The VLDB Journal manuscript No.
(will be inserted by the editor)

Indexing Spatio-temporal Archives

Marios Hadjieleftheriou[1], George Kollios[2], Vassilis J. Tsotras[1], Dimitrios Gunopulos[1]

[1] Computer Science Department, University of California, Riverside
Email: marioh,tostras,dg@cs.ucr.edu
[2] Computer Science Department, Boston University
Email: gkollios@cs.bu.edu

The date of receipt and acceptance will be inserted by the editor

Abstract Spatio-temporal objects — that is, objects that evolve over time — appear in many applications. Due to the nature of such applications, storing the evolution of objects through time in order to answer historical queries (queries that refer to past states of the evolution) requires a very large specialized database, what is termed in this article as a *spatio-temporal archive*. Efficient processing of historical queries on spatio-temporal archives requires equally sophisticated indexing schemes. Typical spatio-temporal indexing techniques represent the objects using minimum bounding regions (MBR) extended with a temporal dimension, which are then indexed using traditional multi-dimensional index structures. However, rough MBR approximations introduce excessive overlap between index nodes which deteriorates query performance. This article introduces a robust indexing scheme for answering spatio-temporal queries more efficiently. A number of algorithms and heuristics are elaborated, which can be used to preprocess a spatio-temporal archive in order to produce *finer object approximations* which, in combination with *a multi-version index structure*, will greatly improve query performance in comparison to the straightforward approaches. The proposed techniques introduce a query-efficiency vs. space tradeoff, that can help tune a structure according to available resources. Empirical observations for estimating the necessary amount of additional storage space required for improving query performance by a given factor are also provided. Moreover, heuristics for applying the proposed ideas in an online setting are discussed. Finally, a thorough experimental evaluation is conducted to show the merits of the proposed techniques.

Key words Spatio-temporal databases, Indexing, Moving objects, Trajectories

1 Introduction

Due to the widespread use of sensor devices, mobile devices, video cameras, etc., large quantities of spatio-temporal data are produced everyday. Examples of applications that generate such data include intelligent transportation systems (monitoring cars moving on road networks), satellite and GIS analysis systems (evolution of forest boundaries, fires, weather phenomena), cellular network applications, video surveillance systems, and more. Due to the massive space requirements of spatio-temporal datasets it is crucial to develop methods that enable efficient management of spatio-temporal objects as well as fast query processing.

A number of methods have been proposed recently to address indexing problems that appear in spatio-temporal environments. They can be divided in two major categories: Approaches that optimize queries about the *future positions* of spatio-temporal objects (including continuous queries on the location of moving objects) [25, 43, 1, 11, 39, 5, 42, 20, 46, 52, 53, 50, 48, 35, 40, 36, 32, 21], and those that optimize *historical* queries, i.e., queries about past states of the spatio-temporal evolution [54, 2, 56, 26, 33, 38, 39, 47, 9, 35]. This article concentrates on historical queries. Furthermore, it addresses the "off-line" version of the problem: Given the complete archive of a spatio-temporal evolution the purpose is to index it efficiently in order to answer the most prevalent spatio-temporal queries, namely range searches and nearest neighbors; for example: "Find all objects that appeared inside area S during time-interval $[t_1, t_2)$" and "Find the object nearest to point P at time t". Both *time period* as well as *time snapshot* queries are examined (queries about a time-interval or a time-instant).

A short version of this article appeared as "Efficient Indexing of Spatiotemporal Objects" in the Proceedings of Extending Database Technology 2002 [19]. This work was partially supported by NSF grants IIS-9907477, EIA-9983445, NSF IIS 9984729, NSF ITR 0220148, NSF IIS-0133825, NRDRP, and the U.S. Department of Defense.

For simplicity, in the rest of this article it is assumed that object evolution is taking place on a 2-dimensional universe (the extension to three dimensions is straightforward). An example of a spatio-temporal 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 ease of exposition, in the rest of this discussion, time is considered to be discrete (a succession of increasing integers). At time 1 objects o_1 (a point) and o_2 (a region) are inserted. At time 2, object o_3 is inserted while o_1 moves to a new position and o_2 shrinks. Object o_1 moves again at times 4 and 5; o_2 continues to shrink and disappears at time 5. Based on its behavior in the spatio-temporal evolution, each object is assigned a record with a *lifetime* interval $[t_i, t_j)$ inferred by the time instants that the object first appeared and disappeared, if ever. For example, the lifetime of o_2 is $L_2 = [1, 5)$. During its lifetime an object is termed *alive*. A spatio-temporal archive that stores the detailed evolution of the 2-dimensional universe consists of the complete update history of all the objects. A simple snapshot query is also illustrated in the same figure: "Find all objects that appeared inside area Q at time 3"; only object o_1 satisfies this query.

Spatio-temporal archives can be indexed, straightforwardly, using any multi-dimensional spatial access method like the R-tree and its variants [17, 45, 4]. An R-tree would approximate the whole spatio-temporal evolution of an object with one Minimum Bounding Region (MBR) that tightly encloses all the locations occupied by the object during its lifetime. While simple to deploy, these approaches do not take advantage of the special properties of the time dimension. Simplistic MBR approximations introduce a lot of empty volume, since objects that have long lifetimes correspond to very large MBRs. This, in turn, introduces excessive overlapping between index nodes and, therefore, leads to decreased query performance [27, 28, 44, 47, 54, 26]. A better approach for indexing a spatio-temporal archive is to use a multi-version index, like the MVR-tree [14, 30, 3, 57, 28, 44, 47]. This index "logically" stores all the past states of the data evolution and allows updates only to the most recent state. The MVR-tree divides long-lived objects into smaller, better manageable intervals by introducing a number of object copies. (However, the size of the index is kept linear to the number of object updates in the evolution.) A historical query is directed to the exact state acquired by the structure at the time that the query refers to; hence, the cost of answering the query is proportional only to the number of objects that the structure contained at that time.

The MVR-tree is an improvement over the straightforward R-tree approach, especially for short time-interval queries, since the individual parts of the MVR-tree corresponding to different versions of the data that will be accessed for answering a historical query are more compact than an R-tree that indexes the whole evolution. However, there is still space for substantial improvement. The indexing algorithms presented here are motivated by the following observation: Object clustering performed by the multi-version structure is affected by both the length of the lifetime of the objects and their spatial properties. Given that the trajectory archive is already available before building the index, it is possible to further improve index quality by creating finer object approximations that take into account the spatial dimensions of the objects as well as the temporal extent.

The rest of this paper presents various algorithms that can be applied on any spatio-temporal archive as a preprocessing step in order to improve index quality and, hence, query performance. The proposed algorithms produce finer object approximations that limit empty volume and index node overlapping. Given N object trajectories the input to the algorithms are N MBRs, one per object trajectory. The output is K MBRs (typically $K = O(N)$) that approximate the original objects more accurately. Some of the N original MBRs may still be among the resulting K (for objects for which no better approximation was necessary to be performed), while others will be split into sets of smaller, consecutive in time MBRs. The resulting MBRs will then be indexed with the aim of answering historical queries efficiently. Additionally, a number of heuristics are introduced that aid in deploying these algorithms only to the most suitable objects — that is, the ones that contribute the most to query performance improvement when approximated more accurately. The necessity of such heuristics will become apparent given that some of the proposed algorithms might have prohibitive computational cost for very large trajectory archives.

We opt at using a multi-version indexing scheme due to two reasons: First, index performance of multi-version structures does not deteriorate when increasing the absolute number of indexed objects, and second, due to the enhanced performance of these structures, especially for historical queries. Motivated by the query cost formula introduced by Pagel et al. [34] which states that the query performance of any MBR based index structure is directly proportional to the total volume, the total surface and the total number of indexed objects, it is argued that the proposed algorithms will benefit mostly the multi-version approach. This is due to the fact that introducing finer object approximations in the R-tree decreases the total volume of tree nodes but, at the same time, increases the total number of indexed objects, ripping off all benefits. On the other hand, in the MVR-tree the number of alive objects per time-instant remains constant independent of the number of MBRs used to approximate the objects, and this fact constitutes the biggest advantage of the multi-version approach. The experimental evaluation corroborates this claim.

This article extends the work by Hadjieleftheriou et al. [19] by providing various heuristics for improving

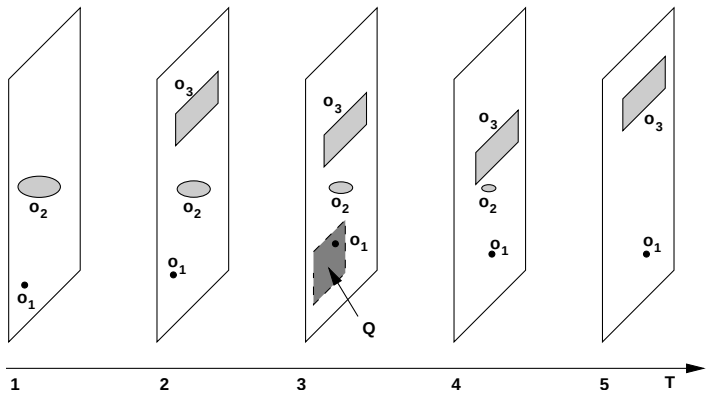


Fig. 1 A spatio-temporal evolution of moving objects.

overall performance, giving empirical observations for estimating the query-efficiency vs. space tradeoff before building the index structures, presenting a technique for indexing spatio-temporal archives online, and, finally, conducting a comprehensive experimental evaluation of the proposed approaches with an extended set of datasets.

In the next section a formal representation for spatio-temporal objects is discussed, and the basics of multi-version indexing structures are presented. Section 3 gives a detailed description of the proposed preprocessing algorithms. Section 4 presents various heuristics for improving the efficiency of the algorithms in the presence of very large trajectory archives. Section 5 presents a discussion on how the algorithms can be adapted for an "online" spatio-temporal archiving system. Section 6 presents a thorough experimental study, while Section 7 discusses related work. Finally, Section 8 concludes the paper.

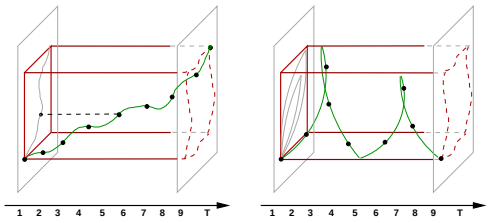
2 Preliminaries

This section gives the necessary background for the development of the proposed algorithms. Various models for representing spatio-temporal objects are discussed and a detailed presentation of the MVR-tree is given.

2.1 Representation of Spatio-temporal Objects

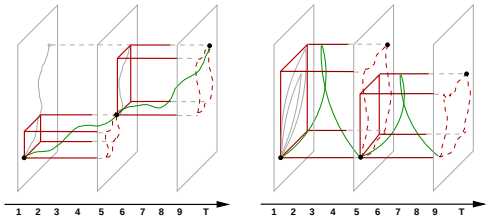
Object movements on a 2-dimensional plane can be modeled using various representations. Most applications represent a moving object as a collection of locations sampled every few seconds: The object movement is a set of tuples $\{< t_1, p_1 >, \ldots, < t_n, p_n >\}$. Previous work assumes that objects follow linear or piecewise linear trajectories. An object movement is then represented by a set of tuples $\{< [t_1, t_i), f_{x_1}(t), f_{y_1}(t) >, \ldots, < [t_j, t_n), f_{x_n}(t), f_{y_n}(t) >\}$, where t_1 is the object insertion time, t_n is the object deletion time, $t_i, \ldots, t_j$ are the intermediate time instants when the movement of the object changes characteristics and $f_{x_i}, f_{y_i} : \mathbb{R} \rightarrow \mathbb{R}$ are the corresponding linear functions. In general, the time-interval between two object updates is arbitrary, meaning that for any two consecutive time-instants t_i, t_j the corresponding time-interval $[t_i, t_j]$ can have an arbitrary length. In the degenerate case, the object can be represented by one movement function per time-instant of its lifetime, but in the rest of this article the more general representation is assumed, where objects issue updates more sparsely. If objects follow complex non-linear movements this approach becomes inefficient as it requires a large number of linear segments in order to accurately represent the movement. A better approach is to describe movements by using combinations of more general functions. All these approaches can be easily extended for higher dimensionality.

