Indexing Animated Objects

George Kollios
Polytechnic University
gkollios@milos.poly.edu

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

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

Abstract

We examine the problem of indexing objects in animated movies. We consider an animated movie as consisting of a frame sequence where each frame is described by objects in a 2-dimensional space. The queries of interest are of the form: "find the objects that were in area S between frames f_i and f_j". The straightforward approach to index such objects is to consider the frame sequence as another dimension and use a 3-dimensional access method (like an R-tree) to index all objects. Instead we propose to reduce the problem to a partial-persistence problem, that is, use a 2-dimensional access method that is made partially persistent. We show that this approach leads to faster query time while still using space proportional to the total number of changes in the frame evolution.

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. Each frame in the sequence contains some 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, disappear, etc.) For the purposes of editing or assembling movie sequences, it is important to have efficient ways to access and replay all, or parts, of movie frames. Example queries are: "find all objects that appear in area S between frames f_i and f_j". (S is part of the 2-dimensional frame screen and $f_i \leq f_j$.)

A conceptual view of a movie sequence appears in figure 1. The x, 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. The figure also shows a simple query: "find all objects inside area S in frame f_3"; only object o_1 satisfies this query.

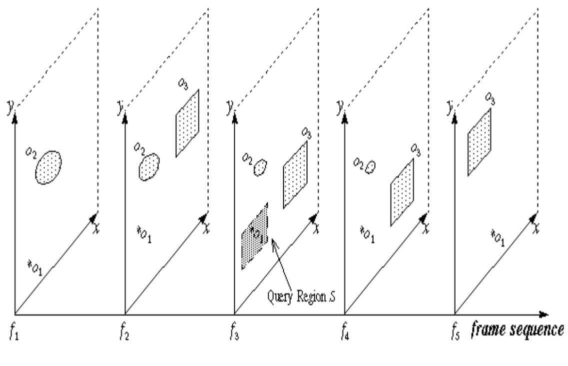

Figure 1: A conceptual view of a movie sequence.

There are two obvious but inefficient ways to address this problem. 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 area A. The main disadvantage however, is the high space redundancy. Many objects that do not change between frames will be stored several times. (At worst, the space can become quadratic to the number of object changes in the frame sequence).

The second straightforward approach is to store the changes between frames in a "log". This approach uses minimal space, but the query time is rather large as the frame of interest has to be reconstructed starting from the beginning of the log. Of course, one could choose to store a number of 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 (the more frequent the snapshots the larger the space), (ii) locating the objects in the query area S still requires extra effort that affects the query response.

We propose instead to view the evolution of the frames in the animated movie as a spatiotemporal application. That is, each frame is a 2-dimensional space that evolves over time. Time in this representation corresponds to the sequence of frame numbers (ids). In the rest we will use the terms time instant and frame number interchangeably. For simplicity, consider objects that simply are added or deleted from one frame to the next. Then, each object is associated with a "lifetime" interval $[f_i, f_j)$ created by the frame f_i where the object was added, until frame f_j where the object was deleted.

While simplistic, this assumption is enough to even represent objects that change position or extent in the frame sequence. For example, in figure 1, the movement of object o_1 from frame f_1 to a new position in frame f_2 can be represented by the artificial deletion of object o_1 at f_2 and the instantaneous rebirth of o_1 at f_2 located in the new position. Thus o_1 will have two consecutive (non overlapping) lifetimes. This paper considers only object additions and deletions as the basic changes in the spatiotemporal evolution.

Note that 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 this object gets deleted at a later frame f_j, its lifetime interval is updated to $[f_i, f_j)$. Note, that since we keep all frames (history) an object deletion corresponds in the database to a logical deletion that simply updates the right side of the object's lifetime interval.

Using the spatiotemporal approach, the problem of indexing animated objects is reduced to finding good index structures to index such spatiotemporal objects. One obvious solution is to consider time (the frame sequence) as another dimension. Then each object can be stored as a 3-dimensional rectangle. The length of the rectangle corresponds to the object's lifetime interval. Figure 2 shows the 3D rectangles of three objects. Object o_1 is added at frame f_1 and remains unchanged until the last frame f_{last}. Object o_2 is added at frame f_2 and is deleted at frame f_4, while object o_3 is added at f_3 and remains unchanged until f_{last}.

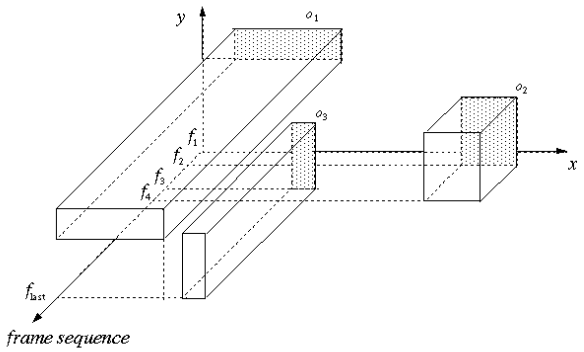

Figure 2: Storing objects as 3-dimensional rectangles.

