Journal of Universal Computer Science, vol. 13, no. 10 (2007), 1411-1433
submitted: 12/10/06, accepted: 19/10/07, appeared: 28/10/07 © *J.UCS*

Efficient Access Methods for Temporal Interval Queries of Video Metadata[1]

Spyros Sioutas
(Department of Informatics, Ionian University, Corfu, Greece
sioutas@ionio.gr)

Kostas Tsichlas
(Department of Informatics, Aristotle University of Thessaloniki, Greece
tsichlas@csd.auth.gr)

Bill Vassiliadis
(Hellenic Open University, Computer Science, Digital Systems & Media Computing Lab
Patras, Greece
bb@eap.gr)

Dimitrios Tsolis
(Computer Engineering and Informatics Department, University of Patras, Greece
dkt@hpclab.ceid.upatras.gr)

Abstract: Indexing video content is one of the most important problems in video databases. In this paper we present linear time and space algorithms for handling video metadata that represent objects or events present in various frames of the video sequence. To accomplish this, we make a straightforward reduction of this problem to the intersection problem in Computational Geometry. Our first result is an improvement over the one of V. S. Subrahmanian [Subramanian, 1998] by a logarithmic factor in storage. This is achieved by using different basic data structures. Then, we present two other interesting time-efficient approaches. Finally a reduction to a special geometric problem is considered according to which we can achieve two optimal in time and space solutions in main and external memory model of computation respectively. We also present an extended experimental evaluation.

Keywords: video databases, data structures, computational geometry
Category: E.1, E.5, H.3.1, H.3.3

1 Introduction

As large-sized video information becomes available through numerous channels, including the Internet, the need for managing it efficiently becomes even more urgent. In a first level of abstraction, managing video means searching and retrieving. But in order to achieve this, the appropriate mechanisms for analysing, representing and indexing video objects need to be available and most of all, efficient. Video

[1] Preliminary parts of this work were presented in **MDDE 2001** and **VLDB/MDDE 2003** workshops

management is a large and fascinating research domain, where an interdisciplinary endeavours from communities such as information retrieval, artificial intelligence, signal processing and knowledge management flourish [Nack, 2005].

Research in searching in video databases, a highly demanding task is in general following three routes: the one of pure content search where queries by content are used, metadata searching where annotated content is searched and mixed where content is analysed and metadata are produced, form searchable indexes [Dimitrova, 2002].

In turn, searching tasks rely on indexing mechanisms to achieve faster responses to user queries. Both content and metadata can be indexed. In fact, metadata management is gaining importance as more and more automatic, semi-automatic annotation and video analysis mechanisms become available [Madhwacharyula, 2006; Nack, 2000].

Indexes are used for facilitating searches in large data corpora so why indexing metadata? The size of video metadata depends on the method used to analyse the video sequence. In this work we make the assumption that a video is analysed and the objects and activities which appear at certain (or all) frame ranges are extracted. An object may be an item, a person or generally something tangible. An activity is an action or a relation between certain objects. There is no difference in the way we handle objects and activities and so we will refer only to objects. We expect that the size of the metadata extracted is quite large. We attack the problem of searching large metadata index corpora produced by such a video analysis algorithm. The indexed produced could serve as a standalone or complementary index for searching very large video sequences in demanding applications. Its practical value lays in the fact that it performs well independently of the level of detail of the metadata produced; indexing may involve only important or user-marked objects, all objects in a frame or all objects in the video sequence. This is a kind of video metadata abstraction in the sense that the maximum information about the target video sequence can be stored in an index. Furthermore, the metadata produced are rather low-level and will largely remain unchanged throughout the video lifecycle. On the other hand, high level metadata may change more often in size, type or semantic value [Kosch, 2005].

We would like our video database to be able to answer efficiently queries of the form: *Find and report all objects that appear in a given range of frames*. In order to accomplish this we must store the range sequences, where each object appears, in a data structure, associate each of these frames to its respective object and query this structure. The query segment is used to extract all the frame sequences intersecting it. Having found these segments it is straightforward to find the objects appearing in this query frame segment. Thus, our query is a simple intersection query of segments on the line, where the query is also a segment. The intersection problem defined above is static because in a video there is a predefined set of objects and respective frame segments.