Regardless of the preferred representation, an object can always be perceived as the collection of the locations it occupied in space for every time-instant of its lifetime. This *exact representation* is clearly the most accurate description of the object's movement that can be attained but, at the same time, it has excessive space requirements. Nevertheless, it is amenable to substantial compression. Given a space constraint or a desired approximation accuracy, one can derive various approximations of a spatio-temporal object. On the one extreme, the object can be approximated with a single MBR — a 3-dimensional box whose height corresponds to the object's lifetime interval and its base to the tightest 2-dimensional MBR that encloses all locations occupied by the object (in the rest, this representation is referred to as the *single MBR approximation*). This simplistic approximation introduces unnecessary empty volume. For example, in Figure 2(a) a point starts at time 1 from the lower left corner of the plane and follows an irregular movement. It is evident that approximating the point's movement with a single MBR introduces too much empty volume (but the space requirements of this approximation are only the two end-points of the single MBR). A similar example is shown in Figure 2(b) where a point moves in circles. On the other extreme, the exact representation is equivalent to keeping one point per time instant of the object's lifetime (for a total of



(a) An irregular movement. (b) A circular movement.

Fig. 2 Single MBR approximations.



(a) An irregular movement (b) A circular movement

Fig. 3 Multiple MBR approximations.

nine points in Figures 2(a) and 2(b)). This representation yields the maximum reduction in empty volume, but also the maximum number of required points, increasing space consumption substantially. Note that these figures depict a moving point for simplicity. In general, objects that have extents that vary over time would require one MBR, instead of one point, per time instant of their lifetime.

Alternatively, spatio-temporal objects can be represented using multiple MBRs per object, which is a compromise in terms of approximation quality and space requirements. For example, in Figures 3(a) and 3(b) two smaller MBRs are used to represent each trajectory. It is obvious that the additional MBRs increase the representation's space requirements. It is also apparent that there are many different ways to decompose a trajectory into consecutive MBRs, each one yielding different reduction in empty volume. Given a trajectory and a space constraint (imposed as a limit on the total number of MBRs), an interesting problem is to find the set of MBRs that produces the best approximation possible, given some optimality criterion (for instance, the minimization of the total volume of the representation). Notice that, it is not clear by simple inspection which are the best two MBRs for optimally reducing the total volume of the representation of the object depicted in Figure 3(b). Finding the optimal MBRs, given a number k, is a difficult problem. Moreover, given a collection of object trajectories and a space constraint (as a total number of MBRs), another interesting problem is how to decide which objects need to be approximated more accurately, and which objects need not be considered at all. In other words, how to find the objects that would contribute the most in empty volume reduction if a larger number of MBRs were used to approximate them, given a fixed total number of MBRs that can be allocated to the archive. Solutions for these two problems are presented in Section 3.

2.2 The Multi-Version R-tree

This section presents the MVR-tree, a multi-version indexing structure based on R-trees. Since this structure is an essential component of the proposed techniques, it is presented here in detail.

Given a set of K 3-dimensional MBRs, corresponding to a total of $N < K$ object trajectories, we would like to index the MBRs using the MVR-tree. The MBRs are perceived by the tree as a set of K insertions and K deletions on 2-dimensional MBRs, since the temporal dimension has a special meaning for this structure. Essentially, an MBR with projection S on the $X-Y$ plane and a lifetime interval equal to $[t1, t2)$ on the temporal dimension represents a 2-dimensional MBR S that is inserted at time t_1 and marked as logically deleted at time t_2.

Consider the sequence of $2 \cdot K$ updates ordered by time. MBRs are inserted/deleted from the index following this order. Assume that an ephemeral R-tree is used to index the MBR projections S that appeared at $t = 0$. At the next time-instant that an update occurs (a new MBR appears or an existing MBR disappears), this R-tree is updated by inserting/deleting the corresponding MBR. As time proceeds, the R-tree evolves. The MVR-tree conceptually stores the evolution of the above ephemeral R-tree over time. Instead of storing a separate R-tree per time-instant — which would result in excessive space overhead — the MVR-tree embeds all the states of the ephemeral R-tree evolution into a graph structure that has linear overhead to the number of updates in the evolution.

The MVR-tree is a directed acyclic graph of nodes. Moreover, it has multiple root nodes each of which is responsible for recording a consecutive part of the ephemeral R-tree evolution (each root splits the evolution into disjoint time-intervals). The root nodes can be accessed through a linear array whose entries contain a time-interval and a pointer to the tree that is responsible for that interval. Data records in the leaf nodes of an MVR-tree maintain the temporal evolution of the ephemeral R-tree data objects. Each data record is thus extended to include the lifetime of the object: *insertion-time* and *deletion-time*. Similarly, index records in the directory nodes of an MVR-tree maintain the evolution of the corresponding index records of the ephemeral R-tree and are also augmented with the same fields.

The MVR-tree is created incrementally following the update sequence of the evolution. Consider an update at time t. The MVR-tree is traversed 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 leaf records encountered during the update. The search visits only the records whose lifetime fields contain t. After locating the target leaf node, an insertion adds the new data record with lifetime $[t, \infty)$, and a deletion updates the deletion-time of the corresponding data record from ∞ to t.

With the exception of root nodes, a node is called *alive* for all time instants that it contains at least $B \cdot P_v$ records that have not been marked as deleted, where $0 < P_v \leq 0.5$ and B is the node capacity. Otherwise, the node is considered *dead* and it cannot accommodate any more updates. This requirement enables clustering the objects that are alive at a given time instant in a small number of nodes, which in turn improves query I/O.

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 triggers a structural change if the target leaf node already has B records. A deletion triggers a structural change if the target node ends up having less than $B \cdot P_v$ alive records. The former structural change is a *node overflow*, while the latter is a *weak version underflow* [3].

Node overflow and weak version underflow require special handling. Overflows cause a *split* on the target leaf node. Splitting a node A at time t is performed by creating a new node A' and copying all the alive records from A to A'. Now, node A can be considered dead after time t; any queries that refer to times later than t will be directed to node A' instead of A. To avoid having a structural change occurring too soon on node A', the number of alive entries that it contains when it is created should be in the range $[B \cdot P_{svu}, B \cdot P_{svo}]$, where P_{svu} and P_{svo} are predetermined constants. This allows a constant number of non-structural changes on A' before a new structural change occurs. If A' has more than $B \cdot P_{svo}$ alive records a *strong version overflow* occurs; the node will have to be *key-split* into two new nodes. A key-split does not take object lifetimes into account. Instead, it divides the entries according to their spatial characteristics (e.g., by using the R*-tree splitting algorithm which tries to minimize spatial overlap). On the other hand, if A' has less than $B \cdot P_{svu}$ alive records a *strong version underflow* occurs; the node will have to be merged with a sibling node before it can be incorporated into the structure (Kumar et al. [28] discuss various merging policies in detail). An improved variant of the MVR-tree, called MV3R-tree, has been proposed by Tao and Papadias [47]. The proposed algorithms can be used with the MV3R-tree without any modifications.

A sample MVR-tree is shown in Figure 4. The horizontal axis represents the time dimension. Index and leaf nodes are shown with thick lines and are labeled with letters. Data boxes are shaded and numbered with increasing integers in order of their insertion time. Shaded projections of the data records on the plane are also depicted. An actual insertion/deletion operation is depicted by a white number, while an object copy (imposed by the MVR-tree version split policies) is depicted by a black number. For this tree let $B = 4$ and $P_v = P_{svu} = \frac{2}{4}, P_{svo} = \frac{3}{4}$. Record 1 is inserted at time 0 and root node **A** is created. Record 2 is inserted at time 4, and record 3 at time 5. Subsequently, record 3 is deleted at time 12 and record 1 at time 13. At this point node **A** contains two dead entries (1 and 3) and one alive entry (2). At time 15 record 4 is inserted and thus node **A** becomes full, with two alive and two dead entries. The next update occurs at time 20 when record 5 is inserted. Since node **A** is already full an overflow occurs which is handled by a version split. The alive entries (2 and 4) are copied into the new node **B**. All of node **A** entries are updated to reflect deletion time equal to 20 (an operation that can be avoided in practice, since the actual deletion time of an entry can be deduced from the deletion time of its parent). Node **A** is now considered dead, spanning the time interval $[0, 20)$. Node **B** is incorporated into the tree, since no version underflow or overflow occurred. At time 21 record 6 is inserted into the structure. Node **B** becomes full, containing 4 alive entries (2, 4, 5 and 6). Record 7 is inserted at time 25. **B** overflows and a version split occurs. Since all records are alive at time 25, a strong version overflow occurs and node **B** is key-split into two new nodes **C** and **D**. Node **B** dies covering the time interval $[20, 25)$. Node **C** and **D** contain 3 and 2 alive records respectively, thus they can be incorporated into the structure. Finally, record 5 dies at time 26. It does not cause any structural updates, since the strong version underflow condition is met, while if record 6 or 7 had died nodes **C** and **D** would have to be merged.

3 Object Approximation Algorithms

This section presents various algorithms for deciding how to approximate the objects contained in a spatio-temporal archive in order to reduce the overall empty volume, improve indexing quality and thus enhance query performance. Henceforth, the optimality of an approximation is considered only in terms of total volume reduction. The problem can be broken up into two parts:

1. Finding optimal object approximations: Given a spatio-temporal object and an upper limit on the number of MBRs that can be used to approximate it, find the set of consecutive MBRs that yield the representation with the minimum volume.
2. Reducing the overall volume of a dataset given a space constraint: Given a set of objects and a space constraint (or, equivalently, a maximum number of MBRs) approximate each object with a number of

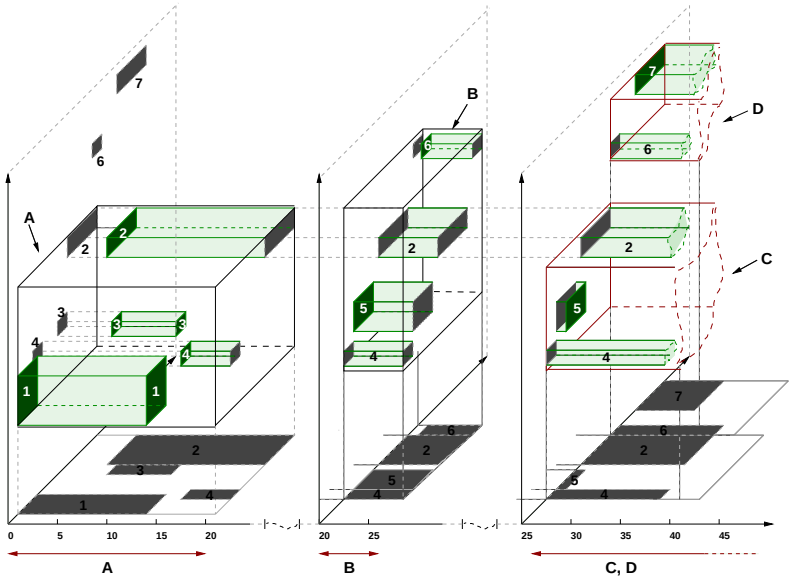


Fig. 4 A sample MVR-tree.

MBRs such that the overall volume of the dataset is reduced.

3.1 Finding Optimal Object Approximations

A simple method for approximating a spatio-temporal object using MBRs is to consider only the time instants when its movement/shape is updated (e.g., the functional boundaries) and partition the object along these points. Although, a few judiciously chosen MBRs are considerably more efficient in decreasing empty volume as shown in Figure 5 with an example. Another way would be to let the MVR-tree split the object trajectories at the time instants when a leaf version split occurs. The resulting MBRs could be tightened directly after the split by looking at the raw trajectory data. Nevertheless, this technique would not yield the optimal per object approximations, neither can it be used to approximate some objects bettern than others. In addition, it has the added cost of reading the raw trajectory data multiple times during index creation. Therefore, more sophisticated approaches for finding optimal object approximations are essential.

