Indexing Animated Objects Using Spatiotemporal Access Methods

George Kollios
Boston University
gkollios@cs.bu.edu

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

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

Alex Delis
Polytechnic University
ad@naxos.poly.edu

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

Abstract

We present a new approach for indexing animated objects and efficiently answering queries about their position in time and space. In particular, we consider an animated movie as a *spatiotemporal* evolution. A movie is viewed as an ordered sequence of frames, where each frame is a 2-dimensional space occupied by the objects that appear in that frame. The queries of interest are range queries of the form: "find the objects that appear in area S between frames f_i and f_j", as well as nearest neighbor queries like: "find the q nearest objects to a given position A between frames f_i and f_j". The straightforward approach to index such objects considers the frame sequence as another dimension and uses a 3-dimensional access method (like an R-Tree or its variants). This however assigns long "lifetime" intervals to objects that appear through many consecutive frames. Long intervals are difficult to cluster efficiently in a 3-dimensional index. Instead, we propose to reduce the problem to a partial-persistence problem. Namely, we use a 2-dimensional access method that is made partially persistent. We show that this approach leads to faster query performance while still using storage proportional to the total number of changes in the frame evolution. What differentiates this problem from traditional temporal indexing approaches is that objects are allowed to move and/or change their extent continuously between frames. We present novel methods to approximate such object evolutions. We formulate an optimization problem for which we provide an optimal solution for the case where objects move linearly. Finally, we present an extensive experimental study of the proposed methods. While we concentrate on animated movies, our approach is general and can be applied to other spatiotemporal applications as well.

1 Introduction

We consider the problem of indexing objects in animated movies. In our setting, an animated movie corresponds to an ordered sequence of frames. In this sequence, each frame (or screen) is a 2-dimensional space that contains a collection of objects. As the movie proceeds, this collection of objects changes from one frame to the next (new objects are added, objects move, change in size, disappear, etc.) A conceptual view of a movie sequence appears in Figure 1. The x and y axes represent the 2-dimensional frame screen while the f axis corresponds to the frame sequence. Frame f_1 contains objects o_1 (which is a point) and o_2 (which is a region). At frame f_2, object o_3 is inserted while o_1 moves to a new position and o_2 shrinks. Object o_1 moves again at frame f_5; o_2 continues to shrink and disappears at frame f_5. For the purposes of editing or assembling movie sequences, it is important to have efficient ways to access and replay all, or parts, of such movies. In particular we are interested in topological range queries of the form: "find all objects that appear in area S between frames f_i and f_j", and nearest neighbor queries like: "find the q nearest located objects to position A between frames f_i and f_j". Variables S and A take values inside the 2-dimensional frame screen. An example query is illustrated in figure 1: "find all objects inside area S in frame f_3"; only object o_1 satisfies this query.

Objects in movie sequences can be referred at three different abstraction levels, namely: *raw, feature* and *semantic* levels [32, 41, 19, 26]. At the raw abstraction level, an object is an aggregation of pixels from a frame. In this level, the interest is mainly in object comparisons which are performed in a pixel by pixel fashion. At the next level, frames are characterized by image features such as gray scale, luminance or color histogram. Objects are identified through frame regions that consist of homogeneous feature vectors. Queries in this level are similarity queries in a multidimensional feature space. At the third level, semantic information about the objects and their relative positions in a frame has already been extracted and can thus be used to index these objects. Such semantic information leads to content-based queries, i.e., queries about the actual objects in a movie.

Most of the previous research on indexing images or movies has concentrated on the raw and feature levels and examines similarity based queries [15, 17]. Our work is different in that: (i) it deals with the semantic level, and, (ii) the queries are topological in nature (i.e., the relative position of objects in space and frame is of importance). This is a novel problem. Recently, [48] examines similar topological queries for multimedia applications but it addresses a special case (the "degenerate" case discussed below).

We propose to index an animated movie sequence as a spatiotemporal evolution. That is, frame ids f_i, with $i > 0$, correspond to consecutive time instants. In the rest of the paper, we will use the terms time instant and frame number interchangeably. By considering an animated movie as a spatiotemporal evolution each object is assigned a record with a "lifetime" interval $[f_i, f_j)$ created by the frames when the object was added (insertion frame) and deleted (deletion frame) from the movie. We consider two types of evolutions, namely the *degenerate* case and the *general* case. In the degenerate case (Figure 2), objects are simply

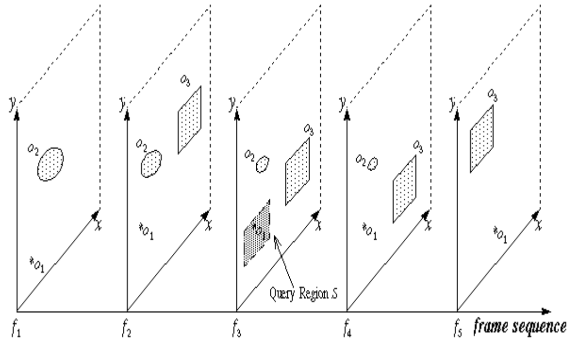

Figure 1: A conceptual view of a movie sequence.

added or deleted from the movie. That is, during its lifetime, an object remains in the same position and retains the same 2-dimensional extent (region). This type of evolution is rather static. The only changes in the degenerate evolution are object insertions and deletions. A deletion is a "logical" operation that simply updates the lifetime interval of the deleted object's record. Important for the design of the index is the number of object insertions N; this represents the total number of records ever created and is a measure for the storage consumed by the index.

More interesting (and realistic) is the general case where objects are allowed to move, and grow/shrink among frames during their lifetime (Figure 1). However, in the general case it is not obvious how position and extent changes can be quantified as object insertions. Consider an object that moves from position A in frame f_i to a new position C in the next frame f_{i+1}. The simplest way to represent such movement is to delete the object from position A in frame f_{i+1} and reinsert it in position C at the same frame. This creates two records for this object, one record with position A and lifetime ending at f_{i+1} and one record with position C and lifetime starting at f_{i+1}. The object's lifetime has been "artificially" truncated into two records with consecutive and non-overlapping intervals. This approach is not efficient if objects alter positions/extents continuously through frames. A large number of artificial insertions is created and thus the index storage requirements increase.

A better solution is to store the functions describing how objects move or vary their extents. In animated movies an object's frame evolution is represented by some function [1]. Even though general functions can be used, for simplicity we assume an object can move or grow/shrink through a *linear* function of time. Then a new record is inserted only when the parameters describing an object's (movement or extent) function change. The new record will maintain the object's lifetime under the new movement/extent function. Thus the number of insertions N in the general evolution case corresponds to: (i) regular object insertions, and, (ii) insertions due to function parameter changes.

We distinguish between two different modes of operation. In the On-Line mode, when a new object is inserted at frame f_i, its deletion frame is not yet known, so its lifetime is initiated as $[f_i, now)$ where now is a variable representing the (ever increasing) current frame number. If the object gets deleted at a later frame f_j, its lifetime interval is updated to $[f_i, f_j)$. In contrast, the Off-Line mode assumes advance knowledge of the insertion and deletion frames for each object as well as its positions and extents during its lifetime. Clearly, in the Off-Line mode, the constructed index is expected to be more efficient since we have more information about the data. This paper concentrates on the Off-Line mode, since this is the case in animated movies. There are other spatiotemporal applications where the future of the evolution is unknown and the On-Line mode is more appropriate (for example storing the evolution of a collection of cars moving in the plane).

Using the spatiotemporal approach, one straightforward way to index animated objects is to consider time as another index dimension. Each object is then stored as a 3-dimensional rectangle in a traditional spatial index (e.g. an R-Tree [16] or its variations [25, 21, 38, 6]) where the "height" of the rectangle corresponds to the object's lifetime interval. The "base" of the rectangle corresponds to the largest 2-dimensional minimum bounding rectangle (MBR) that the object obtained during its lifetime. While simple to implement, this approach does not take advantage of the specific properties of the time (frame) dimension. Objects that remain unchanged for many frames will have long lifetimes and thus, they will be stored as long rectangles. A long-lived rectangle determines the length of the time range associated with the index node in which it resides. This creates node overlapping and leads to decreased query performance [24]. Since all data is available in advance, one way to achieve better clustering is to use "packed" R-Trees, like the Hilbert R-Tree [21] or the STR-Tree [25].

Another attempt to overcome the problems with clustering long lifetime intervals, is to fragment them in smaller and easily manageable pieces. This approach has been proposed in the Segment R-Tree (SR-Tree) [23] which combines properties of the R-Tree and the Segment Tree [37]. Interval fragmentation implies storing fragments of the same interval in many places. This, in the worst case creates a logarithmic storage overhead and requires more elaborate query processing.

In contrast, we propose to use a different approach in indexing animated objects which combines a spatial index (R-Tree) with the partially persistent methodology. A data structure is called *persistent* [14] if an update applied to it creates a new version of the data structure while the previous version is still retained and can be accessed. A data structure that does not keep its past is called *ephemeral*. *Partial* persistence implies that all versions can be accessed but only the newest version can be modified.

Partial persistence fits nicely with the degenerate evolution case since in that case an update corresponds to either an object addition or a deletion. Methods to make a disk based structure (in particular a B-tree) partially persistent have appeared in the area of temporal databases [20, 5, 27, 49, 24, 36]. [24] presents the Bitemporal R-Tree which is a partially-persistent R-Tree used to index bitemporal objects. This partially-persistent R-Tree can be easily extended to index the degenerate case of animated objects.

The general evolution case where objects change continuously is different. One approach is to represent an object's movement or extent change by the largest 2-dimensional MBR that the object obtained during its evolution (*maxMBR*). For example, in Figure 1 the largest MBR in the evolution of object o_2 occurs at frame f_1. Then the evolution of o_2 can be represented by the insertion of this MBR at frame f_1 and the deletion of the *same* MBR at frame f_4. While this representation creates only one record, it creates a large empty space for the partially persistent methodology. Even though object o_2 reduces its extent as frames advance, it is still represented by the larger MBR. Empty space in R-Trees is known to deteriorate query response time.

To reduce the empty space we propose to introduce a limited number of artificial updates. An artificial update deletes an existing object and reinserts it, thus adding an extra record. To maintain the index storage linear to N we limit the number of artificial updates to be a fraction of N. To apply the partially persistent methodology one must first decide: (i) which objects should be artificially updated and, (ii) on what frames the artificial updates should be created. We first formulate these questions as an optimization problem. Then we provide a greedy algorithm that optimally finds the artificial updates for the case when objects move with linear functions. The algorithm is based on a special *monotonicity* property that holds for linear changes. This property holds also when objects change one of their (two) extent dimensions linearly. If both extent dimensions change, the algorithm does not provide the optimal solution. Nevertheless, it serves as a good heuristic that performs very well in practice.

To show the merits of our approach we compare the Partial Persistent R-Tree with: (i) the standard 3-dimensional R-Tree [6], (ii) the packed STR-Tree [25] and (iii) the Segment R-Tree [23]. Both selection and nearest neighbor queries are examined. Extensive experimental results indicate that the query performance of the Partial Persistent R-Tree consistently outperforms its competitor approaches for a number of diverse query workloads. Moreover, the storage of the Partial Persistent R-Tree remains linear to the number of insertions N.

Section 2 provides background on the partially persistent R-Tree and the degenerate case. Section 3 discusses the general case of animated objects as well as the greedy algorithm. Section 4 contains experimental results. Section 5 presents related work while 6 concludes the paper and presents future research work.

2 Preliminaries

For the Off-Line problem we measure the performance of an index using two costs: query response time and storage requirements. Given the large sizes of animated movies we assume the data is disk resident. Hence the indexing scheme should be designed so as to minimize the number of page transfers (I/O's) between the disk and main memory needed to answer a query while keeping the index storage requirements small. There are two basic parameters that affect performance: (i) the number of insertions N, (ii) the number of records that fit in a page B. We assume that one I/O transfers one page. Ideally, we would like our index solutions to use linear storage, i.e., $O(\frac{N}{B})$ disk pages [18]. Note that for the On-Line problem an additional cost measure is the index update time (the time to process an update). This is not critical in the Off-Line setting since the whole set of updates is known in advance and the index is built once.

To further exemplify the above costs, consider two obvious but inefficient ways to address topological queries about animated movies. The first is to store in the database snapshots of all movie frames. This "snapshot" approach provides fast access to the frames of interest, but extra work is needed to locate the objects in the query area S. The main disadvantage however is the high storage redundancy ($O(\frac{N^2}{B})$). The second approach is to store the changes between frames in a "log". The storage requirement is $O(\frac{N}{B})$ but the query time becomes large. At worst the whole log must be read resulting to $O(\frac{N}{B})$ query time. A combination of the above would be to store a number of frame snapshots and the sequences of changes between successive snapshots (similar idea as in MPEG). However, this approach has the following disadvantages: (i) it is not obvious how often to keep snapshots (frequent snapshots increase storage requirements, fewer snapshots increase query time), (ii) locating the objects in the query area S still requires extra effort that affects the query response time.