A traditional spatial index can be utilized for storing such 3-dimensional spatiotemporal objects. Indeed this approach has been proposed in [12], where a 3D R-tree is used. Although simple and easy to implement (since it uses an "of-the-self" spatial index), this approach is problematic. The main disadvantage is that it does not take into account the properties of the time dimension. Objects that remain unchanged for many frames will have long lifetimes and thus will be stored as long rectangles. This makes the clustering of objects into pages difficult and as a consequence decreases the query performance.

We propose to use a different approach to index animated objects. In particular, we combine a spatial index (R-tree) with the partial persistent methodology.

A data structure is called *persistent* if a change 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. In [3] a method to make a main-memory data structure partially persistent is described. The data structure can be any pointer-linked structure with bounded in-degree. Using this method, any past version of the data structure can be accessed by paying only logarithmic overhead to the total number of versions. The space of the structure remains linear to the total number of changes in the evolution.

Partial persistence fits very nicely with the problem we address. Considering figure 1, assume that the objects in frame f_1 are indexed by a 2-dimensional R-tree. (Note that the methodology applies to other spatial indexes; we use a 2D R-tree for simplicity). As the frame number advances, this 2D R-tree evolves, by applying on it the changes as they occur in the appropriate frames. A change corresponds to the insertion or deletion of a 2-dimensional object. Since changes occur in increasing frame number order, they are always applied on the most current frame. Storing this 2D R-tree evolution corresponds to making a 2D R-tree partially persistent. That is, the partially persistent R-tree will conceptually store all versions that the ephemeral 2D R-tree went through while applying the object changes.

Of course, not all versions of the evolving 2D R-tree are stored as this would make the space quadratic to the number of changes. Instead, partial persistence allows to keep the R-tree evolution in space that is linear to the number of object changes. Answering a query about a given frame f_i would be addressed as if we had a 2D R-tree for that frame. The persistent R-tree will basically guide the search to the stored version of the ephemeral 2D R-tree that corresponds to frame f_i. This will provide an efficient way to select objects from frame f_i (in any case, we could not expect to do better than having an individual 2D R-tree on that frame!)

Methods to make an external memory data structure (in particular a B-tree) partially persistent are given in [1, 7, 13]. In [6] a partially-persistent R-tree is presented to support bitemporal queries. We propose here to extent this methodology for indexing animated objects. Note that another approach to store the evolution of the 2D R-tree is based on the notion of "overlapping" R-trees [8]. In the overlapping approach, parts of an evolving data structure that are not changed between subsequent versions, are shared. However, this approach is using more than linear space (there is a logarithmic overhead [9]).

Section 2 gives a background on the R-tree indexing scheme, while section 3 discusses the persistence methodology and the partially-persistent R-tree. Section 4 contains experimental results while section 5 concludes the paper and presents future research work.

2 Preliminaries

An R-tree is a hierarchical, height-balanced external memory data structure proposed by Gutman in [4]. It is a generalization of the B-tree for multidimensional spaces. Multidimensional objects are represented by a conservative approximation, usually the Minimum Bounding Rectangle (MBR). For this paper we will assume that each animated object is represented by the MBR that encloses it.

The R-tree consists of directory and leaf (data) nodes, each one corresponding to one disk page. Directory nodes contain entries of the form $(container, ptr)$ where ptr is a pointer to a successor node in the next level of the tree and $container$ is the MBR of all the entries in the descendent node. Leaf nodes contain entries of the form $(container, oid)$ where oid is an object-identifier and it is used as a pointer to the real object and $container$ is the MBR of the corresponding object. Each page can hold up to B entries and 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. An example of a 2-dimensional R-tree is shown in figure 3.

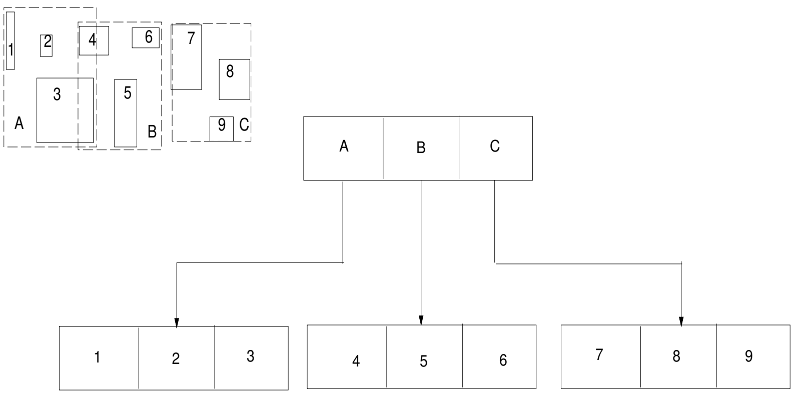

Figure 3: An R-tree example.