Given a continuous time domain there are infinite number of points that can be considered as MBR boundaries. Although, practically, for most applications a minimum time granularity can be determined. Under this assumption, an appropriate granularity can be selected according to various object characteristics. The more agile an object is (i.e., the more complex the object movement is), the more detailed the time granularity that needs to be considered. On the other hand, objects that evolve very slowly can be accurately approximated using very few MBRs and thus a very coarse granularity in this case is sufficient. Of course, the more detailed the granularity,

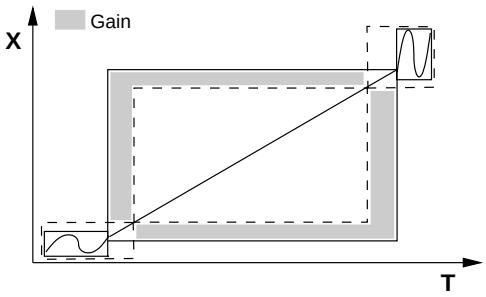


Fig. 5 Approximating the object using the three dashed MBRs yields more reduction in empty volume than the piecewise approach.

the more expensive the algorithms become. A good compromise between computational cost and approximation accuracy per object is application dependent. For ease of exposition in the rest it is assumed that a time granularity has already been decided for a given object. Two algorithms are presented that can be used to find the object approximation that minimizes the empty volume.

*3.1.1 An Optimal Algorithm (*DynamicSplit*)* Given a spatio-temporal object O evolving during the interval $[t_0, t_n)$ that consists of n time instants (implied by the chosen time granularity) the goal is to find how to optimally partition the object using k MBRs, such that the final volume of the approximation is minimized. Let $V_l[t_i, t_j]$ be the volume of the representation corresponding to the part of the spatio-temporal object between time instants t_i and t_j after using l optimal MBRs. Then, the following holds:

$$V_l[t_0, t_i] = \min_{0 \leq j < i} \{V_{l-1}[t_0, t_j] + V_1[t_j, t_i]\}$$

The formula is derived as follows: Suppose that the optimal solutions for partitioning the sub-intervals $[t_0, t_1)$,

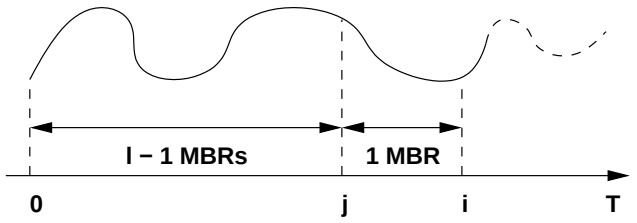


Fig. 6 Iteratively finding the best representation for interval $[t_0, t_i)$ using l MBRs.

$[t_0, t_2), \ldots, [t_0, t_{i-1})$ of the object using $l-1$ MBRs are already known. The goal is to find the optimal solution for partitioning interval $[t_0, t_i)$, using l MBRs. The algorithm sets the $(l-1)$-th MBR boundary on all possible positions $j \in [0, 1, \ldots, i-1]$, dividing the object movement into two parts: The optimal representation for interval $[t_0, t_j)$ using $l-1$ MBRs, which is already known by hypothesis, and interval $[t_j, t_i)$ using only one MBR (Figure 6). The best solution overall is selected, which is the optimal partition of $[t_0, t_i)$ using l MBRs. These steps are applied iteratively until the required amount of MBRs k for the whole lifetime of the object $[t_0, t_n)$ is reached.

Using the above formula a dynamic programming algorithm that computes the optimal positions for k MBRs is obtained. In addition, the total volume of this approximation is computed (value $V_k[t_0, t_n]$).

Theorem 1 *Finding the optimal approximation of an object using k MBRs (i.e., the one that minimizes the total volume) can be achieved in $O(n^2k)$ time, where n is the number of discrete time instants in the object's lifetime.*

Proof A total of nk values of the array $V_l[t_0, t_i]$ $(1 \leq l \leq k, 0 \leq i \leq n)$ have to be computed. Each value in the array is the minimum of at most n values, computed using the above formula. The volume $V_1[j, i]$ of the object between positions j and i can be precomputed for every run i and all values of j using $O(n)$ space and $O(n)$ time and thus does not affect the time complexity of the algorithm.

*3.1.2 An Approximate Algorithm (*GreedySplit*)* The DynamicSplit algorithm is quadratic to the number of discrete time instants in the lifetime of the object. For objects that live for long time-intervals and for very detailed time granularities the above algorithm is not very efficient. A faster algorithm is based on a greedy strategy. The idea is to start with n consecutive MBRs (the exact representation with the given time granularity) and merge the MBRs in a greedy fashion (Algorithm 1). The running time of the algorithm is $O(n \lg n)$, due to the logarithmic overhead for updating the priority queue at step 2 of the algorithm. This algorithm gives, in general, sub-optimal solutions.

Algorithm 1 GreedySplit

Input: A spatio-temporal object O as a sequence of n consecutive MBRs, one for each time-instant of the object's lifetime.

Output: A set of k MBRs that represent O's movement.

1: For $0 \leq i < n$ compute the volume of the resulting MBR after merging O_i with O_{i+1}. Store the results in a priority queue.

2: Repeat $n-k$ 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.

3.2 Reducing the Overall Volume of a Dataset Given a Space Constraint

It is apparent that given a set of objects, in order to represent all of them accurately, some objects may require only few MBRs while others might need a much larger number. Thus, it makes sense to use varying numbers of MBRs per object, according to individual object evolution characteristics. This section discusses methods for approximating a set of N spatio-temporal objects using a given total number of MBRs K such that the total volume of the final object approximations is minimized. [1] In the following, we refer to this procedure as the *MBR assignment process*, implying that, given a set of K "non-materialized" MBRs, each MBR is assigned to a specific object iteratively, such that the volume of the object after being approximated with the extra MBR is minimized (assuming that all objects are initially assigned only one MBR).

*3.2.1 An Optimal Algorithm (*DynamicAssign*)* Assuming that the objects are ordered from 1 to N, let $MTV_l[i]$ be the Minimum Total Volume consumed by the first i objects with l optimally assigned MBRs and $V_k[i]$ be the total volume for approximating the i-th object using k MBRs. The following observation holds:

$$MTV_l[i] = \min_{0 \leq k \leq l} \{MTV_{l-k}[i-1] + V_k[i]\}$$

Intuitively, the formula states that if it is known how to optimally assign up to $l-1$ MBRs to $i-1$ objects, it can be decided how to assign up to l MBRs to i objects by considering all possible combinations of assigning one extra MBR between the $i-1$ objects and the new object. The idea is similar to the one explained for the DynamicSplit algorithm. A dynamic programming algorithm can be implemented with running time complexity $O(NK^2)$. To compute the optimal solution the algorithm needs to know the optimal approximations per

[1] Where K is implied by some space constraint, e.g., the available disk space.

object, which can be computed using the dynamic programming algorithm presented in Section 3.1.1. Hence, the following theorem holds:

Theorem 2 *Optimally assigning K MBRs among N objects takes $O(NK^2)$ time.*

Proof A total of NK values for array $MTV_l[i]$ $(0 \leq l \leq K, 1 \leq i \leq N)$ need to be computed, where each value is the minimum of at most $K+1$ values for $0 \leq k \leq K$, in each iteration.

*3.2.2 A Greedy Algorithm (*GreedyAssign*)* The DynamicAssign algorithm is quadratic to the number of MBRs, which makes it impractical for large K. For a faster, approximate solution a greedy strategy can be applied: Given the MBR assignments so far, find the object that if approximated using one extra MBR will yield the maximum possible global volume reduction. Then, assign the MBR to that object and continue iteratively until all MBRs have been assigned. The algorithm is shown in Algorithm 2. The complexity of step 2 of the algorithm is $O(K \lg N)$ (the cost of inserting an object K times in the priority queue) thus the complexity of the algorithm itself is $O(K \lg N)$ (since it is expected that $N < K$).

Algorithm 2 GreedyAssign

Input: A set of N spatio-temporal objects and a number K.

Output: A near optimal minimum volume required to approximate all objects with K MBRs.

1: Assume that each object is represented initially using a single MBR. Find what the volume reduction $VR_{i,2}$ per object i would be when using 2 MBRs to approximate it. Store in a max priority queue according to $VR_{i,2}$.

2: For K iterations: Remove the top element i of the queue, with volume reduction $VR_{i,k}$. Assign the extra k-th MBR to i. Calculate the reduction $VR_{i,k+1}$ if one more MBR is used for i and reinsert it in the queue.

*3.2.3 An Improved Greedy Algorithm (*LAGreedyAssign*)* The result of the GreedyAssign algorithm will not be optimal in the general case. One reason is the following: Consider an object that yields a small empty volume reduction when approximated using only two MBRs, while most of its empty volume is removed when three MBRs are used (an 1-dimensional example of such an object is shown in Figure 7). Using the GreedyAssign algorithm it is probable that this object will not be given the chance to be assigned any MBRs at all, because its first MBR allocation produces poor results and other objects will be selected instead. However, if the algorithm is allowed to consider more than one assigned MBRs per object per step, the probability of assigning more MBRs to this object increases. This observation gives the intuition on how the greedy strategy can be improved. During each iteration, instead of choosing the object that yields the largest volume reduction by assigning only one more MBR, the algorithm can *look ahead* and find the object that results in even bigger reduction if two, three, or more MBRs were assigned all at once.

For example, the look-ahead-2 algorithm works as follows (Algorithm 3): First, all MBRs are assigned using the GreedyAssign algorithm as before. Then, one new priority queue PQ_1 is created which sorts the objects by the volume reduction offered by their last assigned MBR (if an object has been assigned k MBRs, the object is sorted according to the volume reduction yielded by the k-th MBR). The top of the queue is the *minimum reduction*. A second priority queue PQ_2 is also needed which sorts each object by the volume that would be reduced if it was assigned two more MBRs (if an object has been assigned k MBRs, the object is sorted according to the volume reduction produced by $k+2$ MBRs). The top of the queue is the *maximum reduction*. If the volume reduction of the top element of PQ_2 is bigger than the sum of the reduction of the two top elements of PQ_1 combined, the splits are reassigned accordingly and the queues are updated. The same procedure continues until there are no more redistributions of MBRs. In essence, the algorithm tries to find two objects for which the combined reduction from their last assigned MBRs is less than the reduction obtained if a different, third object, is assigned two extra MBRs. The algorithm has the same worst case complexity as the greedy approach. Experimental results show that it achieves much better results for the small time penalty it entails.

Algorithm 3 LAGreedyAssign

Input: A set of spatio-temporal objects with cardinality N and a number K.

Output: A near optimal minimum volume required to approximate all objects with K MBRs.

1: Allocate MBRs by calling the GreedyAssign algorithm. PQ_1 is a min priority queue that sorts objects according to the reduction given by their last assigned MBR. PQ_2 is a max priority queue that sorts objects according to the reduction given if two extra MBRs are used per object.

2: Remove the top two elements from PQ_1, let O_1, O_2. Remove the top element from PQ_2, let O_3. Ensure that $O_1 \neq O_2 \neq O_3$, otherwise remove more objects from PQ_1. If the volume reduction for O_3 is larger than the combined reduction for O_1 and O_2, redistribute the MBRs and update the priority queues.

3: Repeat last step until there are no more redistributions of MBRs.

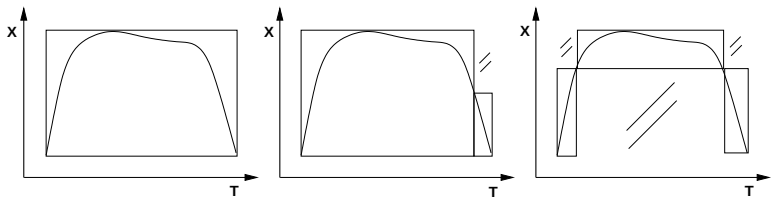


Fig. 7 Two MBRs yield almost no reduction in empty space, while three MBRs reduce it substantially.

4 Heuristics for Improving Performance

Methods to improve the running time of the previous algorithms and heuristics to choose the right value for the number of total MBRs (K) are discussed next. Furthermore, we discuss how to improve the query performance by using clustered index structures where the actual data are stored in the data nodes of the index.

4.1 Reducing the Computational Cost

Intuitively, the most beneficial MBR allocations are the ones that involve objects that consume considerable empty volume. These objects can be determined in advance by taking into account various characteristics of the object evolutions. Logically, by applying the K MBRs only on the subset of the objects that consume most of the empty volume, it is expected that the query performance of the index will improve, while the computational cost of the DynamicAssign and GreedyAssign algorithms will be reduced significantly. In particular, by examining each object's movement it can be decided in advance if it should be included in the MBR assignment process or not.