We proceed with a discussion of the 3-dimensional R-Tree approaches, namely the R-Tree, the STR-Tree and the SR-Tree. Then we summarize the properties of the Partially Persistent R-Tree. The degenerate evolution case is used as an example in this section.

2.1 The 3-dimensional R-Tree approaches

The R-Tree [16] is a hierarchical, height-balanced index. It is a generalization of the B-tree for multidimensional spaces. Multidimensional objects are represented by a conservative approximation, usually their MBR. This approximation may introduce empty or dead space, which is the part of the MBR that is not covered by the object. The R-Tree consists of directory and leaf (data) nodes, each node corresponding to one disk page. Directory nodes contain index records of the form $(container, ptr)$ where ptr is a pointer to a node in the next level of the tree and *container* is the MBR of all the records in the descendent node. Leaf nodes contain data records of the form $(container, oid)$ where oid is the object-identifier of the real object and *container* is its MBR. All the nodes except the root must have at least m records (usually $m = B/2$). Thus the height of the tree is at most $log_m n$ where n is the total number of objects. Searching in the R-Tree is similar to the B-tree. At each directory node all records are tested against the query and then all child nodes that satisfy the query are visited. However, a drastic difference from the B-tree is that the MBRs in a R-Tree node are allowed to overlap. As a result, when answering a query, multiple paths may be followed, although some of these paths may not contribute to the answer. At worst all leaf nodes may be visited, however in practice R-Trees have been shown to work much faster.

For a degenerate evolution (figure 2), an object can be represented by a 3-dimensional MBR starting from the frame the object appears in the screen until the frame when it disappears. The R-Tree will attempt to store records with long lifetimes (like object o_3 in Figure 2) as long MBRs, causing a great deal of overlapping between the nodes of the R-Tree. Large overlapping is known to decrease the R-Tree query performance drastically ([24, 23]).

Another alternative for indexing the 3-dimensional MBRs is to use a packed R-Tree ([21, 25]). In our experiments we consider the STR-Tree [25]. The basic idea in a packed tree is first to sort the indexed objects and then fill the data pages by placing B consecutive objects in each page. The same procedure is repeated recursively for the upper levels of the tree until the root node. The advantage of packing is that the storage utilization is almost 100%. Also for specific type of datasets and queries, a packed R-Tree has been shown to be more efficient for answering queries than a traditional R-Tree (like point datasets and point queries [25]). However, there are cases where a packed R-Tree is worst than a traditional R-Tree, in terms of query performance. This is because the packed R-Tree may place together in a page objects that are consecutive in ordering but which are not close in space. This will create large empty space and possibly some overlap which affect query performance.

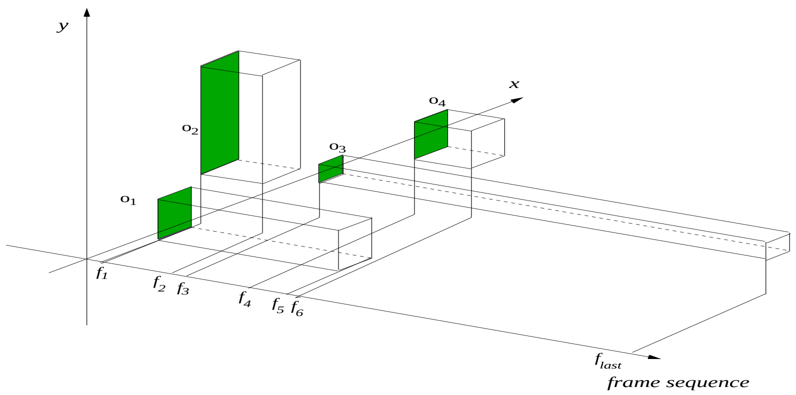

Figure 2: A degenerate spatiotemporal evolution with four objects.

The SR-Tree [23] has been introduced as a remedy for storing objects with long intervals. Intervals can be stored in both leaf and non-leaf SR-Tree nodes. An interval I is placed to the highest level node X such that I spans at least one of the intervals represented by X's children nodes. If I does not span X, spans at least one of its children but is not fully contained in X, then I is fragmented. Using this idea, long intervals will be placed in higher levels of the tree, which in turn decreases overlapping in the SR-Tree leaf nodes. In contrast, a long interval stored in a leaf node of a regular R-Tree will "elongate" the area of this node thus exacerbating the overlap problem.

However, if large numbers of spanning records or fragments of spanning records are stored higher in the SR-Tree, the fan-out of the index may decrease as there is less room for pointers to children nodes. [23] suggests to vary the size of the nodes in the tree, making higher-up nodes larger. "Varying the size" of a node means that several pages are used for one node. This scheme will "add" some page accesses to the query performance cost.

If an interval is inserted at a leaf SR-Tree node (because it did not span any other node) it may cause the boundaries of the MBR covered by this leaf node to be expanded. Similar expansions may also be needed on the MBRs of all nodes on the path to this leaf node. This in turn can change the spanning relationships since records may no longer span children which have been expanded. Such records are reinserted in the tree, possibly being demoted to occupants of nodes they previously spanned. Splitting nodes may also cause changes in spanning relationships as they make children smaller -former occupants of a node may be promoted to spanning records in the parent. Because of fragmentation, the worst case storage requirement for an SR-Tree is $O((N/B)log_B(N/B))$ [36]. However, this is a pathological scenario that rarely happens in practice. To further improve query performance, [23] proposed the Skeleton SR-Tree, an SR-Tree that pre-partitions the entire domain into a number of regions. This pre-partition is based on an initial estimate about the data distribution and the number of intervals to be inserted. After partitioning, the Skeleton SR-Tree is populated with data.

2.2 The Partially-Persistent R-Tree

Consider the example in Figure 2 and assume that the objects in frame f_1 are indexed by a 2-dimensional R-Tree. As the frame number advances, this 2D R-Tree evolves, by applying the updates (object additions/deletions) as they occur in the appropriate frames. Storing this 2D R-Tree evolution corresponds to making a 2D R-Tree partially persistent.

By "viewing" a degenerate evolution as a partial persistence problem, we obtain a double advantage. First we dis-associate the indexing requirements within a frame from the frame sequence. More specifically, indexing within a frame is provided from the properties of the ephemeral 2D R-Tree while the frame evolution support is achieved by making this tree partially persistent. Second, partial persistence avoids the long 3-dimensional rectangles and thus the extensive overlapping due to long lifetimes. Moreover, the partially persistent R-Tree uses storage that is linear to the number of insertions in the degenerate frame evolution. To illustrate the partial persistence methodology we present how a 2D R-Tree is made partially persistent. Note that the methodology applies to other spatial indexes; we use a 2D R-Tree for simplicity.

While conceptually the partially-persistent R-Tree (PPR-Tree) [24] records the evolution of an ephemeral R-Tree, it does not physically store snapshots of all the frames in the ephemeral R-Tree evolution. Instead it records the evolution updates efficiently so that the storage remains linear, while still providing fast query time.

The PPR-Tree is actually a directed acyclic graph of nodes (a node corresponds again to a disk page). Moreover, it has a number of root nodes, where each root is responsible for recording a subsequent part of the ephemeral R-Tree evolution. The various roots can be easily accessed through a linear array called the root*. Each entry in the root* contains a lifetime interval and a pointer to the root responsible for that interval.

Data records in the leaf nodes of a PPR-Tree maintain the frame evolution of the ephemeral R-Tree data objects. Each data record is thus extended to include the two lifetime fields: *insertion-frame* and *deletion-frame*. Similarly, index records in the directory nodes of a PPR-Tree maintain the evolution of the corresponding index records of the ephemeral R-Tree and are also augmented with insertion-frame and deletion-frame fields.

An index or data record is called *alive* for all frames during its lifetime interval. A leaf or a directory node is called *alive* if it has not been *split*. With the exception of root nodes, for all frame numbers that a node is alive it must have at least D alive records ($D < B$). This requirement enables clustering the objects that are alive at a given frame number in a small number of nodes (pages), which in turn will minimize the query I/O. The PPR-Tree is created incrementally following the update sequence. Consider an update (insertion or deletion) at frame f_i. To process this update the PPR-Tree is searched to locate the target leaf node where the update must be applied. This step is carried out by taking into account the lifetime intervals of the index and the data records visited. This implies that the search follows records that are alive at frame f_i. After locating the target leaf node, an insertion update adds a data record with an interval $[f_i, now)$ to the target leaf node. A deletion update will update the deletion-frame of a data record from now to f_i.

An update leads to a *structural* change if at least one new node is created. *Non-structural* are those updates which are handled within an existing node. An insertion update triggers a structural change if the target leaf node already has B records. A deletion update triggers a structural change if the resulting node ends up having less than D alive records as a result of the deletion. The former structural change is a *node overflow*; the latter is a *weak version underflow* [5]. Node overflow and weak version underflow need special handling: a *split* is performed on the target leaf node. This is reminiscent of the time-split operation reported in [27] and the page copying concept proposed in [46]. Splitting a node x at frame f is performed by copying to a new node y the records alive in node x at f. Node x is considered *dead* after frame f. (We can assume that the deletion-frame field of all x's alive records is changed to f even though this is not needed in practice).

To avoid having a structural change on node y soon, when a new node is created the number of alive records must be in the range $D + e$ and $B - e$ (where e is a predetermined constant). This allows at least e non-structural changes on this node before a new structural change. Thus before the new node is incorporated in the structure it may have to be merged with another node (this happens if y has less than $D + e$ alive records and is called a *strong version underflow*), or, "key-split" into two nodes (if y has more than $B - e$ alive nodes, i.e., a *strong version overflow*). For details we refer to [24, 49, 5].

An example of a PPR-Tree is shown in Figure 5 using the evolution presented in Figure 3 and Figure 4. In particular, Figure 3 shows the MBRs of 20 objects (numbered from 1 to 20) that appeared in a small animated video while Figure 4 depicts the lifetimes of these objects. Here $B = 5$, $D = 2$ and e is set to 1. The root* entries show the lifetimes associated with each pointer and the pointers to the root nodes of the PPR-Tree. Similarly, index nodes depict the lifetime intervals and the corresponding pointers to the next level of the tree. For simplicity, data nodes show only the stored object ids (and not their lifetimes). Note that an object can be stored in more than one data page. For example object 14 is stored in 5 data pages since it has a long lifetime.

Answering a range query about region S and frame f has two parts. First, the root which is alive at f is found. This part is conceptually equivalent to accessing the root of the ephemeral R-Tree which indexes frame f. Second, the objects intersecting S are found by searching this tree in a top-down fashion as in a regular R-Tree. The lifetime interval of every record traversed should contain the frame f, and at the same time it's MBR should intersect region S. Answering a query that specifies a frame interval $[f, f')$ is similar. First all roots

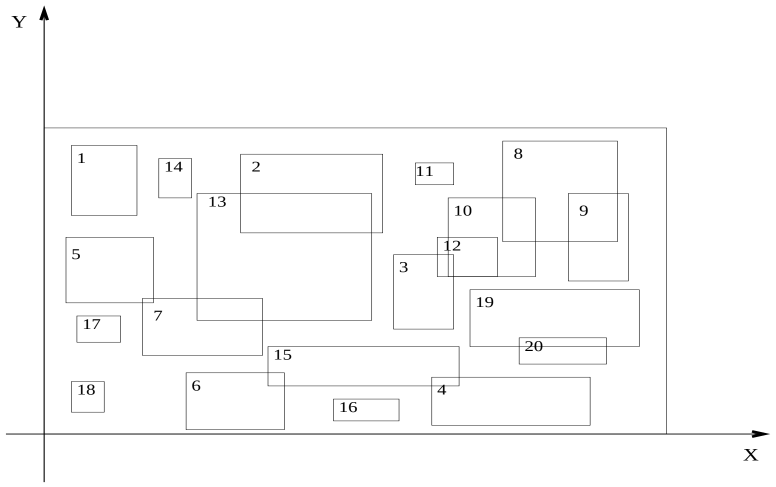

Figure 3: Various object MBRs.