Searching in the R-tree is similar to the B-tree. At each directory node we test all entries against the query and then we visit all child nodes that satisfy the query. However, MBRs in a node are allowed to overlap and this is a potential problem with the R-tree, since, unlike B-trees, we may have to follow multiple paths when answering a query, although some of the paths may not contribute to the answer at all. In the worst case we may have to visit all leaf nodes.

A number of variations have been proposed in order to reduce the overlap among MBRs and therefore increase the query performance of the tree. These variations include the Hilbert R-tree[5] and the R*-tree[2]. Another variation (R+-tree[10]) does not allow overlapping among the MBRs of the same level of the tree.

3 Partially-Persistent R-tree

Our approach to make an R-tree partially persistent follows the approach taken in [1]. The partially-persistent R-tree is a directed acyclic graph of disk pages. This graph embeds many R-trees and has a number of root pages. Each root is responsible for providing access to a subsequent part of the ephemeral R-tree's evolution. Data records in the partially-persistent R-tree leaf pages maintain the time (frame sequence) evolution of the ephemeral R-tree data objects. Each record is thus extended to include two additional fields: *insertion-frame* and *deletion-frame*, representing the frame number the corresponding object was inserted and logically deleted in the database. Similarly, index records in the directory pages of the partially-persistent R-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. Therefore each record has a lifetime interval during which it is called *alive*.

Assume that each page in the partially-persistent R-tree has a capacity of holding B records. A page is called *alive* if it has not been *split*. With the exception of root pages, for all frame numbers that a page is alive it must have at least D alive records ($D < B$). This requirement enables clustering of the alive objects at a given frame number in a small number of pages, which in turn will minimize the query I/O. The first step of an update (insertion or deletion) at frame f_i locates the target leaf page in a way similar to the corresponding operations in an ephemeral R-tree. Note that this step is carried out by taking into account the lifetime intervals of the index and the data records visited, i.e. only the latest state of the ephemeral R-tree. An update leads to a *structural* change if at least one new page is created. *Non-structural* are those updates which are handled within an existing page. After locating the target leaf page, an insert operation adds a data record with an interval $[f_i, now)$ to the target leaf page.

This may trigger a structural change in the partially-persistent R-tree, if the target leaf page already has B records. Similarly, a delete operation at frame f_j finds the target data record and changes the record's interval to $[f_i, f_j)$. This may trigger a structural change if the resulting page ends up having less than D alive records as a result of the deletion. The former structural change is a *page overflow*; the latter is a *weak version underflow* [1]. Page overflow and weak version underflow need special handling: a *split* is performed on the target leaf-page. This is similar to the time-split of [7] or the page copying of [11]. The split on a page x at frame f, is performed by copying to a new page y the records alive in page x at f. Page 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). Then the resulting new page has to be incorporated in the structure (for details we refer to [6, 13, 1]).

Answering a spatiotemporal query about region S and frame f has two parts. First, the root alive at f is found. This part is conceptually equivalent to accessing the ephemeral R-tree which indexes frame f. Second, the answer is 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, while the record's MBR should intersect the region S. Answering a query that specifies a frame interval $[f, f')$ is similar. First all roots with interval intersecting the frame range are found and so on. Since the partially-persistent R-tree is a graph, some pages are accessible by multiple roots. Re-accessing pages can be avoided by keeping a list of accessed pages.

By "viewing" a spatiotemporal query as a persistence problem, we obtain a double advantage. First we disassociate 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 structure partially persistent. Second, changes are always applied on the most current state of the 2D structure and last until updated (if ever) at a later frame. This is beneficial because (i) an update searches only among the spatial objects that are in the current frame f and (ii) the overlapping due to frame sequence dimension is avoided.

4 Experiments

To test the merit of our approach, we experimentally compared the partially-persistent R-tree (PP R-tree) against the 3D-Rtree approach (where the third dimension represents the frame sequence). In our experiments we used the R*-tree which is considered as one of the best methods in R-tree family.

We generated a set of simple spatiotemporal evolutions in a way similar to what described in [12]. Objects in a given frame are approximated by their 2D MBRs. Thus 70% of the objects are small rectangles with small lifetimes. Another 15% of the objects are large rectangles with small lifetimes and the remaining 15% are small rectangles with large lifetimes. The rectangles were generated uniformly in the unit square and the lifetimes follow a Poisson distribution. We generated five datasets with average number objects at any frame ranging form 1000 to 5000 objects while the evolutions (movies) last for $F = 10000$ frames. The queries are uniformly distributed ranges in the unit square at a specific frame between 0 and F (snapshot queries).