The following heuristics are used:

1. The lifetime of an object. Short lived objects do not introduce a lot of empty volume. Their MBRs are short on the time dimension.
2. The total volume of the object. Objects with small volume do not offer any opportunities for volume reduction and vice versa. Choosing an appropriate volume threshold for a given dataset depends on the dataset characteristics, in which case sampling techniques can be used to determine an approximate value for the average volume of the objects.
3. The object velocity. Objects that evolve rapidly consume increased empty volume per time instant (even if they are short lived). On the other hand, objects that are static do not introduce any empty volume.
4. Direction changes. Objects whose evolution is described by a large number of functions offer more possibilities for empty volume reduction (since they have large curvature, many peaks, etc.).

A simple algorithm that uses these heuristics to identify "interesting" objects is shown in Algorithm 4.

A related optimization for reducing the computational cost can be applied when specific *user query patterns* are known in advance. For example, if it is known that most queries refer to specific time-intervals of the dataset evolution, the structures can be optimized for these intervals only. Objects that have lifetimes that do not intersect with the time-intervals of interest can be ignored. Equivalently, assigning the K MBRs can be aimed only towards the intervals of interest.

Algorithm 4 Identifying objects with large empty volume

Input: A set of N object trajectories.

Output: A reduced set of objects which contribute the most in empty volume.

1: Produce a random sample $\mathcal{S}$ using reservoir sampling.

2: From $\mathcal{S}$ compute the average volume $\bar{V}$ of the objects' single MBR approximations, the average object lifetime $\bar{L}$ and the average number of movement functions $\bar{F}$ per object.

3: For each dataset object O **skip** the object if:

1. $V_O < \tau_1 \bar{V}$ or
2. $V_O < \tau_2 V_U$ or
3. $L_O < \tau_3 \bar{L}$ or
4. $F_O < \tau_4 \bar{F}$

(where $\tau_1, \tau_2, \tau_3, \tau_4 \in [0, 1]$ are application dependent thresholds, V_U the total universe volume, V_O is the object's volume, L_O its lifetime and F_O the total number of functions describing its movement).

4.2 Reducing the Space Overhead

The algorithms discussed in the previous section take as input the total number of MBRs and generate new object representations with reduced volume. Thus, an important decision before building an index is choosing the number of MBRs that will provide a good trade-off between query performance improvement and space overhead. A good value for this parameter will affect the performance of the index structure substantially. A small number of MBRs will result in poor query performance, while a large number of MBRs might not be necessary in

all cases. This section gives an overview of two methods for determining a sensible number of MBRs.

The first approach uses analytical models to predict the performance of the index. In theory, an analytical model for a specific index structure tries to predict the average number of accesses that would be needed to answer a query from a random distribution without resorting to building the index first. This is accomplished by using various dataset characteristics, like the total number of objects, the object distribution, etc. For the problem at hand, after deciding which index structure will be used and, given a number of MBRs K, the MBRs are assigned to the objects and essential statistics about the resulting object approximations are estimated. Then, the analytical model can be applied to get a prediction on the expected query performance. This process can be repeated for a number of different Ks, until a satisfactory prediction is produced by the model. Finally, the actual index can be built using the best K overall.

Accurate analytical models for multi-version structures appeared in the literature recently [51]. Generally, these models require uniform data distributions and uniform data extents (i.e., where the MBR sizes are uniformly distributed on all dimensions). Unfortunately, for trajectory archives, even if the objects are uniformly distributed in space and moving in a uniform fashion, no guarantees can be made about the uniformity of the resulting MBR approximations, after applying the proposed algorithms. In general, these MBRs can have arbitrary extents on all dimensions, making the use of analytical models problematic. Recently, Tao and Papadias [49] proposed methods for performance analysis of R-trees with arbitrary node extends. Although, it is not clear if such techniques can be modified easily for the multi-version setting.

This observation leads to a different approach, more appropriate for spatio-temporal trajectories. A better heuristic for estimating a good value for K is the reduction in empty volume produced. Given a set $\mathcal{S} = \{K_1, \ldots, K_m : K_i < K_j, 1 \leq i < j \leq m\}$ of increasing numbers of K, the dataset is approximated using all Ks and the total reduction R in empty volume is computed in each case, where, intuitively, $R(K_{i+1}) = MTV_{K_i}[N] - MTV_{K_{i+1}}[N]$. If $R(K_i) \approx R(K_{i+1})$ for some i, there is no reduction in empty volume from one iteration to the next, thus the maximum number of MBRs that will definitely be beneficial in terms of query performance has been established; using more than K_i MBRs is unnecessary since the objects are approximated as well as they can be. In addition, based on the observations of Section 4.1, the algorithm can assign all K MBRs on a small fraction of the dataset, making the process more efficient.

The proposed algorithms are based on the fact that the improvement in query performance is proportional to the reduction in empty volume produced by finer approximations. Since increasing the number of MBRs reduces the total empty volume, the *reduction* R in empty volume as a function of the number of MBRs K, $R(K) : \mathbb{N} \rightarrow \mathbb{R}$ is a monotonically increasing function. However, this function has a tight upper bound (after all objects have been approximated exactly, i.e., with one MBR per time-instant of their lifetime, no more reduction in empty volume can be achieved). In addition, since MBRs are optimally assigned (or almost optimally when using the GreedyAssign algorithms) this function follows the law of diminishing returns, eventually becoming flat after a number of MBRs has been used. Hence:

Claim For any given spatio-temporal dataset one can always find the point of diminishing returns of the empty volume reduction function $R(K)$.

Beyond that point, more MBRs are no longer beneficial both in terms of empty volume and, as a consequence, query performance. This conjecture is verified in the experimental evaluation.

4.3 Improving Query Performance with the Use of Clustered Index Structures

For most indexing methods there are two factors that contribute to the total number of disk I/Os that need to be performed in order to answer a given query. First, the structure has to be traversed in order to produce a number of candidate trajectories. Then, the candidate trajectories need to be loaded from storage in order to verify the results. The first component is termed the index I/O, while the second is the verification I/O. An efficient index structure needs to minimize the access cost to the structure as well as the number of candidates that need to be retrieved. For non-clustered indices and range queries we know that trajectories retrieved by the index step that have at least one MBR that is completely contained in the query region are definite answers. Therefore, in this case we only need to consider all other candidate trajectories during the verification step. For neareast neighbor queries, all candidate trajectories need to be retrieved from disk, contributing one random I/O per trajectory. Notice that, if the index stores very fine object approximations then the number of candidates that need to be retrieved is minimized and, hence, the verification step becomes more efficient. In contrast, if the index stores very rough object approximations, then it is expected that a given query will intersect with a larger percentage of object approximations, yielding a higher verification cost. Intuitively, for non-clustered index structures storing finer object approximations is essential for minimizing the number of random I/Os that will eventually have to be performed.

Another alternative is to consider clustered index structures where the data associated with every entry of a leaf is stored sequentially on disk along with that leaf. This enables loading the necessary data with an

access cost at least twenty times smaller than a random I/O. In most practical scenarios it is expected that the data associated with a leaf will occupy only a small number of pages, thus the amortized cost per leaf verification will be equivalent to the cost of only one random access. Therefore, for the clustered index case it is expected that the verification cost will be subsumed by the index cost. In the given scenario, since the trajectory archive is available beforehand, clustered indices can be constructed without difficulty. First, a non-clustered index is created and then the structure is reorganized in order to place sequentially on disk all data pages corresponding to every leaf. Alternatively, for the R-tree index, bulk loading techniques can be used [29,22].

It should be stressed here that there exist two cases where clustered index structures are not necessary in practice. The first case is if the objects follow piecewise *linear* movements and are approximated with a number of MBRs such that each MBR encloses exactly one line segment. In this case since every MBR essentially represents a line segment, at the leaf level of the tree every node can contain the actual line segment representations, such that the need to store the actual data on separate storage is obviated. (Note here, that this article considers the more general case where objects can follow arbitrary trajectories that are not approximated by straight lines.) The second case is when the index stores the exact representations of the objects. In that case every leaf contains the actual locations of the object movements per time-instant of the objects lifetime. In both cases the proposed algorithms combined with a clustered index organization will still produce better results due to improved index quality, since for the first case the index contains excessive empty volume, while for the second case the index has excessive size, two drawbacks that are mitigated by wisely selected object approximations.

5 Online Processing and Storage of Spatio-Temporal Archives

All algorithms described so far are suited for off-line query performance optimization. This means that the spatio-temporal archive is already available and the techniques can apply global optimization strategies. Nevertheless, many spatio-temporal applications deal with data that arrive as a stream of transactions in real-time. This section gives an overview of some techniques that can be used in combination with the algorithms proposed previously for archiving and indexing incoming data in an online fashion, while trying to maintain improved query performance.

Imagine a sensor based system where moving agents send alerts to a central server with location and motion information (e.g., using a GPS device and a cellular network). The server collects incoming messages, processes the information and finally inserts the new data in a spatio-temporal index [59,60,7]. Storing the history of the movement of every object in this case can be accomplished using a bounding box based index like the R-tree or a multi-version structure, as long as object insertions arrive in strict temporal ordering. Since corresponding buffering policies for the MVR-tree become more involved, for ease of exposition this discussion considers only the R-tree case.

Lets assume that objects follow linear trajectories between consecutive updates or that the object movement characteristics are send along with the location information so that the bounding box of the trajectory inbetween the updates can be decided. A naive approach for indexing the data on the fly would be to store incrementally all MBRs between consecutive updates for every object. For example, a vehicle issuing one location update per minute can be stored as a sequence of consecutive MBRs, each MBR corresponding to the extent of the object's movement during each minute. Essentially, this approach corresponds to a piecewise storage scheme; no effort is made to decrease the volume and the space requirements of object representations. In environments where object updates are sparse this solution might work well. In most practical applications, though, that deal with thousands of objects and very frequent updates, this simplistic approach will result in numerous expensive index updates that will render the system unusable. Also, multiple object copies (essentially every update creates a copy) will deteriorate index quality substantially.

Alternatively, a better strategy would be to buffer a large number of updates such that, by combining consecutive object movement functions into a single MBR and by performing a batch update for all objects, the number of index operations is reduced. A batch update can occur whenever the buffer is full, or after a maximum object update limit has been attained. Furthermore, it is also possible to assign a number of MBRs judiciously between buffered objects, if necessary, in order to decrease the overall empty volume as much as possible, before the batch update is performed. This operation can be thought of as the reverse of decomposing an object into multiple MBRs given the single MBR representation, as has been assumed throughout the rest of this article. In this case, given a *partial piecewise* representation of an object, the goal is to first combine as many pieces as possible into a single MBR (which depends on the average object update rate and the available buffer size) and then determine an appropriate number of MBRs such that the total volume of the representation is reduced substantially with respect to the initial piecewise representation, while the total number of MBRs remains smaller or close to the initial number of updates (i.e., to the number of piecewise movements before grouping occurred).

A best effort approach can be used in this case. Assume, for instance, that a total of M objects have been buffered so far and U updates have occurred, where $U > M$ (i.e., some objects have issued multiple updates). The piecewise approach corresponds to U MBRs, one per object update. On the other hand, by using only one MBR per object for all consecutive updates, a total of M MBRs are obtained. Next, a total of $K = U$ MBRs (or a number K proportional to U) can be assigned to the M objects in expectation that the total volume of the representation will be reduced, when compared with the piecewise approach. The number K can be determined by maintaining essential statistics about the dataset that has been seen so far, like the average object volume, the average lifetime, etc. By estimating such properties for the buffered objects, it can be decided if it is worth assigning a large or a small number of MBRs for a specific group of buffered updates. Available disk space and the disk space consumed so far should also play a role.

An important aspect of this technique is the number of updates that have occurred per buffer overflow. In the worst case, if every object has issued only one update, then during the MBR assignment process a number $K > U$ of MBRs will need to be assigned in order to decrease the volume of the approximations with respect to the piecewise approach (if $K = U$ the assignment process will yield one MBR per object which is equivalent to the piecewise approximations), resulting in a representation that increases the number of the indexed MBRs beyond that of the piecewise approach. Efficiency in this case will heavily depend on the total volume reduction attained. In such cases, it might be sensible to skip the assignment process and flush the objects directly to the index. If, on the other hand, a large number of updates corresponds to every object, then the MBR assignment algorithms are expected to produce considerably better representations with a fewer number of MBRs (for $K < U$), when compared with the piecewise approximations. Intuitively, the number of updates per object depends both on application characteristics (average update rate, locality of object updates, etc.) and on the total available buffer size.