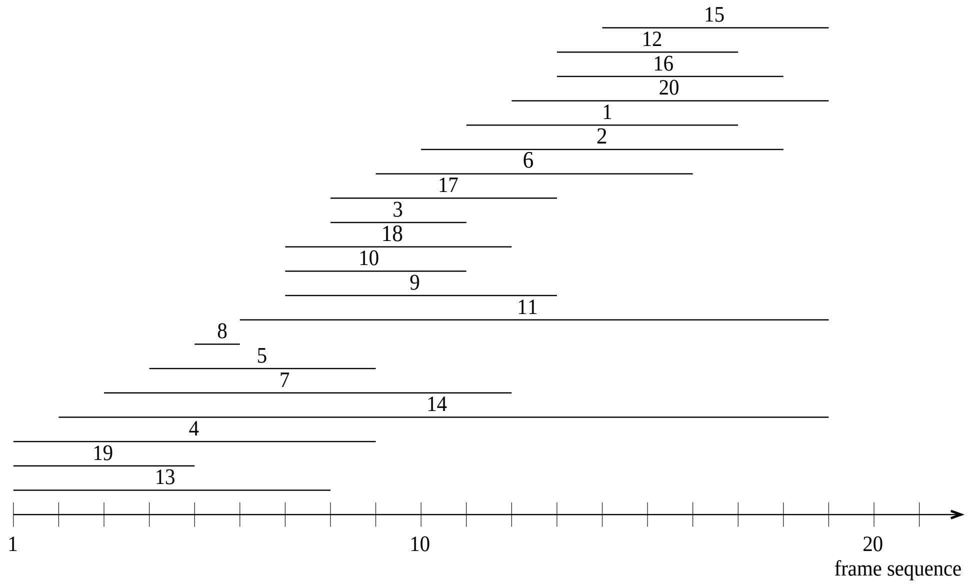

Figure 4: Corresponding object lifespans.

with lifetime intervals intersecting the frame range are found and so on. Since the PPR-Tree is a graph, some nodes are accessible by multiple roots. Re-accessing nodes can be avoided by keeping a list of accessed nodes.

To answer nearest neighbor queries we use the algorithm proposed in [33] and later refined in [10]. The query consists of a point or object and a frame sequence. The answer contains the q nearest objects that are closest to the query object during the specified frame sequence. The algorithm proposed in [33] can be used directly; the only difference is in the way distances are computed. All objects that are not alive during the query frame sequence have infinite distance to the query object. On the other hand for the objects that have lifetimes intersecting the query frame sequence, the distance is computed using their extent dimensions. The algorithm

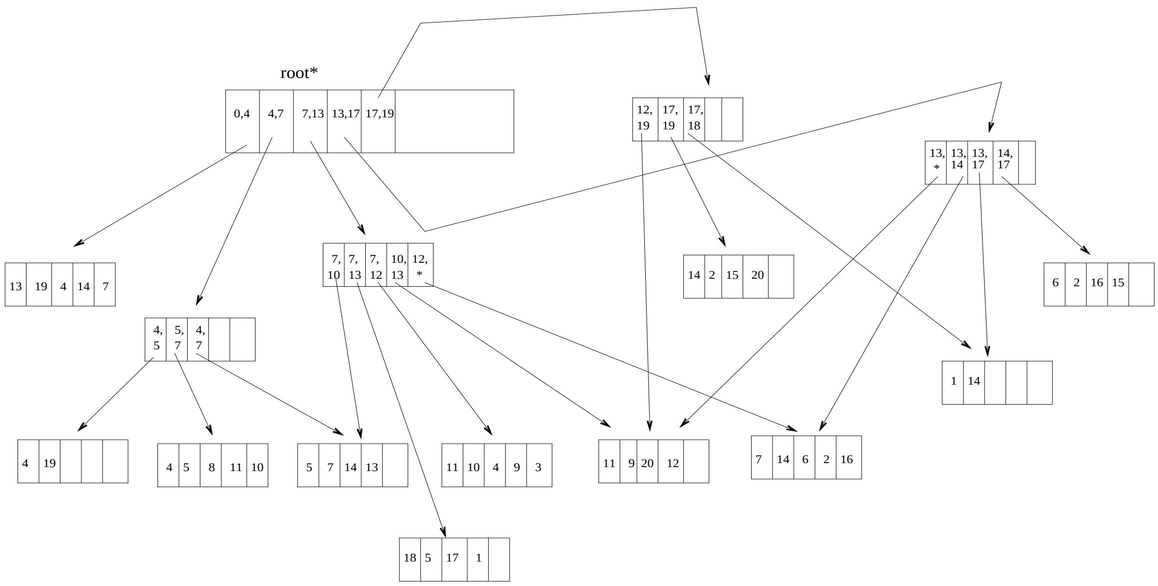

Figure 5: The PPR-Tree created from the above object evolution.

first visits the root of the tree and then traverses the tree in a top-down fashion. At each node, a list of the subtrees is kept, ordered by the minimum distance of each subtree to the query object. The subtrees are then visited in sorted order. A subtree is pruned from the search if the minimum distance of this subtree is larger than the distance of the $q-th$ nearest object found so far. The same algorithm is used with the PPR-Tree, after the root of the corresponding ephemeral R-Tree is found.

3 The General Evolution Case

The problem in the general case, is how to represent objects that continuously change positions and/or extent over time. Objects are still represented by MBRs but an efficient solution should minimize the empty space introduced by the MBR representation. To achieve this goal we introduce artificial deletions and reinsertions of objects. We proceed with some definitions.

Definition 1 *Aspatiotemporal object O^L is defined as the 3-dimensional volume created by a 2-dimensional spatial object o that moves and/or changes its extent during its lifetime interval L.*

In the rest of the paper we use capital letters to represent spatiotemporal objects; we sometimes drop the lifetime exponent to simplify the notation.

Definition 2 *Let G be a set of spatiotemporal objects.* **Empty:** *$G \rightarrow R$ defines a function that takes as input a spatiotemporal object and returns the empty space that is introduced by approximating the spatiotemporal object by a 3-dimensional MBR.*

Figure 6 shows the movement of object o_1 from frame f_1 to frame f_2. The corresponding spatiotemporal object is the shaded volume; the empty space is the volume that is contained inside the 3-dimensional MBR and is not shaded.

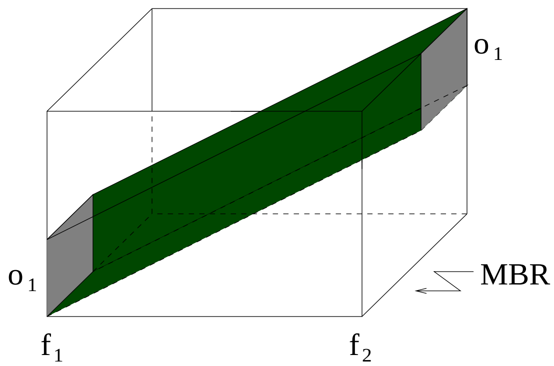

Figure 6: *A spatiotemporal object.*

Next we define the (artificial) split operation. Consider the spatiotemporal object created by the evolution of object o from frame f_i to frame f_j. A split operation at frame f_s, where $f_i < f_s < f_j$, artificially deletes object o at frame f_s and reinserts it at the same frame with the same extent at the same position. As a result, the original spatiotemporal object $O^{[f_i,f_j)}$ is replaced by two new spatiotemporal objects, namely $O^{[f_i,f_s)}$ and $O^{[f_s,f_j)}$. By adding two new spatiotemporal objects instead of the original one, the overall MBR empty space is expected to decrease since the original evolution is represented using more detail. A similar idea has been used in [30, 31] for indexing spatial objects with the help of Z-codes.

Figure 7 shows the result of a split operation performed at frame f_s on the object evolution of Figure 6. The view from the $x - axis$ is depicted. That is, the spatial object is shown as an interval that moved along the $y - axis$ from frame f_1 to frame f_2. The gain in empty space is equal to $E_1 + E_2$. For the partially persistence approach, the above split is seen as having an object y_1 with lifetime $[f_1, f_s)$ and an object y_2 with lifetime $[f_s, f_2)$. Without the artificial split, we had an object y_{tot} with lifetime $[f_1, f_2)$. The rationale in splitting is to decrease the empty space, and consequently the overlapping of nodes in the ephemeral R-Tree. Thus, the query performance of the index is improved at the expense of using more storage. Every split increases the number of the indexed objects by one.

A more general split operation allows a spatiotemporal object to be divided multiple times.

Definition 3 *Let $O^{[f_i,f_j)}$ be a spatiotemporal object.* **Split-k(O)** *is defined as an operation that partitions $O^{[f_i,f_j)}$ into $k+1$ objects using sp_l splitting points, where $f_i \leq sp_l \leq f_j, l = 1, ..., k$.*

For objects that move with a linear motion over time, the best choice for k splitting points over a given spatiotemporal object (so as to minimize the empty space) is to take equidistant splits during the lifetime of the spatiotemporal object.

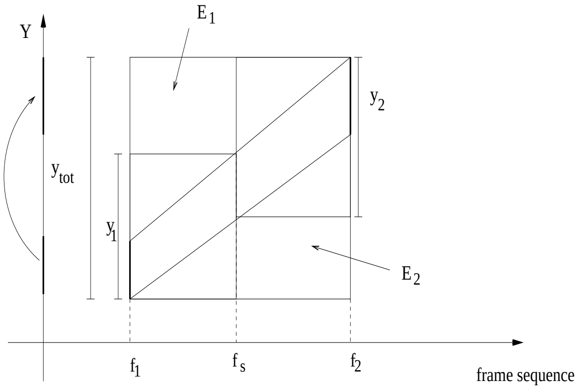

Figure 7: Split operation of an 1-dimensional object that moved continuously from frame f_1 to f_2.

Lemma 1 *Let $O^{[f_s,f_e]}$ be a spatiotemporal object created by a linear movement, and m is the number of splits. The set $\{f_s + i * \frac{f_e - f_s}{m+1}, i = 1, ..., m\}$ is the set of splitting points that maximizes the gain in empty space.*

Proof.

We first consider a point object that moved with a linear motion between frames f_s and f_e. Let α and β be the speed of the object on X and Y directions respectively. If $m = 1$, then we need to find one splitting point that maximizes the gain in empty space. Assume that we split at position $f \in [f_s, f_e]$. Then the gain in empty space is given from the following formula:

$$G(f) = V - (f^3\alpha\beta + (F - f)^3\alpha\beta)$$

where $F = f_e - f_s$, and $V = F * F\alpha * F\beta$.

The value of f that minimizes the above gain function is $f = f_s + \frac{f_e - f_s}{2}$ (by solving the equation $\frac{dG}{df} = 0$.) Consider now the case where $m = 2$ and we want to find the positions of the splitting points f_1 and f_2. Let assume that we decided the optimal position of the first split f_1. Then, we need to find the position of the second split, that is, a position between f_1 and f_e. This problem is equivalent to the problem with $m = 1$, but for the spatiotemporal object $O^{[f_1,f_e]}$. Using the result for $m = 1$, the best splitting position is the middle point between f_1 and f_e. Similarly, the best splitting position for f_1 is the middle point between f_s and f_2. Therefore, the best values for f_1 and f_2 are $f_s + \frac{f_e - f_s}{3}$ and $f_s + 2 * \frac{f_e - f_s}{3}$ respectively. Using the same argument for $m = k$ we get that the best splitting points are $\{f_s + i * \frac{f_e - f_s}{k+1}, i = 1, ..., k\}$.

If the object is a rectangle, let x and y be the size of the the X and Y side respectively (as in figure 9(c)). Then if we split the spatiotemporal object using one splitting point f, the function that gives the gain in empty space is:

$$G(f) = V - \{\alpha\beta((F - f)^3 + f^3) + (\alpha y + \beta x)((f - f)^2 + f^2) + xyF\}$$

where $F = f_e - f_s$, and $V = F * (F\alpha + x) * (F\beta + y)$.

The value of f that minimizes the above function is again $f = f_s + \frac{f_e - f_s}{2}$. For $m > 1$, we can use the same argument as above to prove that the same splitting points are optimal here, too. □

Note that equidistant splits are optimal for objects that (i) move linearly (while retaining the same extent) or (ii) change one of their extent dimensions linearly. However, it is not the optimal choice for objects that change both their extent dimensions. Although, there are ways to compute the best splitting points even in that scenario, these methods are computationally expensive. Therefore, we concentrate on linearly moving objects (i.e., no extent change) for which we will provide an optimal solution. Our solution can then be used as a good heuristic for the optimal choice of splits even for objects that change both extent dimensions linearly.