In the next figures we present the results for the average query time and the space consumption of the compared methods.

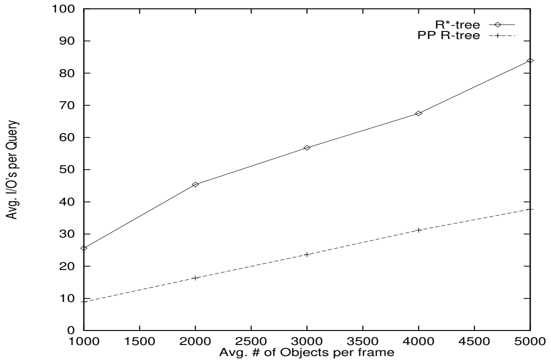

Figure 4: Query performance for snapshot queries.

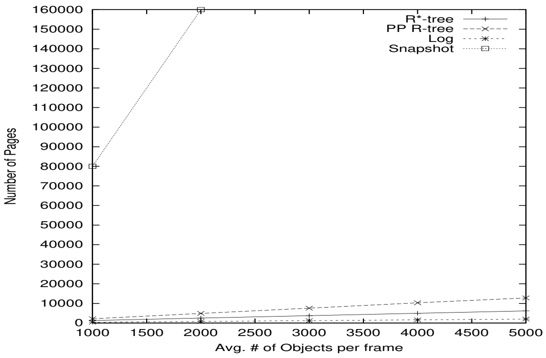

Figure 5: Space consumption.

As expected the partial persistent method has better query performance (about 2 times better). This is because objects are better clustered into pages in the PP R-tree since the long MBRs resulted by the extensive lifetimes are not present. The PP R-tree has to cluster objects per frame (i.e. 2D instead of 3D). The persistence methodology however uses (about 2 times) more space due to the extra copies from the page splits. Note however that the space of the PP R-tree is still linear to the number of changes. This can be observed in figure 5: the space increases linearly with the number of objects (the number of changes is linear to the number of objects). In the same figure we have also printed the space used by the two straightforward approaches (the "snapshot" and the "log"). The "log" corresponds to the minimal space (the number of changes). Note that the space for the "snapshot"approach is only partially shown.

In another set of experiments, we generated range queries with an interval as a time predicate (period queries). We varied the time interval from 0 to 30 time instants. Again the spatial predicate of each query was generated as random rectangles in the unit square. The dataset had about 8000 alive objects per frame. In figure 6 we present the results for these experiments. As we can see the partially persistent R-tree again outperforms the 3D-Rtree.

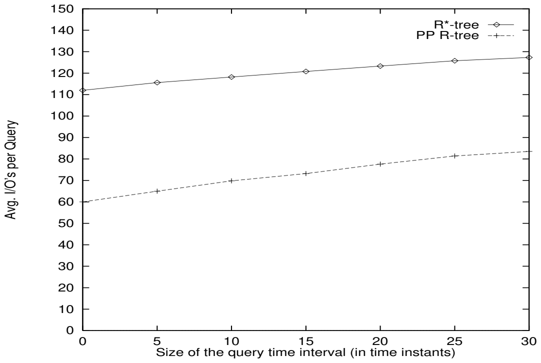

Figure 6: Query performance for period queries.

5 Conclusions and Further Research

We have examined the problem of indexing objects in animated movies. We have proposed a solution that reduces the original problem to a problem of partial persistence. This approach provides very fast query time at the expense of some extra space, which however is linear to the number of changes in the frame evolution. We have shown the merit of our approach by comparing it with an approach that sees the frame sequence as simply another dimension and uses a 3D index. In this preliminary report we have only considered "static" changes, that is, object additions and deletions. These changes can represent efficiently objects that do not move fast from frame to frame. However they are not so efficient for cases where objects change position and/or extent very frequently; the result would be creating artificial deletions and rebirths for all such objects from frame to frame, i.e., a large number of changes and of course large space [8]. We are currently examining better ways to apply the partial persistence methodology on such scenarios. The problem is how to efficiently choose when an object should be artificially deleted. Finally we note that this approach has merit for general spatiotemporal applications (as for example keeping the history of objects moving over a plane, etc.). In the extended version of the paper we will also compare with the overlapping R-tree approach of [8].

References

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

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

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

[8] M. Nascimento, J. Silva and Y. Theodoridis. Evaluation of Access Structures for Discretely Moving Points. to appear in STDBM'99, Edinburgh, 1999.

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

[10] 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.

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

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

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