One extreme scenario occurs in situations where objects issue very frequent updates, e.g., approximately every time-instant. Clearly, in this case the objects are already represented using very fine approximations. Any grouping strategy that tries to re-assign MBRs in order to decrease the overall volume of an object's representation is bound to produce minimal gain, if any. Nevertheless, in such cases it is also obvious that index update and query performance will greatly improve simply by grouping a very large number of updates, and decreasing the overall number of insertions and MBRs that have to be indexed. In this case, the total volume of the indexed approximations increases with respect to the piecewise approach, meaning that the number of false hits also increases during query execution, but maintaining a clustered structure in this scenario is straightforward (because historical data will never need to be modified and, thus, can be safely stored sequentially on disk during a batch update whenever a leaf becomes full, for a small additional cost). Hence, it is expected that the overall query performance will improve considerably.

Another extreme scenario occurs when object issue extremely sparse updates, in which case a technique that can be used to improve the update/object ratio is to pin objects with too few updates into the buffer and apply the algorithms only to the objects that have already issued an adequate number of updates. Eventually, the pinned objects will have issued a sufficient number of updates, in which case they will be removed from the buffer and included in the MBR assignment process. In addition, objects that have been pinned in the buffer for a long time should also be considered for MBR assignment, since their movement will already encompass a large number of time-instants, yielding ample approximation opportunities (long lived objects contribute to increased empty volume). Straightforwardly, pinning objects into the buffer results in more frequent buffer overflows, and thus more frequent MBR assignment operations and tree updates. In order to alleviate this problem, if the number of buffer overflows or the total size of pinned entries exceed a prespecified threshold, all entries can be flushed to the tree, purging the buffer completely. Using buffering techniques with the MVR-tree poses additional difficulties, since it is required that entries are inserted in the tree in chronological order. For that reason, pinning objects into the buffer is not possible when multi-version structures are considered.

By using the proposed techniques it is assured that update throughput and query performance will improve compared to the simple piecewise insertions, since it can be guaranteed that the total volume of the trajectory representations will be reduced while the number of MBRs will not grow larger. The experimental evaluation corroborates this intuition.

6 Experimental Evaluation

In order to evaluate the techniques proposed in this article three synthetic datasets were used to run a number of simulations. Each dataset represented a set of moving objects following various trajectories on a 2-dimensional universe. The simulations were run for 100 time instants in total, resulting in three trajectory archives. The first simulation represented a network of freeways and highways (the states of Illinois and Indiana) where every road is a collection of connected line segments. The universe was clipped to 500 miles in each direction and the objects followed random routes on the network, with randomly generated velocities between 10 and 90 mph selected using a skewed distribution. This dataset is referred to as FREEWAY (Figure 8(a)). Two archives

Table 1 Dataset Statistics.

	FREEWAY	OLDENBURG	ATTRACTOR
Total objects	400K/100K	150K	400K/100K
Avg. objects per time-instant	400K/100K	83K	400K/100K
Avg. object lifetime	100	55	100
Total movement functions	9,548,302/2,039,329	8,312,267	1,283,464/214,406
Average movement functions per object	23.8/20.3	55.4	3.2/2.1

were used, containing 100,000 and 400,000 trajectories each. For the second simulation a trajectory archive was gathered by using the road network generator introduced by Brinkhoff [6] on the city of Oldenburg with 10 object classes, 5 external object classes, 10,000 initial objects and 1,000 new objects per time-instant. This archive consists of 150,000 trajectories in total and is referred to as OLDENBURG (Figure 8(b)). Finally, for the last simulation objects were initially positioned using a skewed distribution and let to randomly select one out of five fixed destinations with a preset probability. The destinations "attracted" and eventually "repulsed" the objects, which had to choose a new destination to go to. The objects followed movements that represent combinations of polynomial functions of 1st and 2nd degrees. Objects were forced to update their movement every 30 time-instants by changing their velocity or the coefficients of their movement, but making sure that they always headed towards the same destination. Object velocities were chosen at random using a uniform distribution. The purpose of this archive is to illustrate the efficiency of the proposed techniques for non-linear, random movements. This dataset is referred to as ATTRACTOR (Figure 8(c)). Table 1 summarizes various archive characteristics [2]. The final object trajectories were normalized in the unit square and stored as piecewise segments of polynomial functions along with the time-interval corresponding to every segment. The proposed algorithms were used to perform scaling and performance evaluation tests. All experiments were run on an Intel Pentium III 1GHz processor, with 512 Mbyte of main memory.

In the rest of this section, in order to have consistent scales in the graphs, the number of allocated MBRs K is expressed as a percentage of the total number of objects N contained in the dataset. More specifically, when referring to $S\%$ MBRs, $K = N + \frac{S}{100}N$ MBRs are used in total to approximate the dataset. Intuitively, this parameter cannot be varied for the piecewise approaches. Although, in order to make comparisons easier between various techniques, the piecewise approaches appear in the same graphs, but the horizontal axes do not apply for them.

[2] The generators can be downloaded from [18].

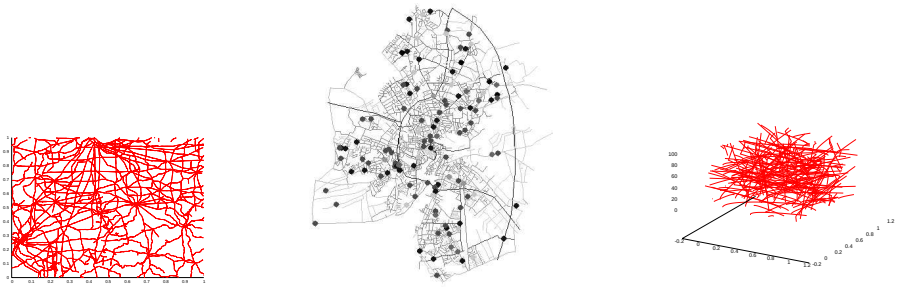
6.1 Evaluation of DynamicSplit *and* GreedySplit *Algorithms*

An experimental evaluation of the DynamicSplit and GreedySplit algorithms in terms of computational cost as well as efficiency (reduction in empty volume) was conducted. For simplicity, a fixed time granularity of 1 time instant was set for all objects, which is the finest granularity available for the generated datasets. Varying the granularity to coarser levels would have a positive effect on the cost of the approximation algorithms, while the effect on index performance would be similar to the effect of reducing the approximation accuracy by reducing the total number of allocated MBRs. Thus, coarser time granularity levels are not considered in this experiment.

The efficiency of the algorithms was evaluated by assigning from 10% to 300% MBRs on all datasets. A total of K MBRs were assigned to the objects using the DynamicAssign algorithm twice: The first time using the DynamicSplit and the second time the GreedySplit algorithm for approximating all objects before the assignment process. Figure 9(a) plots the computational cost required by the two algorithms in order to approximate all objects, when using the 100K FREEWAY dataset. Figure 9(b) shows the efficiency of the algorithms in terms of reducing the total empty volume, for the same dataset. More specifically, the graph plots the total volume reduction of the approximated objects as a percentage of the total volume of the original objects. It can be seen from these graphs that even though GreedySplit is much faster than DynamicSplit (111 times faster), the reduction in empty volume is very close to the optimal for any given K. The same behaviour was observed for the rest of the datasets. For ease of exposition, the results for the experimental evaluation using the GreedySplit algorithm will be presented only.

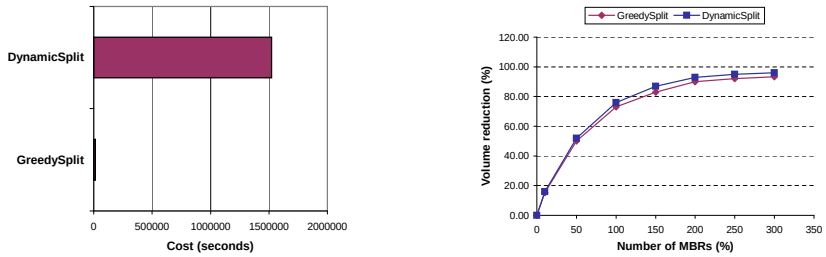
6.2 Evaluation of MBR Assignment Algorithms

Various properties of the DynamicAssign, GreedyAssign and LAGreedyAssign algorithms were evaluated. In particular, the scaling behaviour of the algorithms



(a) FREEWAY (b) OLDENBURG (c) ATTRACTOR

Fig. 8 Datasets.



(a) Computational cost (b) Efficiency

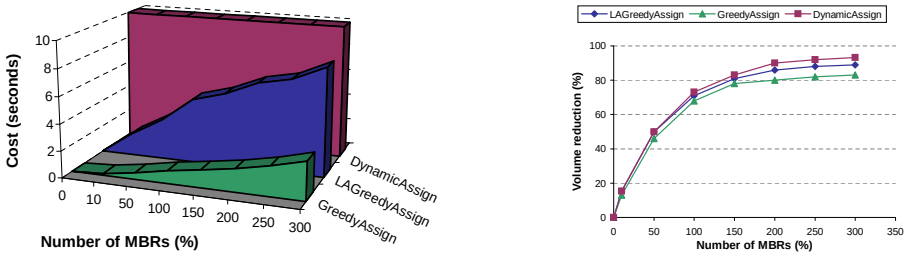
Fig. 9 Comparison of approximation algorithms using a FREEWAY dataset with 100,000 objects.

was observed using increasing numbers of objects and approximation accuracy. The approximation and assignment algorithms require one full database scan in order to compute the best approximation per object. Since the overhead of this step occurs only once before building the index, it can be considered as an off-line amortized cost, and for that reason it is not considered further in the following experiments.

First, 10% up to 300% MBRs were assigned on the 100K FREEWAY dataset using the three algorithms. Then, the computational cost of every approach was measured. Figure 10(a) plots the corresponding results. The two greedy approaches were very efficient, with LAGreedyAssign being slightly slower than GreedyAssign. DynamicAssign was 126 times slower than LAGreedyAssign for 300% MBRs (it is clipped in the graph in order to preserve detail). The efficiency of the algorithms in terms of the total volume reduction is shown in Figure 10(b). LAGreedyAssign was within at most 5% difference from the optimal algorithm in all cases. The GreedyAssign algorithm was within 10% difference for 300% MBRs. The small time penalty of LAGreedyAssign is outweighed by the total volume reduction, when compared with GreedyAssign. Similar remarks can be made for the other datasets. For the rest of the experiments only the results for the LAGreedyAssign algorithm are presented.

6.3 Performance Evaluation of Index Structures

All datasets were indexed using the 3-dimensional R*-tree and the MVR-tree approaches. Both indices were converted to clustered structures off-line, by storing the raw data associated with every leaf sequentially on disk. Moreover, for completeness a piecewise approximation technique was also considered where every trajectory was represented as a set of MBRs, one MBR per movement function of the trajectory (notice that a trajectory may consist of a single movement function for the whole lifetime of the object on one extreme, and of up to one movement function per time-instant of the object's lifetime on the other extreme; for the datasets at hand the average number of functions per object is shown in Table 1. Notice that the piecewise structures indexing objects that follow linear movements are essentially clustered structures. Although, for non-linear piecewise movements like the ATTRACTOR dataset, these structures have to be converted to clustered indices, as well. It is apparent that the piecewise approximation has space utilization that is dataset dependent. The resulting number of MBRs cannot be tuned, thus the space required for a piecewise index is predetermined. It is expected that the efficiency of this approach will highly depend on core dataset characteristics, like the average number of movement functions per object and the average volume consumed per movement func-



(a) Computational cost (b) Efficiency

Fig. 10 Comparison of MBR assignment algorithms using a FREEWAY dataset with 100,000 objects.

tion. On the other hand, the proposed algorithms provide the capability to tune the size of the structures according to given space constraints while, at the same time, guaranteeing good query performance irrespective of such dataset properties. It should be added here that indexing the exact object representations (one point per time-instant of an object's lifetime) was not considered since the sizes of such indices would be excessive for the larger datasets (e.g., 40,000,000 points vs. 1,200,000 MBRs), except only for the OLDENBURG dataset where the piecewise approximation coincides with the exact representation by construction of the dataset, since all objects report their position and movement function for every time-instant of their lifetime.