Consider now the problem of choosing the best splits that decrease the empty space over a set of (linearly moving) spatiotemporal objects. Clearly, as the number of splits increases, a more accurate representation of the spatiotemporal objects is achieved and thus the empty space is reduced. On the one extreme is the case when splitting occurs for every spatiotemporal object. However, this creates high storage overhead. A more realistic assumption is to put an upper limit on the number of splits. Then the challenge is to find which spatiotemporal objects to split and where to split them. More formally we consider the following problem, also termed the Minimization of Empty Space (MES) problem:

> **Problem Statement.** Given a set of spatiotemporal objects G and an upper limit on the number of splits k, find the optimal way to apply these splits so as to minimize the empty space.

The gain function below measures the gain in empty space after k splits.

Definition 4 *Let G be a set of spatiotemporal objects. Function* **gain:** $G \times \mathcal{N} \to \mathcal{R}$, *takes as input a spatiotemporal object O and an integer k and returns the following real value:*

$$gain(O,k) = Empty(O) - \sum_{1 \le i \le k+1} Empty(O_i)$$

where O_i are all the objects generated after applying the operation split-k(O).

For example, in the 1-dimensional case that is shown in Figure 7, we have $gain(O,1) = E_1 + E_2$.

We show that a special *monotonicity* property holds when objects move linearly over time. This property is used to prove the correctness of our splitting algorithm.

Lemma 2 *Let O be a spatiotemporal object created by a linear movement. Then, $f(k) = gain(O,k) - gain(O,k-1)$ for each O and $k \ge 1$ is a monotonically decreasing function of k.*

Proof. The position alteration is described by an equation of the form: $f(\bar{t}) = \alpha\bar{t} + \beta$. First we provide formulas for the gain function and then show that the monotonicity property holds. Consider first the case where objects move or change their extent linearly on a 1-dimensional environment. An example is an interval that moves linearly over time on a line. The gain obtained by splitting k times such a spatiotemporal object O is given by the equation:

$$gain(O,k) = \frac{k}{k+1} Empty(O)$$

where $Empty(O)$ is the empty space introduced by approximating the original spatiotemporal object with an MBR.

Figure **??** depicts an 1-dimensional object O that is first split once and then twice. With one split, the best split position is at the middle of the horizontal side of the original spatiotemporal object. The gain in empty space is $gain(O,1) = \frac{1}{2}E_1 + \frac{1}{2}E_2 = \frac{1}{2}Empty(O)$. With two splits, the best split positions are in the first third and the second third of the horizontal side. Now $gain(O,2) = \frac{2}{3}Empty(O)$. (Note that the above equation holds also for 1-dimensional objects that linearly change extent, or move and change extent.)

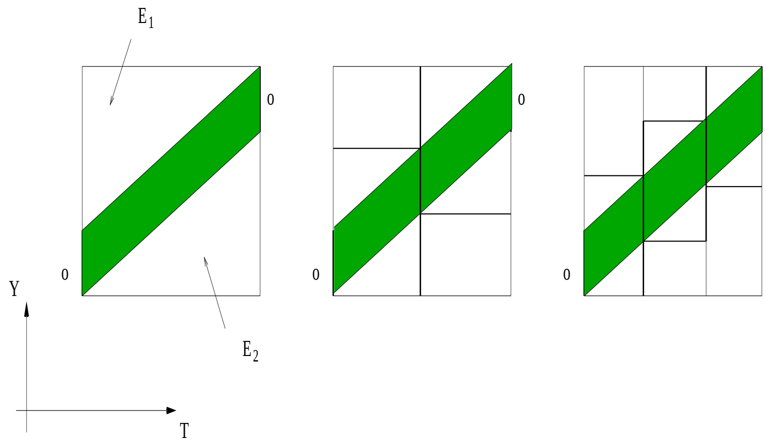

Figure 8: A 1-dimensional moving object after one and two splits.

The gain formula for a 2-dimensional moving object depends on whether the object has extent. For the case of a point moving linearly, the gain obtained after k splits is:

$$gain(O,k) = \frac{(k+1)^2 - 1}{(k+1)^2} Empty(O)$$

Assume that the moving point has initial position (x_1, y_1, t_1) and final position (x_2, y_2, t_2), where $x_1 \neq x_2$, $y_1 \neq y_2$ and $t_1 \neq t_2$. Then the MBR has volume $V = abc = (x_2 - x_1)(y_2 - y_1)(t_2 - t_1)$ which is equal to the empty space, since the moving point does not have extent (see Figure 9). After k splits, we get $k+1$ spatiotemporal objects, approximated with $k+1$ MBRs. Since we split in equidistant points, each rectangle (MBR) has sides $\frac{a}{k+1}$, $\frac{b}{k+1}$ and $\frac{c}{k+1}$. The total volume for these rectangles is:

$$V_{splits} = (k+1)\frac{a}{k+1}\frac{b}{k+1}\frac{c}{k+1}$$

and finally the gain in empty space from the k splits is:

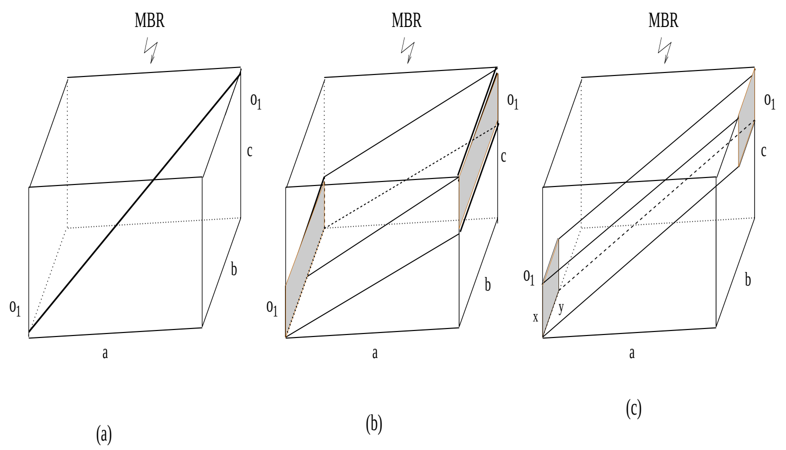

Figure 9: Three cases for 2-dimensional moving objects, (a) point, (b) moving rectangle with the same starting and ending x-coordinates and (c) moving rectangle with different starting and ending x and y coordinates.

$$gain(O,k) = V - V_{splits} = \frac{(k+1)^2 - 1}{(k+1)^2} abc$$

and this is equal to the previous equation.

An object with extent is represented by its 2-dimensional MBR. Hence consider a rectangle object that moved from some initial position to a final one. The position of this rectangle is defined by the position of its center. If the initial and final positions have one common coordinate (x or y), the gain is described by a similar formula as in the 1-dimensional space. Note however that the empty space in the 1-dimensional case refers to an area while in two dimensions it refers to a volume.

If the initial and final positions have different x and y coordinates (see Figure 9), the gain formula involves also the spatial extent of the object. Using the same arguments as for point objects it can be shown that:

$$gain(O,k) = \frac{(k+1)^2 - 1}{(k+1)^2} abc - \frac{k}{(k+1)^2}(aby + acx) - \frac{k^2}{(k+1)^2} axy$$

Using the above gain functions it is easy to prove that $f(k) = gain(O,k) - gain(O,k-1)$ for each O and $k \geq 1$ is a monotonically decreasing function of k, i.e.,

$$\frac{df(k)}{dk} \leq 0.$$

□

To indicate that the above property does not hold for spatiotemporal objects created by non-linear movement functions, consider the example in Figure 10. Here two splits on a 1-dimensional moving object provide a gain (shown as a shaded area) that is larger than the gain with one split. Similar examples exist for 2-dimensional objects.

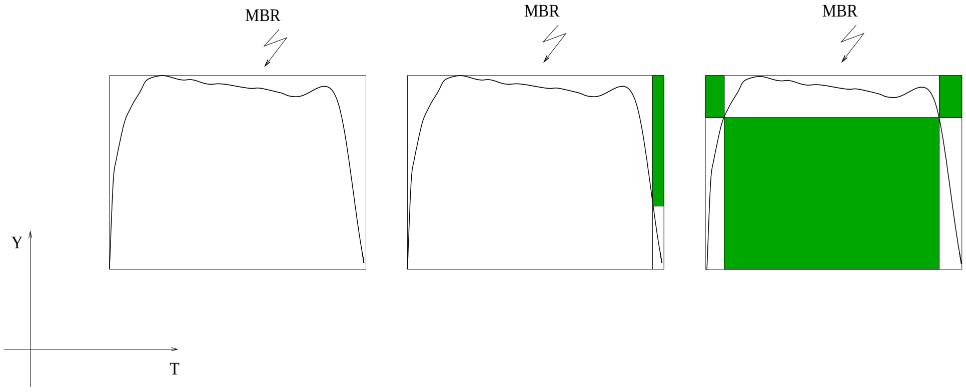

Figure 10: The spatiotemporal object created by a 1-dimensional moving point and the gain after performing one and two splits.

The monotonicity property simply states that after some point, the more we split a spatiotemporal object the less gain we get, in terms of empty space. So the first few splits will give higher gain in empty space.

The MES problem minimizes the empty space in the 3-dimensional space. However by minimizing this empty space, we also minimize the total empty space for the PPR-Tree. Empty space in the PPR-Tree is introduced due to approximating a moving object with the 2-dimensional rectangle that encloses the object for all time instants during its lifetime (maxMBR). Introducing the artificial splits enables the PPR-Tree to better approximate an object's evolution. Hence its query performance is expected to improve.

On the other hand, the 3D R-Tree will not be significantly affected by the splits. To justify this, we use the results presented in [43]. In this paper, the authors give an analytical model to approximate the number of pages accessed in an R-Tree, given a range query. This number is proportional to the number of indexed objects and also proportional to the density of the dataset. In particular, they give the following equation for the number of data pages accessed for a 3-dimensional dataset of m hyper-rectangles.

$$DA(q) = \frac{m}{f}((\frac{D_1 f}{m})^{1/3} + q_x)((\frac{D_1 f}{m})^{1/3} + q_y)((\frac{D_1 f}{m})^{1/3} + q_z)$$

and

$$D_1 = (1 + \frac{D^{1/3} - 1}{f^{1/3}})^3$$

where f is the capacity of each node in the tree, and $q = (q_x, q_y, q_z)$ is a range query. Also D is the density of the data objects and is defined as the average number of objects that contain a given point in the data space. These equations show that split operations will not necessarily decrease the query overhead. While a split operation decreases the density of the dataset (D), at the same time increases the number of indexed objects (m).

3.1 An Optimal Greedy Algorithm

Here we introduce an optimal greedy algorithm for the MES problem with linearly moving objects. We also discuss possible implementation methods of the algorithm.

Figure 11 depicts the algorithm. We use the notation Q_i to denote a vector of size N (the number of spatiotemporal objects created by the linear movements). Each position in this vector corresponds to an object and stores the number of splits for the associated object in the optimal solution. We initiate this vector with the N dimensional zero vector $\bar{0} = (0, ..., 0)$. We define vector e_j to have zero values in all positions except j, where the value is equal to one (1). Now, we find the optimal solution for one, two, ..., up to K splits. The basic idea is that the optimal solution for i splits can be derived from the solution for $i-1$ splits, if we choose to split some object one more time. Thus we choose from all possible objects, the one that gives the higher gain in empty space.

A naive implementation of this algorithm will have complexity $O(KN)$ operations in main memory. Note that to find the object that gives the optimal solution with one more split, one needs to check only the objects that give the maximum gain. Hence, the objects can be stored in a priority queue, sorted by the gain obtained if each object is split once more. Then at each step the object that gives the highest gain is chosen. Suppose that at some point object o_j is chosen to be split and assume this object has already l splits. Then the algorithm computes the difference between the gain obtained by splitting the object using $l+1$ splits ($gain(o_j, l+1)$) and its current gain. That is, the object is inserted in the queue with value $gain(o_j, l+1) - gain(o_j, l)$.

Given a set G of N (linear) spatiotemporal objects and a value for the input parameter K (an upper limit to the number of splits),

1. Set $Q_0 = \bar{0}$. Compute for each object the gain obtained with one split and insert it to a priority queue.
2. Repeat for $i = 1$ to K
   - (a) Get the top element of the queue, delete this object and let this be object j.
   - (b) Set $Q_i = Q_{i-1} + e_j$
   - (c) Compute the difference in gain obtained if object j is split one more time and insert the object into the queue using this value.

Figure 11: The Optimal GREEDY algorithm

Next we state and prove the following theorem:

Theorem 1 *There is an algorithm that solves the MES problem for linearly moving objects optimally. This algorithm can be implemented in main memory with complexity* $O(N+$

$KlogN$) and in external memory with $O(\frac{N}{B}log_{\frac{M}{B}}\frac{N}{B})$ I/O's, where M is the size of main memory in records.