Previous main memory solution ware based on a well-known data structure used extensively in the domain of Computational Geometry, the *segment tree* [Bentley, 1977],[Mark de Berg, 1997],[Mehlhorn, 1984],[Preparata, 1985]. This is a very simple and elegant data structure but exhibits certain deficiencies in specific geometric problems. If we assume that each object is appearing in a sequence of frame segments, then we store each of these segments in $O(\log n)$ nodes of the

segment tree, where n is the number of distinct endpoints of the frame segments stored in the segment tree. This is a sheer waste of space if we imagine that each video may have many objects associated to many frame segments. In addition, the solution described in [Subramanian, 1998] may report each of these ranges $O(\log n)$ times, which in many cases may be undesirable. To remedy these problems we resort to a more suitable data structure, the *interval tree* [Mark de Berg, 1997],[Mehlhorn, 1984],[Preparata, 1985]. The interval tree uses linear space because it stores each range only once, as we show in the following sections. Furthermore, to optimize the query performance we can also resort to more efficient methods including fusion trees [Willard, 1992] and duality geometric transformations.

In previous external memory solutions the problem we study is known as timeslice or timestamp queries. These queries retrieve all objects that intersect a window at a specific timestamp. Interval queries include several (usually consecutive) timestamps. Dealing with the temporal dimension, one of the first spatiotemporal data structures was the RT-tree [Xu, 1990], which stores both spatial and temporal information in the nodes of an R-tree [Guttmann, 1984] (i.e., in addition to its spatial extent, each node contains the time interval during which the corresponding object is alive). Historical R-trees (HR-trees) [Nascimento, 1998] applied the concept of overlapping B-trees [Manolopoulos, 1990] to R-trees. The main idea was to construct an R-tree for each timestamp in history. 3DR-trees [Vazirgiannis, 1998] were based on 3-dimensional R-trees where the third dimension corresponds to time. MV3R-trees [Tao, 2001] include a small auxiliary 3D R-tree on the leaf nodes (not on the actual objects) and they usually outperform traditional 3D R-trees on interval queries. Finally, the SEST-Index [Gutierrez, 2005] combines snapshots and events. By using an R-tree structure for storing snapshots and a log data structure for storing events, which occur between consecutive snapshots, it outperforms HR-trees.

In section 2 we are going to give a thorough review of the previous solution while in section 3 certain preliminary data structures are going to be described synoptically. In section 4 the data structure is presented and the result is given respectively. In section 5 a reduction to another geometric problem is considered while in section 6, fusion tree methods are applied. In section 7 a reduction to the quadrant-range searching is considered according to which we present two optimal solutions in main memory model of computation. In section 8 we externalize the previous optimal solution. In section 9 we present an experimental evaluation. In section 10 we conclude.

2 Previous work

2.1 Main Memory Solution

Assume that initially we are given a table of n objects o_i and the associated frame segments of a given video v with total number of frames equal to $framenum(v)$. We want to organize this metadata in order to answer efficiently video content queries.

Assume that $[s_1, e_1), \ldots, [s_w, e_w)$ are all the intervals in this table. Let $q_1, \ldots, q_n$ be an enumeration, in ascending order, of all members of $\{s_i, e_i \mid 1 \le i \le w\}$, with

duplicates eliminated. If n is not an exponent of 2, then do as follows: let r be the smallest integer such that $2^r > n$ and $2^r > framenum(v)$. Add a number of new elements $q_{n+1},\ldots,q_{2^r}$ such that $q_{2^r} = framenum(v)+1$ and $q_{n+j} = q_n + j$ (for $j > 0$ such that $n + j < 2^r$). Now we may proceed under the assumption that n is an exponent of 2. The next step is to construct the indexing data structure that is called the frame segment tree. This is a full binary tree.

Each node in the frame segment tree represents a frame sequence $[x, y)$, starting at frame x and including all frames up to, but not including, frame y. All leaves are at level r, where obviously $r = O(\log n)$. The leftmost leaf denotes the interval $[q_1, q_2)$, the 2nd from the left represents the interval $[q_2, q_3)$ and so on. If u is a node with two children representing the intervals $[p_1, p_2), [p_2, p_3)$, then u represents the interval $[p_1, p_3)$. Thus, the root of the frame segment tree represents the interval $[q_1, q_n)$ if q_n is an exponent of 2, otherwise it represents the interval $[q_1, +\infty)$.

Without proof we give below some elementary results on segment trees.

1. The segment tree uses $O(n \log n)$ space
2. Each segment is stored in $O(\log n)$ nodes.

The second property of the segment tree is a cause of problems because of two reasons:

1. There is a sheer waste of space.
2. In the reporting procedure one particular segment may be reported $O(\log n)$ times.