In order to evaluate the query performance of the structures, various period range query workloads containing 1,000 queries each were generated uniformly at random. The query area was fixed to 1% of the universe with small period queries corresponding to time ranges of 1 or 2 time instants, medium to 10 time instants and large to 30 time instants.

First, a series of increasing numbers of MBRs were assigned to all datasets using the LAGreedyAssign algorithm. Then, the resulting MBRs after each round were indexed using both structures. The piecewise structures were created as well. The page size was set to 4 Kbytes for all datasets and all indices. A 10 page LRU buffer was used which was reset before the execution of every query. In addition, for the MVR-tree the minimum alive records per node parameter was set 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, as suggested by Tao and Papadias [47]. The time dimension extent has no significance for the MVR-tree structures since the multi-version approach clusters the data independently on the time dimension.

Figures 11, 12 and 13 plot the results for the 400K FREEWAY, 150K OLDENBURG and 400K ATTRACTOR datasets for random workloads of small period range queries. A detailed exposition of the results for the FREEWAY dataset is given first. Then, the results for the rest of the datasets are presented. Similar results were obtained for the 100K datasets and, thus, the corresponding graphs are omitted.

Figure 11(a) plots the size of the FREEWAY structures given a wide range of object approximations, from 10% up to 300% MBRs (this graph includes only the actual size consumed by the index structures, excluding the data pages that contain the raw trajectories). For ease of exposition the sizes of the piecewise R-trees and MVR-trees are plotted in the same graph (and for subsequent graphs). In what follows, the piecewise approaches are referred to as pw-R-tree and pw-MVR-tree, while for the rest of the trees the terms R-tree and MVR-tree are used, respectively. It can be observed that pw-R-tree and pw-MVR-tree have at least 2.5 and 4.4 times larger sizes than an MVR-tree with judiciously chosen approximations. It is also clear that the MVR-tree is at most twice as large as a corresponding R-tree (i.e., an R-tree that indexes the same number of MBRs) in the worst case. The tree sizes of the piecewise approaches are large due to the large number of movement functions, as can be seen in Table 1. Indicatively, for this dataset 9,548,302 MBRs correspond to 2,387% MBRs being assigned using the proposed algorithms, in order to produce R-trees and MVR-trees of comparable sizes.

Figure 11(b) plots the average I/O cost for producing the exact answer to a small period query. As expected, increasing the approximation accuracy has a negative effect on the R-trees and a positive effect on the MVR-trees. In this case it is clear that the pw-MVR-tree performs the least amount of I/Os. Although, it is expected that by increasing the allocated number of MBRs, the MVR-tree will achieve similar performance. In order to understand which structure provides the largest benefits, it is important to take into account the query-efficiency vs. space tradeoff characteristics, all together. Figure 11(c) plots the improvement in query efficiency as a function of the space overhead imposed by the finer object approximations, when comparing the MVR-tree

with the best overall R-tree, which in this case is the one with 0% MBRs (i.e., the single MBR trajectory approximations). For consistency, the piecewise structures along with their space requirements are shown in the same graph (the straight lines and their corresponding space overhead labels). It can be deduced that the MVR-tree offers a 20% improvement in I/O for 9 times larger space, while the pw-MVR-tree offers 40% improvement for 36.5 times larger space. Following the trend one can forecast that the MVR-tree will achieve the same performance for 11 times the space.

The results for the OLDENBURG dataset are presented in Figure 12. Figure 12(a) plots the structure sizes. As already mentioned, by construction in this dataset the objects update their movement characteristics on every time-instant. Hence, the piecewise approaches correspond to one MBR per location per time-instant, yielding very large index sizes that are equivalent to structures indexing the exact object representations (i.e., one point per time-instant of an object's lifetime). On the other hand, the MVR-trees in this case are at most 3 times larger than the corresponding R-trees, for equal numbers of allocated MBRs.

Figure 12(b) plots the efficiency of the structures in terms of the average number of I/Os needed to answer a small period query. Once again, it can be deduced that the MVR-tree benefits from an increased number of judiciously assigned MBRs, in contrast with the R-tree. It is also apparent that the performance of the 300% MVR-tree is very close to that of the pw-MVR-tree. Figure 12(c) plots the improvement in efficiency as a function of space utilization, when compared with the 0% R-tree (which is the best R-tree alternative in this case). The added benefits of using the proposed algorithms are clearly illustrated in this graph. The MVR-tree can achieve substantial improvements in I/O cost (up to 63% fewer I/Os than the R-tree) for even a small increase in extra space. Moreover, it can achieve better performance than pw-MVR-tree for less than one twelfth of the space.

Finally, the results for the ATTRACTOR dataset are shown in Figure 13. As can be clearly seen in Figure 13(a), the sizes of the structures follow similar trends with the previous datasets. The MVR-trees are at most 2.5 times larger than the corresponding R-trees, for the same number of allocated MBRs. The piecewise indices are smaller in size in comparison with the previous datasets since, as can be seen in Table 1, the total number of functions for the ATTRACTOR dataset is small.

Figure 13(b) plots the average number of I/Os needed by each structure in order to answer small period queries. The MVR-tree performance is the best overall for this dataset, as well. Surprisingly, the performance of the R-tree improves with increasing number of MBRs, contrary to the observed behaviour in the previous datasets. Intuitively, a small increase in the number of indexed objects is outweighed by the total volume reduction offered by the finer approximations, in this case. Figure 13(c) plots the overall efficiency/space tradeoff of the structures, when compared with the 300% R-tree which is the best R-tree alternative for this dataset. Yet again, the MVR-tree enables to tune the performance of the structure according to the available space resources, by providing the best performance tradeoff overall.

6.4 Evaluation of Query Workloads With Larger Temporal Extents

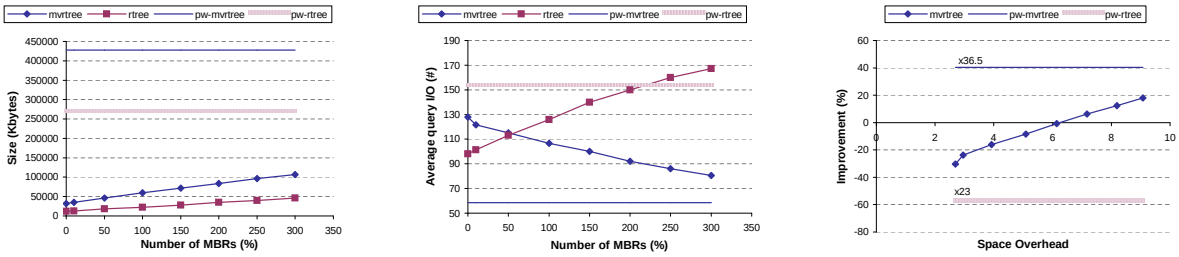
The proposed techniques were also tested using medium and large period query workloads. The results are shown in Figure 14 for all datasets. Even for medium period queries it can be claimed that the MVR-tree performance is comparable to that of the best R-tree. Especially for the OLDENBURG dataset the MVR-tree can still achieve up to 20% performance improvement making it the best choice overall. For the ATTRACTOR dataset the 300% MVR-tree achieves substantial benefit for small queries and comparable performance for medium queries in comparison with the best R-tree, making it a desirable candidate for a wide range of queries.

6.5 Evaluation of Computational Cost Reduction Heuristics

This section presents an evaluation of the heuristics proposed for reducing the computational cost associated with the MBR assignment algorithms. In this experiment a number of trajectories were excluded from the assignment process according to various dataset properties. Results are presented for the FREEWAY dataset. Similar behaviour was observed for the rest of the datasets. In this particular case an object was excluded from the assignment process if the volume of its single MBR approximation was less than the average single MBR object volume. The lifetime of the objects was not taken into account since for the FREEWAY dataset all objects have the same lifetime, equal to the total simulation length. This yielded a total of 50% of the objects being excluded. It can be inferred from Figure 15 that the heuristics help reduce the computational cost in half, while the biggest percentage of empty volume is removed.

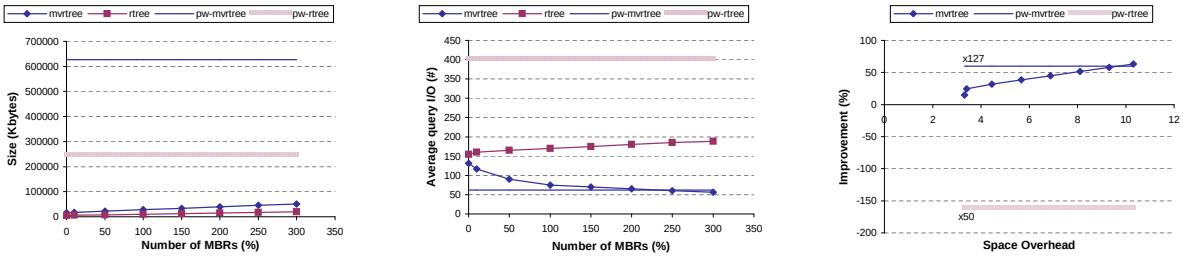
6.6 Identifying the Approximation Accuracy Curve

The total volume reduction yielded by the finer object approximations is a good indicator for identifying the ideal number of MBRs that need to be assigned to a given dataset in order to tune the MVR-tree performance. Figure 16 plots the total volume reduction as a percentage of the total volume of the single MBR object approximations, for various MBR numbers (depicted as Reduction and corresponding to the right axis on the



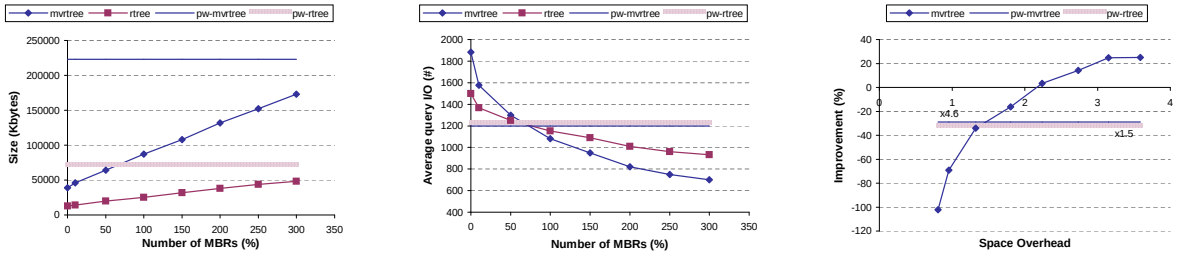
(a) Size (b) Average query I/O (c) Efficiency/Space tradeoff

Fig. 11 Experiments using a FREEWAY dataset with 400,000 objects.



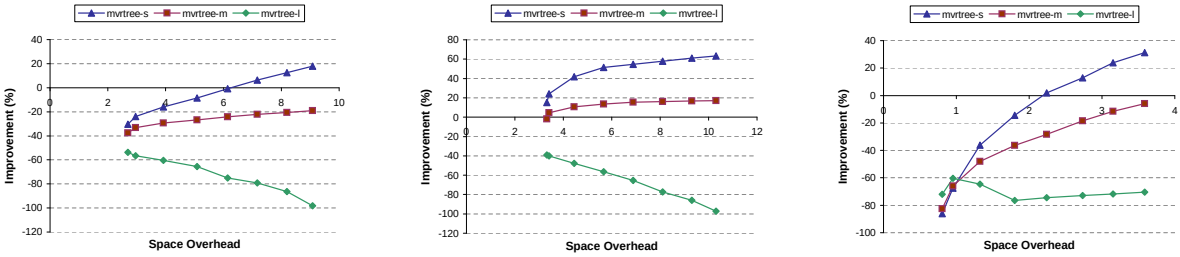
(a) Size (b) Average query I/O (c) Efficiency/Space tradeoff

Fig. 12 Experiments using an OLDENBURG dataset with 150,000 objects.