Proof.

First we prove that indeed the GREEDY algorithm finds the optimal solution. Let Q_k be the vector that stores the optimal solution for the MES problem of N objects with k splits. That is, the solution that minimizes the empty space by using k splits. We then derive the solution for $k+1$ splits.

Let $Q_k = \{k_1, k_2, ..., k_N\}$, where $k_i, i = 1, .., N$ are the number of splits for each object. Thus the first object has to be split k_1 times, the second object k_2 times and so on. We also know that $\sum_{i=1}^{N} k_i = k$. We claim that the optimal solution for $k+1$ splits has the form $Q_{k+1} = \{k_1, ..., k_i + 1, ..., k_N\}$ for some $i \in \{1, ..., N\}$.

Lets assume that this is not true and that the optimal solution for $k+1$ splits has the form: $Q_{k+1}\prime = \{k_1, ..., k_i + 2, ..., k_j - 1 ..., k_N\}$ for some i and j.

Since Q_k is the optimal solution for k splits, we have that:

$$gain(o_i, k_i + 1) + gain(o_j, k_j - 1) \leq gain(o_i, k_i) + gain(o_j, k_j)$$

$$\Rightarrow gain(o_i, k_i + 1) - gain(o_i, k_i) \leq gain(o_j, k_j) - gain(o_j, k_j - 1)$$

Also by Lemma 2 it holds that:

$$gain(o_i, k_i + 2) - gain(o_i, k_i + 1) \leq gain(o_i, k_i + 1) - gain(o_i, k_i)$$

Therefore,

$$gain(o_i, k_i + 2) - gain(o_i, k_i + 1) \leq gain(o_j, k_j) - gain(o_j, k_j - 1)$$

$$\Rightarrow gain(o_i, k_i + 2) + gain(o_j, k_j - 1) \leq gain(o_i, k_i + 1) + gain(o_j, k_j)$$

The last inequality implies that Q_{k+1} is an optimal solution since we can split object o_i $k+1$ times and object o_j k_j times and have a better solution (or at least the same) with a solution of the form $Q_{k+1}\prime$. The same can be shown for any other solution with $k+1$ splits.

Thus, the optimal solution for $k+1$ splits can be derived by the optimal solution with k splits. The algorithm in Figure 11 does exactly that.

To implement the greedy algorithm efficiently we need to implement a priority queue. For this queue we use a heap. The creation time of this heap is $O(N)$ for N objects [11]. Then each insertion or deletion takes $O(logN)$ operations and the running time of the algorithm is $O(N + KlogN)$. Under the assumption that $K = o(N)$, the running time of the algorithm becomes $O(NlogN)$.

Since for the applications we have in mind the number of spatiotemporal objects is large and cannot be kept in main memory, an external memory priority queue is needed. We propose using an implementation of an external memory priority queue that is based on the buffer tree [3]. The basic idea is to perform operations (insertions and deletions) off-line in such a way that the amortized complexity of each operation is $O(\frac{1}{B} log_{\frac{M}{B}} \frac{N}{B})$ [4]. As a result the running time of the algorithm in external memory becomes $O(\frac{N}{B} log_{\frac{M}{B}} \frac{N}{B})$ I/O's. □

Note that the above proof works similarly for the case where objects do not move but change only one of their extent dimensions linearly.

4 Performance Evaluation

In section 4.1 we describe the datasets and outline the workloads used in our experimental evaluation. A discussion regarding how to choose the number of artificial splits for the GREEDY algorithm is presented in section 4.2. Section 4.3 discusses various optimization methods for tuning the performance of the PPR-Tree. Finally, we present experimental results for both types of object evolutions, namely the degenerate case (section 4.4) and the general case (section 4.5.)

4.1 Experimental Setup

For all the methods used the page size was set to 1 kbytes and the maximum number of records per page was equal to 50 (B=50). For the insertion and query operations a buffer of 10 pages was used with an LRU replacement policy. For all methods, during the query phase the buffer is invalidated before a new query gets executed (so that strengths and weaknesses of the particular implementations are revealed). For the 3D R-Tree method, we used an implementation of the R*-tree [6]. We also implemented a Skeleton SR-Tree based on the description in [23]. Our implementation of the Skeleton SR-Tree allows index nodes to have varying page sizes (starting from the leaf nodes the page size doubles as the level reaches the root). For a given index page, one third of the page is allocated for storing spanning segments, while the rest is used to store index records. Overflow segments still appeared in higher level nodes; such segments were stored in additional pages. However, the reported query times for the Skeleton SR-Tree do not include accessing these pages (i.e., the reported SR-Tree query times are underestimates of the actual ones). Finally, we also experimented with the packed STR-Tree [25]. Our method is Off-Line and since all data are available at index creation, we would expect that clustering the data first and building the index bottom-up would yield better results.

We generated various spatiotemporal datasets to compare the performance of the different methods. The datasets for the degenerate case are similar to the spatiotemporal datasets described in [48]. Objects in a given frame are approximated by their 2-dimensional MBRs and the size of the frame is 1.0×1.0 (unit square). Moreover, 70% of the objects are small rectangles with small lifetimes. The length of each rectangle in the x and y axes is uniformly chosen from the interval (0, 0.04] and the centers of the rectangles are uniformly distributed in the unit square. The lifetime of each object follows a Poisson distribution with mean value equal to 50. Another 15% of the objects are large rectangles with small lifetimes. Here the length of each rectangle in spatial dimensions is uniformly chosen from (0, 0.6] and the lifetimes are the same as above. The remaining 15% of the objects are small rectangles with large lifetimes. The lifetimes for these objects are uniformly chosen between 250 and 500 frames. Each object may appear and disappear a number of times, which is randomly chosen between 1 and 10000. The number of intermediate frames between subsequent lifetimes is once more uniformly chosen between 250 and 500. We generated five different datasets with objects per frame ranging from 250 to 2500. We call this type of datasets DG (degenerate).

For the general case, we created a collection of datasets containing only moving rectangles (the MV dataset). Each rectangle starts at a specific position and moves with a linear motion to its final destination. Each set has one-third of "slowly" moving rectangles whose sides are uniformly chosen from $(0, 0.02]$, and speeds between 0 and 0.001. One-third has sides in $(0, 0.01]$ and speeds between 0 and 0.006 and finally "fast" objects with the same side lengths and speeds between 0 and 0.01. The rectangles retain their size as they move and only their center position changes. The lifetime of each object has mean value 50. Again, the average number of objects per frame ranges from 500 to 2500.

We also generated a collection of datasets that is a mixture of the previous ones (the GN or generic collection), and consists of static objects, moving objects and objects that change extent over their lifetime. In particular, one-third of the objects are static objects with the characteristics of the DG datasets. One-third are moving objects and the rest are objects that change position and extent, always linearly over the frame sequence. To generate some of the above datasets we used the GSTD generator [45]. In Table 1, we give the main characteristics of the datasets.

Table 1: The datasets used for testing the index structures.

Dataset	Avg Number of Objects per Frame	Total Number of Spatiotemporal Objects
DG	500, 1000, 1500, 2000, 2500	86807, 173925, 260933, 348006, 435056
MV	500, 1000, 1500, 2000, 2500	74017, 147996, 222096, 296012, 369858
GN	500, 1000, 1500, 2000, 2500	74346, 148597, 222970, 297272, 371598

Finally, query workloads were generated for range and nearest neighbor queries. A query workload consists of 1,000 queries. A query is spatiotemporal in nature, i.e., it has a spatial and a temporal predicate. For the range queries, the spatial part contained 2D rectangles with three different sizes, Small, Medium and Large. The Small rectangles had lengths between 0 and 0.1, Medium between 0.1 and 0.3 and Large between 0.2 and 0.6. For the temporal predicate, we distinguished between "snapshot" queries, where the temporal part was a single frame, and, "period" queries where each query specified a frame interval of length between 0 and 100. For the nearest neighbor queries, the spatial part was either a query point or a small rectangle uniformly chosen inside the data space. The temporal part was a "period" selected randomly, with length between 0 and 100.

It should be noted that before inserting the data in the PPR-Tree, we sorted them over the object insertion and deletion frames. For the 3D R-Tree the dataset is first sorted on the object insertion frames and objects are inserted in that order. For the Skeleton SR-Tree inserting the spatiotemporal objects according to insertion frame order tend to affect overlapping (since the ordering implies that an interval will probably overlap the next inserted interval). We got better performance when the spatiotemporal objects were inserted randomly. Finally, the STR-Tree clusters the data in a specific way before building the tree bottom-up, thus pre-sorting the dataset has no effect on the resulting structure.

4.2 Deciding on the number of splits

Before inserting objects into the PPR-Tree we use the GREEDY algorithm to split the dataset with a given number of artificial splits. A good number of splits depends not only on the type of the dataset at hand but also on the available storage space. More splits minimize empty space, but linearly increase the number of objects. We performed a number of experiments with varying number of splits for the MV and GN datasets with 1000 objects per frame. We evaluated the query performance using snapshot queries. Figures 12 and 13 depict the results in terms of I/O per query.

In general, we expect query performance to increase as we increase the number of splits. This continues up to a point after which it stabilizes and ultimately deteriorates. The reason for the latter is that the size of the index structure will become very large, but the gain in empty space by introducing more splits, will not be comparable. A good choice for the number of artificial splits would be the point where the curve begins to stabilize. Obviously, there is a trade-off between storage and query performance, so the final choice of the number of artificial splits depends on the cost of extra storage per split. Judging from the results of our experiments we decided to use a number of splits equal to 50% of the total number N of objects contained in each dataset, meaning that the final number of objects produced was equal to $1.5N$.

We tested the above datasets with the varying number of splits against the 3D R-Tree, the STR and the SR-Tree, too. As expected, splitting did not improve the query performance of the non persistent indices.

4.3 Tuning the PPR-Tree

A number of optimization issues have to be addressed when implementing the PPR-Tree. The most important of them are the merging and splitting policies. When an underutilized page in the PPR-Tree needs to be merged there may be many possible sibling pages for merging. We used three merging policies. The first one, called *Overlap* chooses as a sibling the currently alive page that has the same parent and shares the most overlap with the underutilized page. The second one, (*Min_Area*), selects as sibling the page whose bounding rectangle area needs the least geometric expansion to incorporate the objects of the underutilized page. Finally, the third policy (*Margin*), finds the page that when merged with the underutilized page, has the least margin. The latter is defined as the sum of the lengths of all sides of the bounding rectangle.

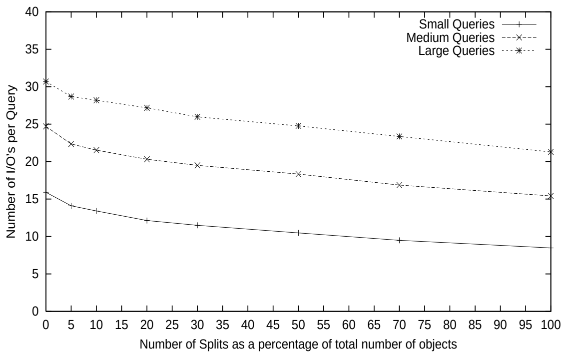

Figure 12: Query performance of the Greedy-PPR-Tree for snapshot queries and different number of splits and MV datasets.

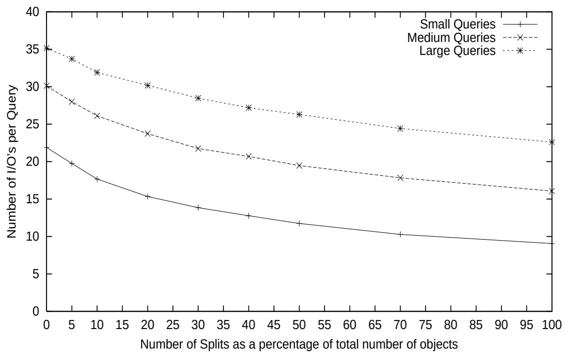

Figure 13: Query performance of the Greedy-PPR-Tree for snapshot queries and different number of splits and GN datasets.

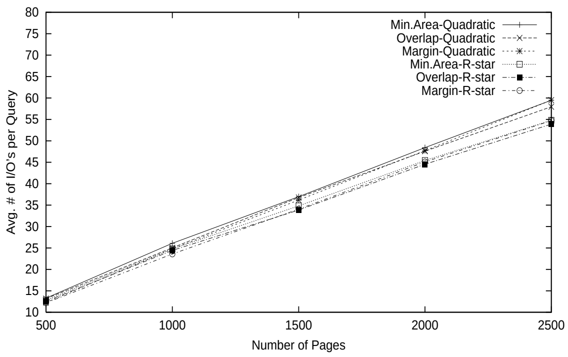

Figure 14: Query performance for snapshot queries and different merging/splitting policies.

