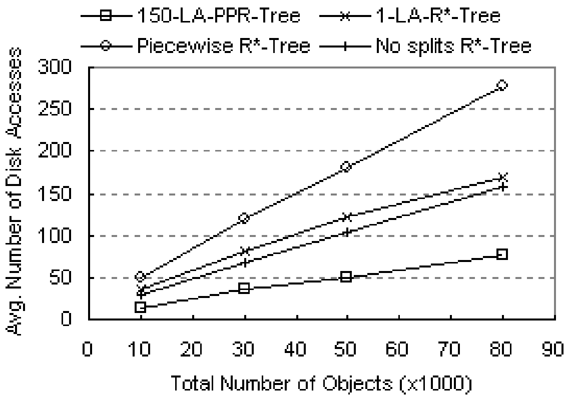

Figure 21: Average number of disk accesses for mixed snapshot queries, using the random datasets.

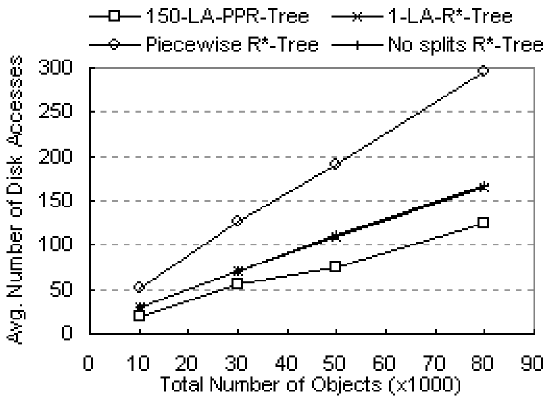

Figure 22: Average number of disk accesses for small range queries, using the railway datasets.

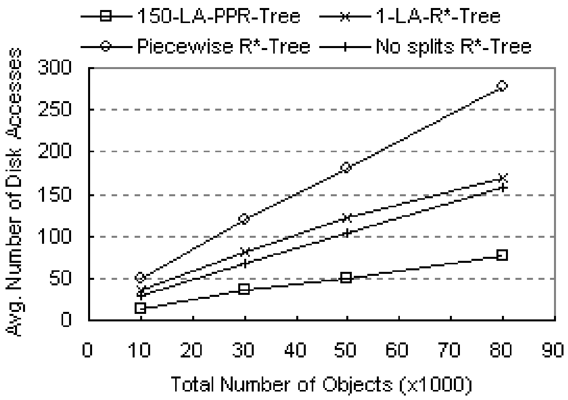

Figure 23: Average number of disk accesses for mixed snapshot queries, using the railway datasets.

moving with more complex functions. [25] examines indexing moving points that have piecewise linear trajectories. Two approaches are used, the Native Space Indexing where a 3D R-tree is used to index the line segments of object's trajectories and the Parametric Space Indexing, where a parametric approach to store the moving points is used. An idea similar to Parametric Space Indexing is used in [7]. [24] presents methods to answer efficiently navigational and trajectory historical queries. These queries are different than the topological queries examined here and therefore the methods presented in [24] will not be optimized for answering the type of queries that we investigate in the current paper. Another related paper is [22] that 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 [32, 21, 38, 7, 36]. These approaches are based either on the overlapping or on the multi-version approach for transforming a spatial structure in to a partially persistent one. Another related paper is [10] where general structures to index spatiotemporal objects are discussed.

Finally, methods to index the future location of moving points appeared in [15, 27, 28, 34, 8, 3]. These methods assume that the function that describes the future positions of each point is known. Three different approaches have been proposed for indexing objects in this environment for range and nearest neighbor queries. One is based on the space-time representation [34, 8, 30], the other using a dual transformation [15, 3] and the last one using time parameterized spatial structures (R-Trees) [28, 27].

7 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 efficiently snapshot and small interval spatiotemporal range queries. The obvious approach to index spatiotemporal objects is to approximate each object with a minimum bounding (hyper-)rectangle (MBR) and use a spatial access method to index these MBRs. However this approach is problematic due to extensive empty space and overlap. In this paper we show how to split a set of spatiotemporal objects in order to reduce this overlap and empty space and improve the query performance. We present algorithms to find good split positions for a single spatiotemporal object and methods to distribute a given number of splits to a collection of spatiotemporal objects. Also, we discuss how to find a good 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. Among the presented approaches, the two greedy algorithms, namely the MergeSplit and the LAGreedy, provide the best performance for a small processing overhead. 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. Furthermore, we plan to address the spatiotemporal indexing problem in an environment of high update rates that appears in various spatiotemporal applications. Using parallel and distributed spatiotemporal index methods is one promising approach.

References

[1] A. Aboulnaga and J. Naughton. Accurate estimation of the cost of spatial selections. In *Proc. of IEEE ICDE*, pages 123–134, 2000.

[2] S. Acharya, V. Poosala, and S. Ramaswamy. Selectivity estimation in spatial databases. In *Proc. of ACM SIGMOD*, pages 13–24, 1999.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

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

[8] H. D. Chon, D. Agrawal, and A. El Abbadi. Storage and retrieval of moving objects. In *Mobile Data Management*, pages 173–184, 2001.

[9] 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.

[10] 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.

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

[12] J. Jin, N. An, and A. Sivasubramaniam. Analyzing range queries on spatial data. In *16th International Conference on Data Engineering (ICDE' 00)*, pages 525–534, Washington - Brussels - Tokyo, March 2000. IEEE.

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

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

[15] 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.

[16] 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.

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

[18] 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.

[19] 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.

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

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

[22] 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.

[23] 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.

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

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

[26] G. Proietti and C. Faloutsos. I/O complexity for range queries on region data stored using an R-tree. In *15th International Conference on Data Engineering (ICDE '99)*, pages 628–635, Washington - Brussels - Tokyo, March 1999. IEEE.

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

[28] 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.

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

[30] Z. Song and N. Roussopoulos. Hashing moving objects. In *Mobile Data Management*, pages 161–172, 2001.

[31] Chengyu Sun, Divyakant Agrawal, and Amr El Abbadi. Exploring spatial datasets with histograms. In *Proc. of IEEE ICDE*, 2002.

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

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

[34] J. Tayeb, Ö. Ulusoy, and O. Wolfson. A quadtree-based dynamic attribute indexing method. *The Computer Journal*, 41(3):185–200, 1998.

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

[36] 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.

[37] V. Tsotras and N. Kangelaris. The Snapshot Index, an I/O-Optimal Access Method for Timeslice Queries. *Information Systems, Vol. 20, No. 3*, pages 237–260, 1995.

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

[39] 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.

[40] M. Wang, J.S. Vitter, L. Lim, and S. Padmanabhan. Wavelet-based cost estimation for spatial queries. In *Proc. of SSTD*, pages 175–196, 2001.