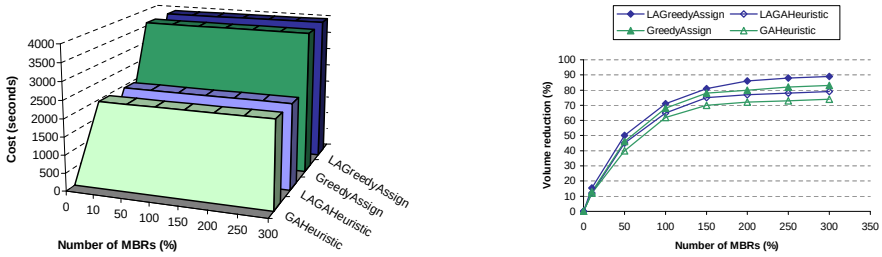
(a) Size (b) Average query I/O (c) Efficiency/Space tradeoff

Fig. 13 Experiments using an ATTRACTOR dataset with 400,000 objects.



(a) FREEWAY (b) OLDENBURG (c) ATTRACTOR

Fig. 14 Experiments using small, medium and large period queries for all datasets.



(a) Computational cost. (b) Efficiency.

Fig. 15 Evaluation of optimizations for reducing the computational cost of MBR assignment algorithms.

graphs). It can be observed from these plots that the volume reduction curve follows closely the query performance curve of the MVR-tree approach for all datasets, which validates the conjecture. If the volume reduction curve is known, then the query performance for an MVR-tree can be estimated, and thus an adequate number of MBRs can be decided in advance.

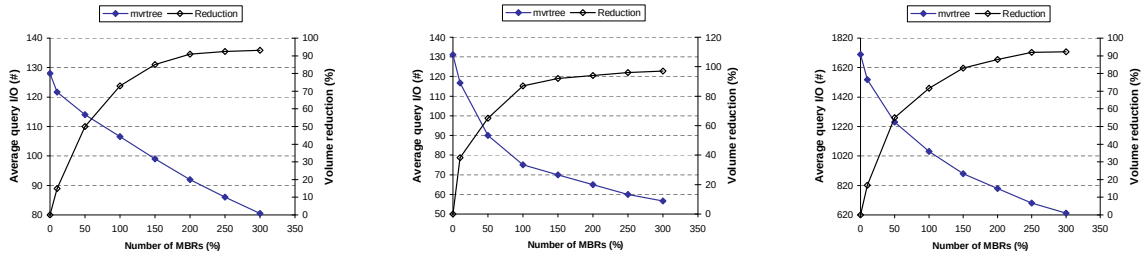
6.7 Evaluation of The Online Assignment Algorithm

The last experiment evaluated the efficiency of the online MBR assignment algorithm. Buffer sizes ranging between 16 Kbytes and 10 Mbyte were tested, and the following buffering policy was used: If the elapsed time between two consecutive updates was less than 10 time-instants, the object was pinned in the buffer. Pinned objects remained in the buffer in-between buffer overflows. The intuition behind this is to allow objects to span an adequate amount of time-instants before being inserted into the index. Object updates corresponding to more than 10 time-instants were included in an MBR assignment process as soon as the buffer became full, in order to decrease the volume of their representation before inserting them into the index. The total number of MBRs assigned to these objects was equal to 5% of the total time-instants spanned by the objects. Essentially, this implies that every 100 time-instants of an object's lifetime, the object would be assigned a total of 5 MBRs, on average. If the pinned objects occupied more than 50% of buffer space, then an assignment process between all objects was performed and the buffer was purged.

Results for the ATTRACTOR dataset are shown in Figure 17. The performance of the piecewise approach is also depicted in the graphs. The piecewise approach corresponds to the simplest policy, where object updates are directly inserted in the index as they arrive, without trying to re-assign any MBRs. Figure 17(a) plots the final size of the structures. In general, the number of MBRs allocated during each assignment process can be tuned in real-time in order to keep the size of the index (i.e., the total number of MBRs) smaller or close to the piecewise approach. In this experiment we aimed at achieving an almost fixed index size. Figure 17(b) depicts the index I/O performance using various buffer sizes. It is clear that the structures using intelligent MBR allocations outperform the piecewise approach, yielding 14% fewer query I/Os in the best case. It is also clear that 512 Kbytes buffer size is adequate in this case to achieve the best performance. Although, this structure has slightly higher space requirements than the structure corresponding to the 1 Mbyte buffer. Larger buffer sizes do not improve MBR allocations substantially, meaning that 512 Kbytes is adequate given the objects/update ratio in this case. Finally, Figure 17(c) plots the total volume reduction achieved using various buffer sizes, when compared with the total volume of the piecewise approach. All buffer sizes achieve approximately close to 40% volume reduction. In the same graph, the total number of MBRs allocated in each case are also shown. Clearly, by using a larger buffer the same volume reduction is achieved by assigning a smaller number of MBRs.

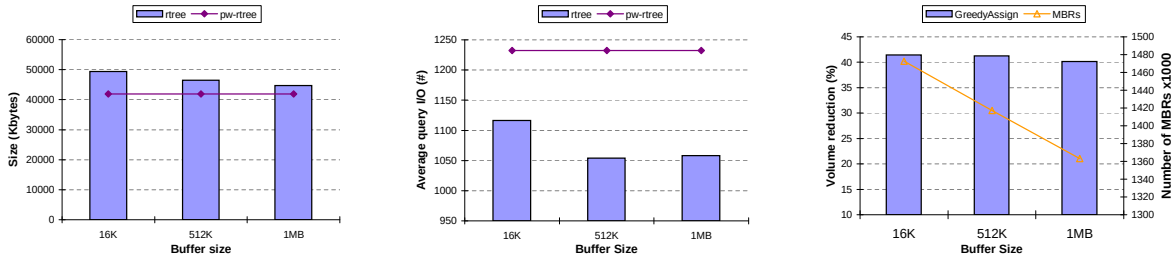
Figure 18 presents the results for the OLDENBURG dataset. The online buffering policies shine in this particular example. By construction, the OLDENBURG dataset issues one update per object per time-instant. As can be seen in Table 1 an average of 83,000 objects per time-instant are alive in this dataset, which straightforwardly implies that on average 83,000 updates per time-instant are expected to enter the buffer. It is evident that the piecewise index will have the minimum possible empty volume and the maximum possible number of MBRs. Intuitively, due to the very large number of indexed copies it is expected in this case that the performance of the piecewise index could be improved by using less space and indexing fewer MBRs. This can be accomplished easily by combining consecutive object updates into single MBRs using the online buffering policies. It is also expected that larger buffer sizes will furnish increased space benefits without affecting query performance, since the buffer will be able to hold a larger number of consecutive object updates.

Figure 18(a) shows that, indeed, the size of the structure decreases considerably when larger buffer sizes are used. Figure 18(b) confirms that query performance im-



(a) FREEWAY. (b) OLDENBURG (c) ATTRACTOR

Fig. 16 The volume reduction curves.



(a) Size (b) Average query I/O (c) Efficiency

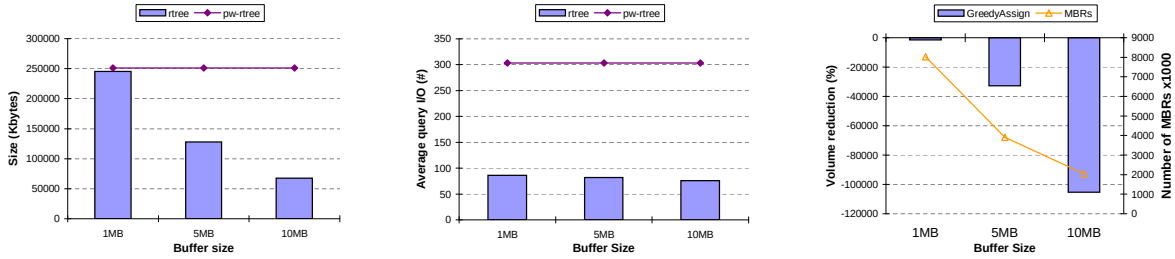
Fig. 17 Online experiments using an ATTRACTOR dataset with 400,000 objects.

proves, as well. The improvement attained is up to 77% in the best case. Finally, Figure 18(c) plots the empty volume reduction for each tree when compared with the piecewise approach. Evidently, the empty volume increases, as anticipated, but the total number of MBRs that need to be indexed decreases considerably.

7 Related Work

Spatio-temporal 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 moving objects have been developed.

Güting et al. [16] discussed the fundamental concepts of indexing spatio-temporal objects. Kollios et al. [26] presented methods for indexing the history of spatial objects that move with linear functions of time. The present article is an extension of this work for arbitrary movement functions. Porkaew et al. [39] proposed techniques for indexing moving points that follow trajectories that are combinations of piecewise functions. Two approaches were presented: The Native Space Indexing, where a 3-dimensional R-tree was used to index individual segments of the movement using one MBR per segment (what was referred to in this article as the piecewise approach), and the Parametric Space Indexing, which uses the coefficients of the movement functions of the trajectories to index the objects in a dual space (where only linear functions can be supported by this approach), augmented by the time-interval during which these movement parameters were valid, in order to be able to answer historical queries. A similar idea was used by Cai and Revesz [9]. The present work indexes the trajectories in the native space — being able to support arbitrary movement functions — and is clearly more robust than the piecewise approach in terms of selecting a query-efficiency vs. space tradeoff. Aggarwal and Agrawal [2] concentrated on nearest neighbor queries and presented a method for indexing trajectories with a special convex hull property in a way that guarantees query correctness in a parametric space. This approach is limited to specific classes of object trajectories and is targeted for nearest neighbor queries only. Pfoser et al. [38] introduced the TB-tree which is an indexing method for efficient execution of navigation and historical trajectory queries. TB-trees are optimized for trajectory preservation, targeting queries that need the complete trajectory information to be retrieved in order to be evaluated, in contrast to conventional range and nearest neighbor queries that need to acquire only partial information. Finally, Zhu et al. [61] proposed the Octagon-Prism tree which indexes trajectories by using octagon approximations. The Octagon-Prism tree is mostly related to the TB-tree. Other issues concentrating on comparison and



(a) Size (b) Average query I/O (c) Efficiency

Fig. 18 Online experiments using an OLDENBURG dataset with 150,000 objects.

similarity of trajectories were discussed by Vlachos et al. [58] and Meratnia et al. [31].

A number of techniques for approximating 1-dimensional sequences have appeared in time-series research. Faloutsos et al. [15] presented a sliding window technique for grouping streaming values into sets of MBRs in order to approximate the time series and reduce the size of the representation. Keogh et al. [23] presented similar algorithms to the ones discussed here for segmenting time-series consisting of piecewise linear segments in an online fashion.

The problem of approximating piecewise linear 2-dimensional curves is of great importance in computer graphics and has received a lot of attention from the field of computational geometry as well, for at least the past thirty years. Kolesnikov [24] presents a concise analysis of all the algorithms that have been proposed in the past, including dynamic programming and greedy solutions like the ones discussed here. The pivotal work on this problem was introduced by Douglas and Peucker [12] and Pavlidis and Horovitz [37]. This work exploited these algorithms in 3-dimensional spaces and also introduced new algorithms for distributing a number of MBRs to a set of 3-dimensional curves.

Methods that can be used to index static spatio-temporal datasets include [10,56,33,47,54,26]. Most of these approaches are based either on the overlapping [8] or on the multi-version approach for transforming a spatial structure into a partially persistent one [13,30,3,57, 28,44]. Finally, a number of spatio-temporal data generators have been proposed recently [55,41,6], and we used some of them in our experiments.

8 Conclusions

This article investigates the problem of indexing spatio-temporal archives. Assuming that objects evolve arbitrarily in time, several techniques are presented that can be used for answering time-instant and small time-period spatio-temporal queries efficiently. The obvious approach for indexing a spatio-temporal archive is to approximate each object using single MBR approximations and then use a multi-dimensional spatial access method. However, this approach is problematic due to excessive empty volume and overlap. This article considers the use of finer object approximations in combination with a multi-version indexing scheme in order to reduce empty volume and overlapping and improve query performance. Algorithms for finding the optimal object approximations in terms of empty volume reduction are presented. In addition, various methods for assigning a given number of MBRs to a set of objects in order to minimize the overall volume are introduced. Moreover, two optimizations for reducing the computational cost of the proposed algorithms are proposed. Various heuristics for finding an appropriate number of MBRs for approximating a dataset by achieving a good trade-off between query performance improvement and volume overhead are discussed. Finally, the online version of the problem is introduced and various methods for improving the overall index quality are presented. Experimental results validate the robustness of all techniques. The combination of the proposed algorithms and the MVR-tree can achieve substantial speed-up and introduce an easily quantifiable query efficiency vs. space utilization tradeoff. Future work should focus on the online version of the problem, where the individual characteristics of spatio-temporal applications make the problem of creating robust indices that preserve historical information in an online manner challenging. In particular, it would be interesting to explore how dynamically evolving object update distributions can be detected in order to tune buffering policies automatically in real-time.