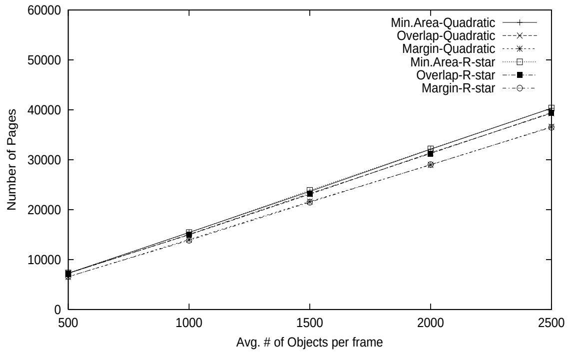

Figure 15: Storage consumption for different merging/splitting policies.

For splitting, we use two methods. The first is called *Quadratic* and it has been proposed in the original R-Tree paper [16]. The second (*R-star*) is the policy used by the R*-tree [6]. The first policy assigns objects in two groups, initializing these groups by picking the pair of objects that would waste the most area if inserted in the same group. The R-star policy is based on determining various distributions of objects in a page, after ordering all objects in each dimension. The best distribution is selected, based on a set of criteria, such as minimizing the sum of margin values and also minimizing the overlap-area between the two generated pages.

Figure 14 plots the query performance (in average number of pages read per query) for all combinations of splitting and merging methods. We used the DG datasets and a snapshot query workload. As the figure shows, the query performance is mainly affected by the splitting policy (with the R-star policy providing better results than Quadratic). The merging policy has small effect. The storage consumption of the PPR-Tree is depicted on Figure 15. Here the important factor is the merging policy and the Margin policy gives the best results. For the rest of our experiments, we implemented the PPR-Tree using the R-star splitting policy and the Margin policy for merging nodes.

4.4 Degenerate Case

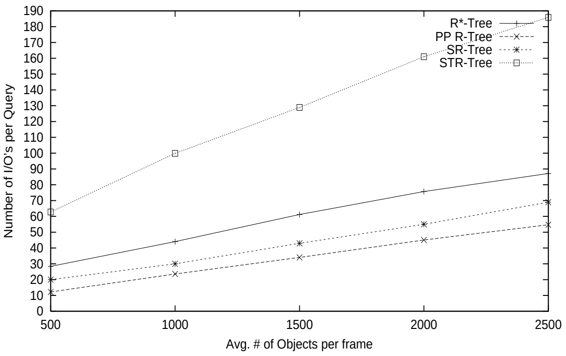

Figure 16: Query performance for small/ snapshot queries and DG datasets.

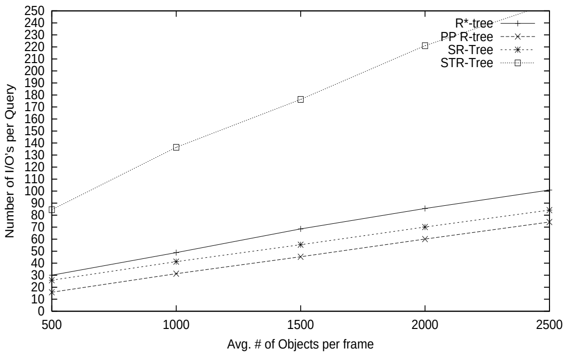

Figure 17: Query performance for medium/ snapshot queries and DG datasets.

We proceed with experimental results about the degenerate case. Since it contains objects with no position/extent changes it serves as a reference point for our later experiments. Figures 16-18 report the results for snapshot queries with Small, Medium and Large size (in spatial extent) respectively. The average lifetime of the objects is about 50 frames. In all cases the partially persistence methodology outperforms the Skeleton SR-Tree, the 3D R-Tree and the STR-Tree. The difference is higher for smaller queries. The SR-Tree behaves better than both 3D R-Tree and STR-Tree since by placing spatiotemporal objects with long lifetimes higher in the hierarchy reduces overlapping. It should be noted that in our SR-Tree implementation, the experiments with average number of objects/frame equal to 2,000 and 2,500, produced comparatively very large number of overflow pages. Since these pages were not counted for the query I/O's, the depicted performance corresponds to interpolation from the behavior of the method for the 500, 1,000 and 1,500 experiments. The performance of the STR-Tree deteriorates with the increase in size of the dataset.

Figure 19 shows the results for Small/period queries with query frame period ranging from 0 to 100, and a dataset with 1,000 objects/frame. Interestingly, the R-Tree behaves better than the SR-Tree for period queries. This is due to object fragmentation, since the larger the query period, the more object copies this period will overlap with. The PPR-Tree's performance is also affected by the query period size. Since partial persistence is optimized towards frame queries, a query involving a large period (many subsequent frames) will overlap with many object copies thus decreasing query performance.

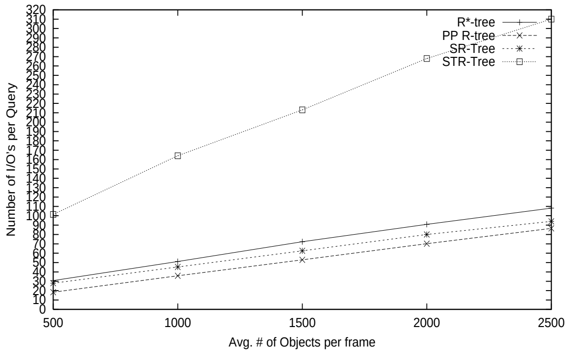

Figure 18: Query performance for large/ snapshot queries and DG dataset.

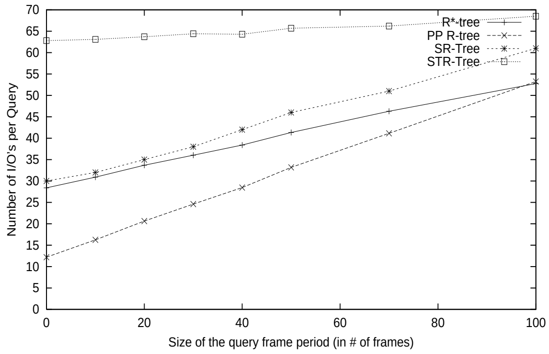

Figure 19: Query performance for frame period queries and DG dataset.

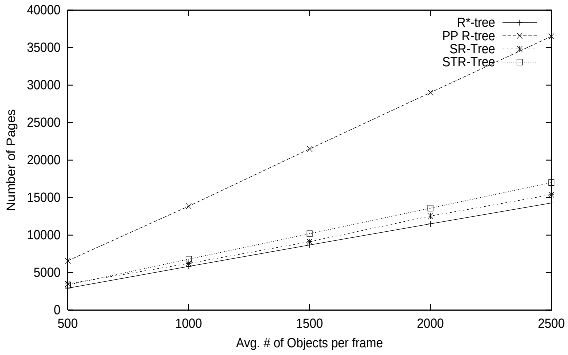

Figure 20: Storage consumption for DG datasets.

Figure 20 depicts the storage consumption of all methods, for DG datasets. As expected storage for the PPR-Tree is higher than that of the SR-Tree, 3D R-Tree and STR-Tree (but it remains linear to the number of objects). The STR-Tree has the smallest storage requirements, since packing eliminates empty records in data and index pages. However, the query performance of the STR-Tree was clearly worse than that of the other methods. The reason is that packing may cluster together objects that are consecutive in order even though they may correspond to small and large intervals. This leads to more overlapping and empty space. We observed similar behavior with all experiments (in the degenerate as well as the general case), hence for brevity we omit the STR-Tree from the latter figures.

4.5 General Case

First we present our results for the moving rectangles datasets (MV) and then for the general datasets (GN). Given a dataset, the GREEDY algorithm derives first all spatiotemporal objects that yield the best gain in terms of empty space when split. Then these objects are split and the MBRs of the newly generated spatiotemporal objects are computed. Subsequently these MBRs are indexed by the PPR-Tree (marked as Greedy-PPR-Tree in the figures). To validate the expectation that a 3D R-Tree will not gain much by the artificial splits of the GREEDY algorithm, we indexed the resulting MBRs with a 3D R-Tree too (Greedy-3D R-Tree). We compare the two GREEDY approaches against the approach where no artificial split is considered. That is, we used a 3-dimensional MBR around each spatiotemporal object and indexed them using (i) a plain 3D R-Tree and (ii) a Skeleton SR-Tree. Finally, we used the $maxMBR$ approach for the PPR-Tree (maxMBR-PPR-Tree.)

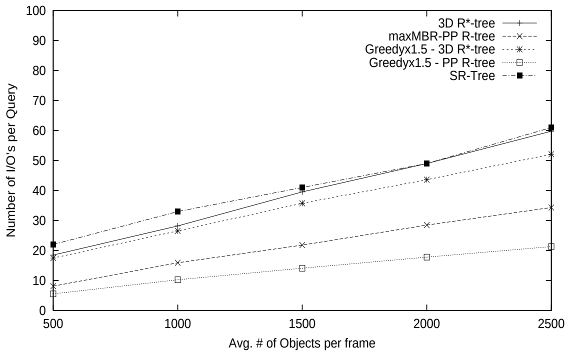

Figure 21: Query performance for small/ snapshot queries and MV datasets.

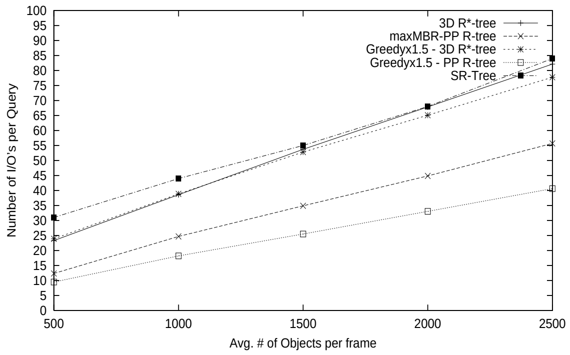

Figure 22: Query performance for medium/ snapshot queries and MV datasets.

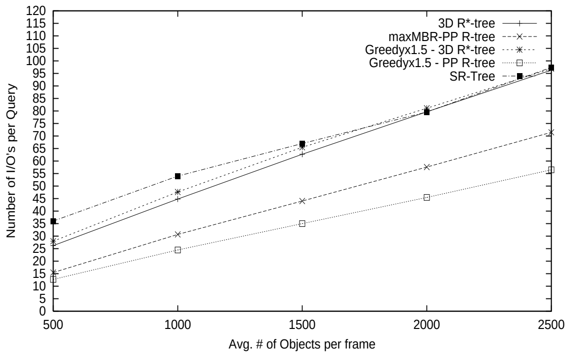

Figure 23: Query performance for large/ snapshot queries and MV datasets.

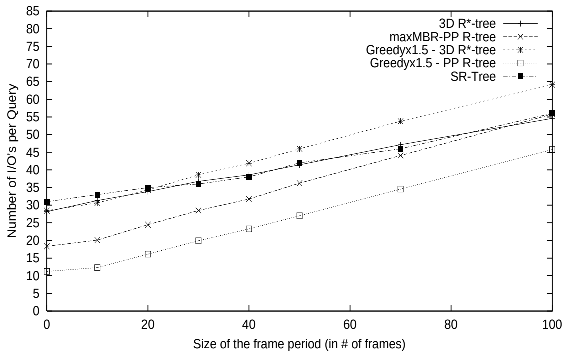

Figure 24: Query performance for frame period queries and MV datasets.

Figures 21-23 depict the results for snapshot queries and MV datasets. The greedy algorithm combined with the PPR-Tree provides the best query performance. The second best is the PPR-Tree that follows the maxMBR approach. It is interesting to note that the 3D R-Tree performs similarly with splits or no splits (i.e., as expected, the greedy splits do not

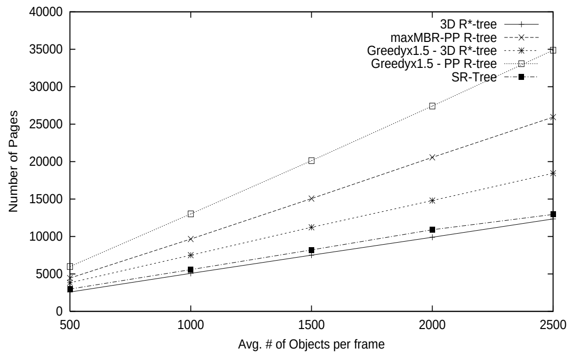

Figure 25: Storage consumption for MV datasets.

provide a large advantage). A split may decrease the empty space but it increases the number of objects, affecting the 3D R-Tree query performance. The Skeleton SR-Tree behaves worse that the 3D R-Tree for the MV datasets. Since objects move, the corresponding MBR is rather large, not only on the frame dimension, but on the X and Y dimensions as well. The SR-Tree clustering based on the lifetimes is not so efficient anymore, and the method tends to perform like a regular R-Tree. Frame period queries appear in Figure 24 using a dataset with 1000 objects per frame. The Greedy-PPR-Tree method remains better than the other methods even for the larger periods we tried. It is also clear that as the query period increases, the performance of the greedy 3D R-Tree deteriorates against the conventional 3D R-Tree. This is because the splits from the greedy approach introduce copies that the R-Tree considers as separate objects. Again, the SR-Tree behaves very similar to the 3D R-Tree. The storage for a method that uses the greedy approach is about 1.5 times the storage of the same non-greedy method (Figure 25). Since the behavior of the SR-Tree is very close to that of the 3D R-Tree (in both query and storage performance) for brevity we omit the SR-Tree from the following figures.