References

1. P. K. Agarwal, L. Arge, and J. Erickson. Indexing moving points. In *Proc. of ACM Symposium on Principles of Database Systems (PODS)*, pages 175–186, 2000.
2. C. C. Aggarwal and D. Agrawal. On nearest neighbor indexing of nonlinear trajectories. In *Proc. of ACM Symposium on Principles of Database Systems (PODS)*, pages 252–259, 2003.

3. B. Becker, S. Gschwind, T. Ohler, B. Seeger, and P. Widmayer. An asymptotically optimal multiversion B-Tree. *The VLDB Journal*, 5(4):264–275, 1996.
4. N. Beckmann, H. Kriegel, R. Schneider, and B. Seeger. The R*-tree: An efficient and robust access method for points and rectangles. In *Proc. of ACM Management of Data (SIGMOD)*, pages 220–231, 1990.
5. R. Benetis, C. Jensen, G. Karciauskas, and S. Saltenis. Nearest neighbor and reverse nearest neighbor queries for moving objects. In *Proc. of International Database Engineering and Applications Symposium (IDEAS)*, pages 44–53, 2002.
6. T. Brinkhoff. A framework for generating network-based moving objects. *GeoInformatica*, 6(2):153–180, 2002.
7. T. Brinkhoff and J. Weitkämper. Continuous queries within an architecture for querying xml-represented moving objects. In *Proc. of Symposium on Advances in Spatial and Temporal Databases (SSTD)*, pages 136–154, 2001.
8. F. Burton, J. Kollias, V. Kollias, and D. Matsakis. Implementation of overlapping B-trees for time and space efficient representation of collection of similar files. *The Computer Journal*, 33(3):279–280, 1990.
9. M. Cai and P. Revesz. Parametric R-tree: An index structure for moving objects. In *Proc. of the COMAD*, 2000.
10. V. P. Chakka, A. Everspaugh, and J. M. Patel. Indexing large trajectory data sets with seti. In *Proc. of Biennial Conference on Innovative Data Systems Research (CIDR)*, 2003.
11. H. D. Chon, D. Agrawal, and A. El Abbadi. Storage and retrieval of moving objects. In *Proc. of the International Conference on Mobile Data Management (MDM)*, pages 173–184, 2001.
12. D. H. Douglas and T.K. Peucker. Algorithms for the reduction of the number of points required to represent a digitised line or its caricature. *The Canadian Cartographer*, 10(2):112–122, 1973.
13. J. Driscoll, N. Sarnak, D. Sleator, and R.E. Tarjan. Making data structures persistent. In *Proc. of ACM Symposium on Theory of Computing (STOC)*, 1986.
14. J. Driscoll, N. Sarnak, D. Sleator, and R.E. Tarjan. Making data structures persistent. *Journal of Computer and System Sciences*, 38(1):86–124, 1989.
15. C. Faloutsos, M. Ranganathan, and Y. Manolopoulos. Fast subsequence matching in time-series databases. In *Proc. of ACM Management of Data (SIGMOD)*, pages 419–429, 1994.
16. R. H. Güting, M. H. Böhlen, M. Erwig, C. S. Jensen, N. A. Lorentzos, M. Schneider, and M. Vazirgiannis. A foundation for representing and querying moving objects. *ACM Transactions on Database Systems (TODS)*, 25(1):1–42, 2000.
17. A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proc. of ACM Management of Data (SIGMOD)*, pages 47–57, 1984.
18. M. Hadjieleftheriou. Spatio-temporal generators. http://www.cs.ucr.edu/~marioh/generators/index.html.
19. M. Hadjieleftheriou, G. Kollios, V. J. Tsotras, and D. Gunopulos. Efficient indexing of spatiotemporal objects. In *Proc. of Extending Database Technology (EDBT)*, pages 251–268, 2002.
20. G. S. Iwerks, H. Samet, and K. Smith. Continuous k-nearest neighbor queries for continuously moving points with updates. In *Proc. of Very Large Data Bases (VLDB)*, pages 512–523, 2003.
21. D. V. Kalashnikov, S. Prabhakar, and S. Hambrusch. Main memory evaluation of monitoring queries over moving objects. *Distributed and Parallel Databases (DADP)*, 15(2):117–135, 2004.
22. I. Kamel and C. Faloutsos. On packing R-Trees. In *Proc. of Conference on Information and Knowledge Management (CIKM)*, pages 490–499, 1993.
23. E. J. Keogh, S. Chu, D. Hart, and M. J. Pazzani. An online algorithm for segmenting time series. In *Proc. of International Conference on Management of Data (ICDM)*, pages 289–296, 2001.
24. A. Kolesnikov. *Efficient algorithms for vectorization and polygonal approximation.* PhD thesis, University of Joensuu, Finland, 2003.
25. G. Kollios, D. Gunopulos, and V. J. Tsotras. On indexing mobile objects. In *Proc. of ACM Symposium on Principles of Database Systems (PODS)*, pages 261–272, 1999.
26. G. Kollios, V.J. Tsotras, D. Gunopulos, A. Delis, and M. Hadjieleftheriou. Indexing animated objects using spatiotemporal access methods. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 13(5):758–777, 2001.
27. C. Kolovson and M. Stonebraker. Segment Indexes: Dynamic indexing techniques for multi-dimensional interval data. In *Proc. of ACM Management of Data (SIGMOD)*, pages 138–147, 1991.
28. A. Kumar, V. J. Tsotras, and C. Faloutsos. Designing access methods for bitemporal databases. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 10(1):1–20, 1998.
29. S. T. Leutenegger, J. M. Edgington, and M. A. Lopez. Str: A simple and efficient algorithm for R-Tree packing. In *Proc. of International Conference on Data Engineering (ICDE)*, pages 497–506, 1997.
30. D. Lomet and B. Salzberg. Access methods for multiversion data. In *Proc. of ACM Management of Data (SIGMOD)*, pages 315–324, 1989.
31. N. Meratnia and R. A. de By. Aggregation and comparison of trajectories. In *Proc. of ACM Symposium on Advances in Geographic Information Systems (GIS)*, pages 49–54, 2002.
32. M. F. Mokbel, X. Xiong, and W. G. Aref. SINA: Scalable incremental processing of continuous queries in spatiotemporal databases. In *Proc. of ACM Management of Data (SIGMOD)*, 2004.
33. M. Nascimento and J. Silva. Towards historical R-trees. In *Proc. of ACM Symposium on Applied Computing (SAC)*, 1998.
34. 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 Symposium on Principles of Database Systems (PODS)*, pages 214–221, 1993.
35. D. Papadias, Y. Tao, J. Zhang, N. Mamoulis, Q. Shen, and J. Sun. Indexing and retrieval of historical aggregate information about moving objects. *IEEE Data Engineering Bulletin*, 25(2), June 2002.

36. D. Papadias, J. Zhang, N. Mamoulis, and Y. Tao. Query processing in spatial network databases. In *Proc. of Very Large Data Bases (VLDB)*, pages 802–813, 2003.
37. T. Pavlidis and S.L. Horovitz. Segmentation of plane curves. *IEEE Transactions on Computers*, 23(8):860–870, 1974.
38. D. Pfoser, C. S. Jensen, and Y. Theodoridis. Novel approaches in query processing for moving object trajectories. In *Proc. of Very Large Data Bases (VLDB)*, pages 395–406, 2000.
39. K. Porkaew, I. Lazaridis, and S. Mehrotra. Querying mobile objects in spatio-temporal databases. In *Proc. of Symposium on Advances in Spatial and Temporal Databases (SSTD)*, pages 59–78, 2001.
40. S. Prabhakar, Y. Xia, D. Kalashnikov, W. G. Aref, and S. E. Hambrusch. Query indexing and velocity constraint indexing: Scalable techniques for continuous queries on moving objects. *IEEE Transactions on Computers*, 51(10):1–17, 2002.
41. J-M. Saglio and J. Moreira. Oporto: A realistic scenario generator for moving objects. *GeoInformatica*, 5(1):71–93, 2001.
42. S. Saltenis and C. S. Jensen. Indexing of moving objects for location-based services. In *Proc. of International Conference on Data Engineering (ICDE)*, pages 463–472, 2002.
43. S. Saltenis, C. S. Jensen, S. T. Leutenegger, and M. A. Lopez. Indexing the positions of continuously moving objects. *SIGMOD Record*, 29(2):331–342, 2000.
44. B. Salzberg and V. J. Tsotras. Comparison of access methods for time-evolving data. *Communications of the ACM (CACM)*, 31(2):158–221, 1999.
45. T. K. Sellis, N. Roussopoulos, and C. Faloutsos. The R+-Tree: A dynamic index for multi-dimensional objects. In *Proc. of Very Large Data Bases (VLDB)*, pages 507–518, 1987.
46. Y. Tao, C. Faloutsos, D. Papadias, and B. Liu. Prediction and indexing of moving objects with unknown motion patterns. In *Proc. of ACM Management of Data (SIGMOD)*, pages 611–622, 2004.
47. Y. Tao and D. Papadias. MV3R-Tree: A spatio-temporal access method for timestamp and interval queries. In *Proc. of Very Large Data Bases (VLDB)*, pages 431–440, 2001.
48. Y. Tao and D. Papadias. Time-parameterized queries in spatio-temporal databases. In *Proc. of ACM Management of Data (SIGMOD)*, pages 334–345, 2002.
49. Y. Tao and D. Papadias. Performance analysis of R*-Trees with arbitrary node extents. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 16(6):653–668, 2004.
50. Y. Tao, D. Papadias, and Q. Shen. Continuous nearest neighbor search. In *Proc. of Very Large Data Bases (VLDB)*, pages 287–298, 2002.
51. Y. Tao, D. Papadias, and J. Zhang. Cost models for overlapping and multi-version structures. *ACM Transactions on Database Systems (TODS)*, 27(3):299–342, 2002.
52. Y. Tao, J. Sun, and D. Papadias. Analysis of predictive spatio-temporal queries. *ACM Transactions on Database Systems (TODS)*, 28(4):295–336, 2003.
53. Y. Tao, J. Sun, and D. Papadias. Selectivity estimation for predictive spatio-temporal queries. In *Proc. of International Conference on Data Engineering (ICDE)*, pages 417–428, 2003.
54. Y. Theodoridis, T. Sellis, A. Papadopoulos, and Y. Manolopoulos. Specifications for efficient indexing in spatiotemporal databases. In *Proc. of Scientific and Statistical Database Management (SSDBM)*, pages 123–132, 1998.
55. Y. Theodoridis, J. R. O. Silva, and M. Nascimento. On the generation of spatiotemporal datasets. In *International Symposium in Spatial Databases, (SSD)*, pages 147–164, 1999.
56. T. Tzouramanis, M. Vassilakopoulos, and Y. Manolopoulos. Overlapping linear quadtrees and spatio-temporal query processing. *The Computer Journal*, 43(3):325–343, 2000.
57. P.J. Varman and R.M. Verma. An efficient multiversion access structure. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 9(3):391–409, 1997.
58. M. Vlachos, G. Kollios, and D. Gunopulos. Discovering similar multidimensional trajectories. In *Proc. of International Conference on Data Engineering (ICDE)*, pages 673–684, 2002.
59. O. Wolfson, A. P. Sistla, S. Chamberlain, and Y. Yesha. Updating and querying databases that track mobile units. *Journal of Distributed and Parallel Databases (DAPD)*, 7(3):257–387, 1999.
60. J. Zhang, M. Zhu, D. Papadias, Y. Tao, and D. L. Lee. Location-based spatial queries. In *Proc. of ACM Management of Data (SIGMOD)*, pages 443–454, 2003.
61. H. Zhu, J. Su, and O. H. Ibarra. Trajectory queries and octagons in moving object databases. In *Proc. of Conference on Information and Knowledge Management (CIKM)*, pages 413–421, 2002.