The performance comparisons for the general datasets (that include mixtures of moving/static/extending objects) appear in Figures 26 to 29. All methods behave very similar to the results for the moving objects datasets. Despite using the greedy algorithm as an approximation for the extending objects, the Greedy-PPR-Tree still provides the best performance.

The performance for nearest neighbor queries is similar to the range queries. For brevity, we report results for the general datasets (GN), but the same trend was observed for the other datasets as well. In Figure 31 the average query performance is shown for a set of 50-Nearest Neighbor queries (that is, find the 50 nearest objects to the query object). The frame period was 20 frames. Figure 32 reports results for nearest neighbor queries with different frame periods. The Greedy-PPR-Tree has again the best query performance.

Finally, in Figures 33 and 34 we present the total number of I/O's needed to create each of the index structures. Here, we assume a cache of only 10 pages. Using a larger cache the

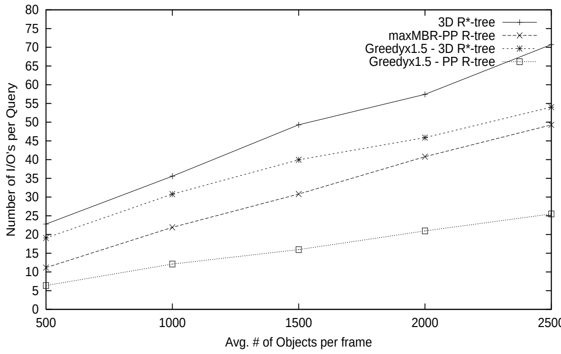

Figure 26: Query performance for small/ snapshot queries and GN datasets.

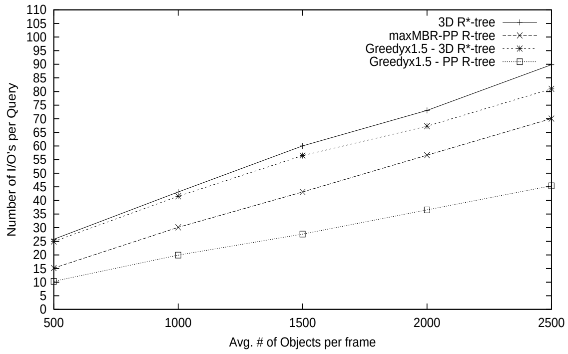

Figure 27: Query performance for medium/ snapshot queries and GN datasets.

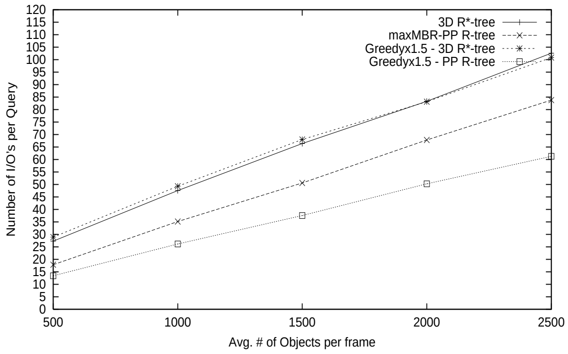

Figure 28: Query performance for large/ snapshot queries and GN datasets.

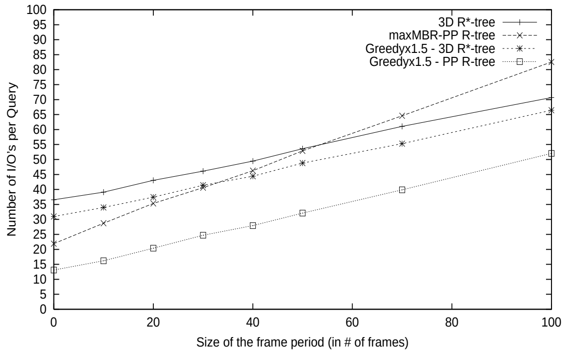

Figure 29: Query performance for frame period queries and GN datasets.

construction time can be decreased considerably. The 3D R-Trees have lower construction time than the PPR-Trees. This is not surprising. Clearly, for the partially persistent methods the index is accessed twice for each spatiotemporal object: once at the insertion frame and again at the deletion frame. On the other hand for the 3D R-Trees, the index is accessed only when the MBR of the spatiotemporal object is inserted. However, for the Off-Line problem the index is created only once and then is used for querying only.Thus, the update cost is not that critical.

5 Related Work

Although recently there has been extensive work on multimedia and video databases, the approach discussed in this paper is novel. The work in [48] considers only static objects (degenerate case) and uses a 3D R-Tree approach to index the objects. Another work that proposes indexing video objects in order to answer mostly temporal queries appears in [1].

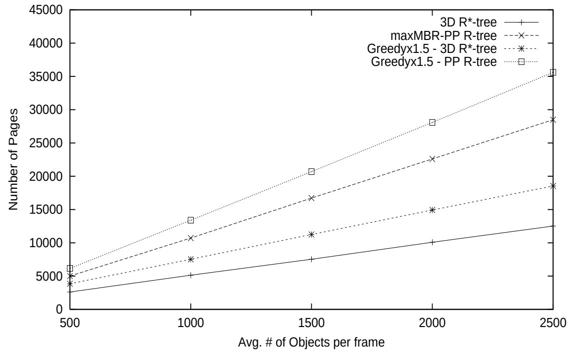

Figure 30: Storage consumption for the GN datasets.

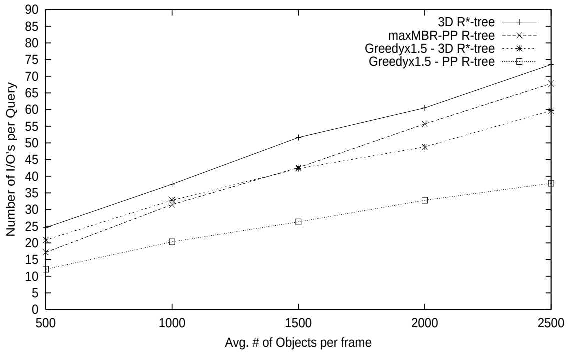

Figure 31: Nearest Neighbor query performance for GN datasets.

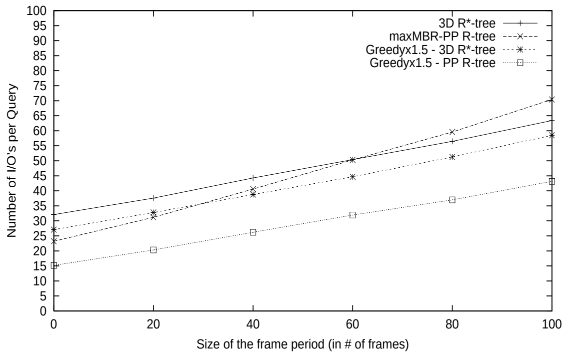

Figure 32: Nearest Neighbor query performance for different frame periods and GN datasets.

In this paper, video movies are pre-processed and all entities of interest such as objects, activities, and events, are identified. Subsequently, these entities are associated with specific frames in which they appear. Therefore, every entity is coupled with a set of frames which can be viewed as a set of line segments (if consecutive frames are put in one line segment). A main-memory Segment Tree [37] is used to store the resulting line segments. Queries that this structure can answer are of the type: "find the objects that appear when a specific event happened" or "find the objects that appear in all frames where a specific object appears". Also the authors discuss how to store higher level information for each object in order to answer more complex queries. However, most of the complex queries have query time linear to the total number of video objects.

Another interesting approach to index video data has been proposed in [8, 9]. With their approach, video data is indexed using not only information about the color or texture (as in image databases) but also motion and spatiotemporal information. First a video movie is partitioned into shots or scenes. Then all objects that appear inside each shot (called video objects) are found. For each object information about its features (color, texture and shape),

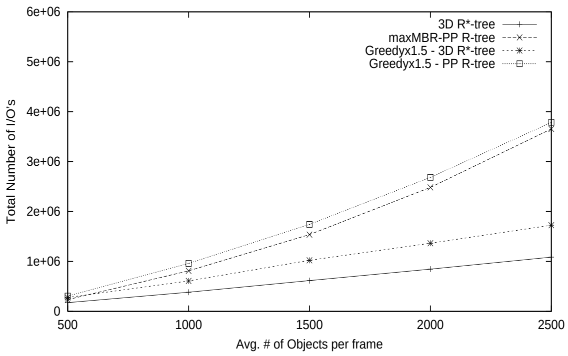

Figure 33: Construction cost for MV dataset.

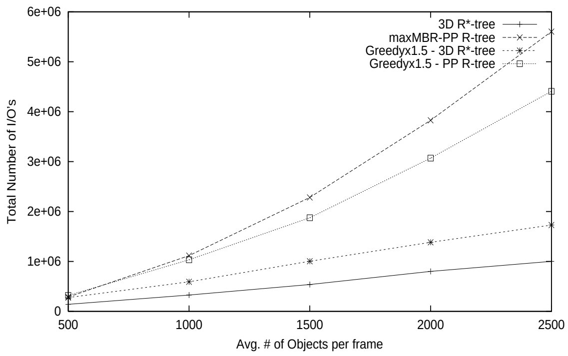

Figure 34: Construction cost for GN dataset.

but also about its motion is stored. In particular, the motion of an object is stored as a trail of the object position from one frame to another. The user can ask queries using a visual interface, and can give different weights for each feature. In [42], algorithms to index these video objects are presented. Each object is mapped to a high dimensional space which is then split into a few low dimensional feature vectors. Querying is performed for each vector separately. Yet another work that represents the motion of an object by using its trail is [12]. Our approach is complementary to these works and can thus be used to enhance the query capabilities of the aforementioned systems.

Content-based retrieval has also been an active research area in the past few years and several systems have been developed. These systems allow image indexing by using low-level image features such as color histograms, texture and shape. The user specifies a target image (QBE) or a sketch and the system retrieves the most similar images to the target image. Some examples of very successful systems in this area include QBIC [15], Virage [17] and VisualSEEk [39]. However, all these systems support retrieval of still images. Some of these ideas have been used to index movie databases by using low level features combined with some semantic information [40, 8].

Research in the area of spatiotemporal database indexing is also quite related to our work. In particular, [44] summarizes the issues that a spatiotemporal index needs to address. In an early paper [50], the RT-tree is presented, an R-Tree that incorporates time into its nodes. Each object has a spatial and a temporal extent. For an object that is entered at time t_i the temporal extent is initialized to $[t_i, t_i)$. This temporal extent is updated (increased) every time instant that the spatial extent remains unchanged. If the spatial extent changes at time t_j, a new record is created for this object with a new temporal extent $[t_i, t_j)$. Clearly, this method is inefficient due to its large update overhead. In [28, 47, 50, 29] the idea of overlapping trees is used to make an index partially persistent. Different indices are created for each time instant, but to save storage, common paths are maintained only once since they are shared among the structure. However, the overlapping method has a logarithmic storage overhead, since every time an update is made, the whole path from the root to the updated leaf node has to be copied. Indeed, in an experimental evaluation presented in [29] the overlapping R-Tree (HR-Tree) has an order of magnitude higher storage overhead than the 3D R-Tree. It should be noted that the GREEDY algorithm presented in this paper is general and can be used to enhance the performance of any partially persistent method (including the overlapping approach). In another recent work [34], an R-Tree is extended to support transaction and valid time. However, this work concentrates on the combination of degenerate evolutions and bitemporal datasets. Spatiotemporal indexing as examined here deals with historical queries about the spatiotemporal evolutions. Work dealing with future queries about the position of moving objects (assuming knowledge of movement functions) appears in [35, 22, 2].

6 Conclusions and Further Research

We have examined the problem of indexing objects in animated movies. We proposed to represent a movie as a spatiotemporal evolution and reduce the original problem to a problem of partial persistence. However, the partial persistence approach considers only objects that remain unchanged during their evolution (i.e., between the frames they appear). This is not realistic in animated movies where objects can change their extent/position among frames. We presented an efficient way to represent such complex objects. In particular, we formulated this problem as an optimization problem and provided an optimal greedy algorithm for the case of linearly moving objects. Our solution is also optimal for objects that change linearly only one of their extent dimensions. However, it is suboptimal for objects that change both their extent dimensions. The presented approach provides very fast query time at the expense of some extra storage, which however is linear to the number of changes in the frame evolution. We have shown the merit of our method by comparing it with an approach that sees the frame sequence as simply another dimension and uses (i) a regular 3D R-Tree, (ii) a Skeleton Segment R-Tree, (iii) an STR-Tree.

An interesting future direction is to consider objects that change position and/or extent with non-linear functions. Clearly, for this case the monotonicity property does not hold. We are examining the existence of efficient algorithms that approximate the optimal solution with a good approximation ratio.

Another problem that we plan to investigate is the case of On-Line indexing. This paper considered only the Off-Line case, where all objects and their evolution is known beforehand. However, in many real life applications, objects are inserted in an on-line fashion in the dataset. We expect that an on-line version of the optimal greedy algorithm will give a good approximation of the optimal solution.

Yet another interesting avenue of research is to extend the techniques presented here to different query scenarios. This includes queries where the view point changes in time. One application we can consider is the following: assume that the original two-dimensional model that we use to build an animated movie extends further than the screen, and that the actual animated movie that we see is in fact a specific cut. The cut (that is, the visible part of the movie) depends on where we position the screen window. Assuming that this position remains constant, we can find all visible objects by answering a three-dimensional range query. That is, a two-dimensional range query in the visible screen is translated into a three-dimensional spatiotemporal query. If however the position of the screen does not remain constant, the shape of the spatiotemporal query becomes more complicated. Consider for example answering range queries while the screen zooms in or out. Assuming that the size of the range query on the screen remains constant relative to the size of the screen, if we are zooming-in objects will appear larger and fewer objects will be in the query area. This query can be mapped to a spatiotemporal query that looks like a pyramid. We can approximate this query by a number of spatiotemporal range queries using the same technique that we use to optimally bound a moving object with minimum bounding rectangles. When zooming, the viewpoint changes location along an axis perpendicular to the frame plane. A more involved problem is answering such range queries when the viewpoint is translated as well as moving closer or further from the frame, or, if we consider three-dimensional objects, when the view point moves and rotates in space.

Finally, we note that our approach is general and can be applied to other spatiotemporal applications as well (for example indexing forest extends or city boundaries over time, etc.).

7 Acknowledgment

This research has been supported by NSF grants IIS-9509527, IIS-9907477 and by the Department of Defense. The authors would like to thank Elias Koutsoupias for many helpful discussions and Bernhard Seeger for providing us with the R*-tree code.

References

[1] S. Adali, K. Candan, S. Chen, K. Erol, V. S. Subrahmanian. The Advanced Video Information System: Data Structures and Query Processing. In *ACM Multimedia Systems*, 4 (4):172-186,1996.

[2] P.K. Agarwal, L. Arge and J. Erickson. Indexing Moving Points. In *Proc. of 19th ACM-PODS*, Dallas, Texas, 2000.

[3] L. Arge. The Buffer Tree: A New Technique for Optimal I/O Algorithms. In *Proc. Workshop on Algorithms and Data Structures,* LCNS 955, pages 334-345, 1995.

[4] L. Arge. External-Memory Algorithms with Applications in Geographic Information Systems. In *Algorithmic Foundations of Geographic Information Systems,* LNCS 1340, 1997.

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

[6] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: An Efficient and Robust Access Method For Points and Rectangles. In *Proc. ACM-SIGMOD International Conference on Management of Data*, pages 322–331, Atlantic City, May 1990.

[7] J.L. Bentley. Algorithms for Klee's Rectangle Problems. *Technical Report*, Computer Science Department, Carnegie-Mellon University, Pittsburgh, 1977.

[8] S.F. Chang, W. Chen, H. Meng, H. Sundaram, D. Zhong. VideoQ- An Automatic Content-Based Video Search System Using Visual Cues. In *Proc. of the 5th ACM Multimedia Conference,* pp. 313-324, 1997.

[9] S.F. Chang, W. Chen, H. Meng, H. Sundaram, D. Zhong. A Fully Automated Content Based Video Search Engine Supporting Spatio-Temporal Queries. In *IEEE Trans. on Circuits and Systems for Video Technology,* 8(5), pp. 602-615, 1998.

[10] K. L. Cheung and A. Wai-Chee Fu. Enhanced Nearest Neighbor Search on the R-Tree. In *SIGMOD Record*, 27(3): 16-21, 1998.

[11] T. Cormen, C. Leiserson and R. Rivest. Introduction to Algorithms. *The MIT Press,* Cambridge, Mass., 1990.

[12] S. Dagtas, W. Al-Khatib, A. Ghafoor, A. Khokhar. Trail-Based Approach for Video Data Indexing and Retrieval. In *Proc. IEEE ICMCS,* pp. 235-239, 1999.

[13] V. Delis, D. Papadias, N. Mamoulis. Assessing Multimedia Similarity: A Framework for Structure and Motion. In *Proc. ACM Multimedia*, pp. 333-338, 1998.

[14] J. Driscoll, N. Sarnak, D. Sleator and R.E. Tarjan. Making Data Structures Persistent. In *Proc. of the Eighteenth Annual ACM Symposium on Theory of Computing*, Berkeley, California, 1986.

[15] C. Faloutsos, R. Barber, M. Flickner, J. Hafner, W. Niblack, D. Petkovic, W. Equitz. Efficient and Effective Querying by Image Content. In *JIIS*, 3 (3/4), 231-262, 1994.

[16] A. Guttman. R-Trees: A Dynamic Index Structure For Spatial Searching. In *Proc. ACM-SIGMOD International Conference on Management of Data*, pages 47–57, Boston, June 1984.

[17] A. Hamrapur, A. Gupta, B. Horowitz, C.F. Shu, C. Fuller, J. Bach, M. Gorkani and R. Jain. Virage Video Engine. In *Proc. SPIE*, pp 188-197, 1997.

[18] J.M. Hellerstein, E. Koutsoupias, and C. Papadimitriou. On the Analysis of Indexing Schemes. In *Proc. 16th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 249–256, Tucson, May 1997.

[19] H. Jiang and A. Elmagarmid. Spatial and Temporal Content-Based Access to Hypervideo Databases *VLDB Journal*, 7(4):226-238,1998.

[20] C.S. Jensen and R.T. Snodgrass. Temporal Data Management. In *IEEE TKDE*, 11(1): 36-44, 1999.

[21] I. Kamel and C. Faloutsos. Hilbert R-Tree: an Improved R-Tree Using Fractals. In *Proc. of the 20th VLDB Conf.*, 500-509, Chile, 1994.

[22] G. Kollios, D. Gunopulos and V.J. Tsotras. On Indexing Mobile Objects. In *Proc. of the 18th ACM-PODS*, pp. 261-272, Philadelphia, PA, 1999.

[23] C. Kolovson and M. Stonebraker. Segment Indexes: Dynamic Indexing Techniques for Multi-dimensional Interval Data. In *Proc. ACM SIGMOD Conf.*, pp 138-147, 1991.

[24] A. Kumar, V.J. Tsotras and C. Faloutsos. Designing Access Methods for Bitemporal Databases. *IEEE Transactions on Knowledge and Data Engineering*, 10(1):1-20, 1998.

[25] S.T. Leutenegger, M.A. Lopez, J.M. Edgington. STR: A Simple and Efficient Algorithm for R-Tree Packing. In *ICDE*, 1997

[26] J.Z. Li, I. Goralwalla, M.T. Ozsu and D. Szafron. Modeling Video Temporal Relationship in an Object Database Management System. In IS&T/SPIE International Symposium on Electronic Imaging: Multimedia Computing and Networking, pp 80-91, San Jose, CA, 1997.

[27] D. Lomet and B. Salzberg. Access Methods for Multiversion Data. In *Proc. ACM SIGMOD Conf.*, 315-324, 1989.

[28] M. Nascimento and J. Silva. Towards Historical R-Trees. In *Proc. ACM Symp. on Applied Computing,* pp. 235-240, 1998.

[29] M. Nascimento, J. Silva and Y. Theodoridis. Evaluation of Access Structures for Discretely Moving Points. In *Proc. of the STDBM'99,* LCNS 1678, pp. 171-188, 1999.

[30] J. A. Orenstein. Redundancy in Spatial Databases. In *Proc. of ACM SIGMOD Conference,* pp. 326-336, Washington, D.C., 1986.

[31] J. A. Orenstein. A Comparison of Spatial Query Processing Techniques for Native and Parameter Spaces. In *Proc. of ACM SIGMOD Conference,* pp. 343-352, Atlantic City, NJ, 1990.

[32] S.V. Raghavan and Satish K. Tripathi. *Networked Multimedia Systems: Concepts, Architectures, and Design.*, Prentice Hall, 1998.

[33] N. Roussopoulos, S. Kelley and F. Vincent. Nearest Neighbor Queries. In *Proc. ACM-SIGMOD Int. Conf. on Management of Data*, pages 71-79, June 1992.

[34] S. Saltenis and C. Jensen. R-Tree Based Indexing of General Spatio-Temporal Data. TimeCenter, Tech-Report, TR-45, 1999.

[35] S. Saltenis, C. Jensen, S. Leutenegger and M. Lopez. Indexing the Positions of Continuously Moving Objects. In *Proc. of the 19th ACM-SIGMOD Int. Conf. on Management of Data*, Dallas, Texas, 2000.

[36] B. Salzberg and V.J. Tsotras. A Comparison of Access Methods for Time-Evolving Data. *ACM Computing Surveys*, June 1999.

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

[38] T. Sellis, N. Roussopoulos and C. Faloutsos. The R+-Tree: A Dynamic Index for Multi-Dimensional Objects. In *Proc. 13rd International Conference on Very Large Data Bases*, pages 507-518, Brighton, England, September 1987.

[39] J. R. Smith and S.F. Chang. VisualSEEk: A Fully Automated Content-Based Image Query System. In *Proc. ACM Multimedia*, pp. 87-98, 1996.

[40] A. P. Sistla, C. T. Yu, R. Venkatasubrahmanian. Similarity Based Retrieval of Videos. In *Proc. IEEE ICDE*, pp. 181-190, 1997.

[41] V.S. Subrahmanian. *Principles of Multimedia Database Systems.*, The Morgan Kaufmann Series in Data Management Systems, Jim Gray, Series Editor, 1998.

[42] H. Sundaram and S.F. Chang. Efficient Video Sequence Retrieval in Large Repositories. In *Proc. SPIE Storage and Retrieval for Image and Video Databases,* 1999.

[43] Y. Theodoridis, T. Sellis. A Model for the Prediction of R-Tree Performance. In *Proc. of the 15th Symposium on Principles of Database Systems (PODS),* pp. 161-171, 1996.

[44] Y. Theodoridis, T. Sellis, A. Papadopoulos and Y. Manolopoulos, Specifications for Efficient Indexing in Spatiotemporal Databases. In *Proc. SSDBM*, pp. 123-132, 1998.

[45] Y. Theodoridis, J. Silva and M. Nascimento. On the Generation of Spatiotemporal Datasets. In *Proc. SSD*, pp. 147-164, 1999.

[46] V.J.Tsotras, N. Kangelaris. The Snapshot Index, an I/O-Optimal Access Method for Timeslice Queries. *Information Systems*, 20(3), 1995.

[47] T. Tzouramanis, M. Vassilakopoulos, Y. Manolopoulos. Overlapping Linear Quadtrees: A Spatio-Temporal Access Method. In *Proc. ACM-GIS,* pp. 1-7, 1998.

[48] M. Vazirgiannis, Y.Theodoridis, T.K. Sellis. Spatio-Temporal Composition and Indexing for Large Multimedia Applications. In *Multimedia Systems*, 6(4): 284-298, 1998.

[49] P.J. Varman, R.M. Verma. An Efficient Multiversion Access Structure. In *IEEE TKDE*, 9(3): 391-409, 1997.

[50] X.Xu, J. Han, W. Lu. RT-tree: An Improved R-Tree Index Structure for Spatiotemporal Databases. In *Proc. Intr. Symp. on Spatial Data Handling (SDH),* 1990.