The reporting procedure given by Subrahmanian is given below. By R we represent the subtree rooted by the current node. In the first call of the procedure, R is the whole tree. The parameters s and e define the endpoints of the query segment. The variable S is the output of the procedure. In each node of the frame segment tree we store a linked list of the objects. These objects have frame segments associated to the frame sequence of the specific node. In this way, if a node's frame sequence intersects the query frame segment then the linked list of this node is appended to variable S. The frame sequence of a node v is represented by $[v.LB, v.RB)$. The right and left child of the node v is represented by $v.RLINK$ and $v.LLINK$ respectively.

```
FindOInV(R, s, e);
{
  S = NIL;
  if (R = NIL) then {Return(S); Halt};
  else
  {
    if ([R.LB, R.UB) ⊆ [s, e))
      then S = append(S, preorder(R))
    else
    {
```

```
        if (([R.LB, R.UB) ∩ [s, e)) ≠ ∅) then
        {
            S = append(S, R.obj);
            S = append(S, FindOInV(R.LLINK, s, e));
            S = append(S, FindOInV(R.RLINK, s, e));
        }
      }
    }
    Return(S); end;
}
```

The running time of the algorithm is proportional to the total number of nodes visited, which may be at most $O(n)$.

2.2 External Memory Solutions

Figure 1 shows an example of the evolution of a set of multimedia or spatio-temporal objects of a video database in different instants of time. For simplicity, an example in a two-dimensional space is considered. In Figure 1, the axes x and y represent the two- dimensional space while t corresponds to the temporal dimension. In instant of time t1, objects O1 is inserted. In instant of time t2, object O1 moves. In instant of time t3, object O2 appears. In instant of time t4 objects O1 and O2 disappear while a new object O3 appears. A time slice query Q is also shown in Figure 1. This query is expressed in the following way: "find objects that appear in the rectangle Q at time interval $[t_2,t_3]$.

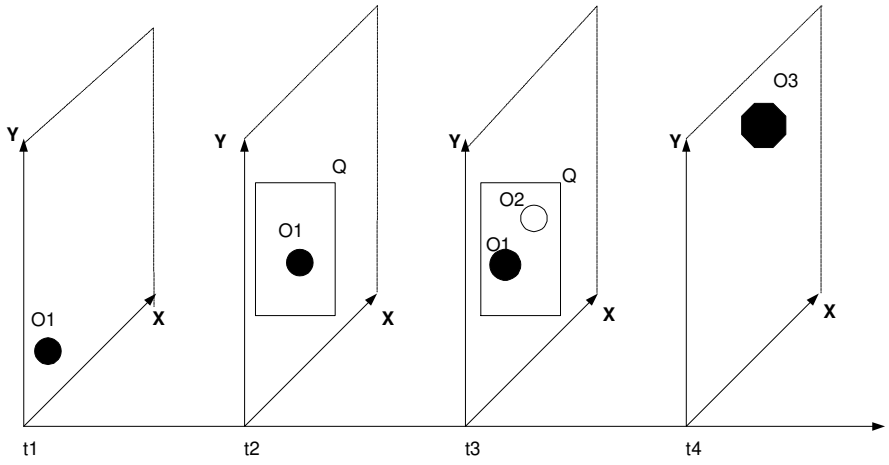

Figure 1: The evolution of a set of multimedia objects of a video database in different instants of time

Dealing with the temporal dimension the most famous previous access methods are the following:

RT-trees [Xu, 1990]: RT-tree stores both spatial and temporal information in the nodes of an R-tree [Guttmann, 1984]. In addition to its spatial extent, each node contains the time interval during which the corresponding object is alive. The insertion and split strategies, however, can hardly favour the spatial and temporal attributes at the same time. One of the methods proposed in [Xu, 1990] is to split nodes according to spatial information only; thus the temporal field plays a complementary role. This makes RT-tree inefficient when handling time-related queries, as it is likely that all the objects satisfying the spatial predicate are retrieved before being filtered according to the temporal predicate.

HR-trees [Nascimento, 1998]: Historical R-trees (HR-trees) apply the concept of overlapping B-trees [Manolopoulos, 1990] to R-trees. The main idea is to construct an R-tree for each timestamp in history. However, since consecutive R-trees can make use of common paths if objects do not change their positions, new branches are created only for objects that have moved. HR-trees are very efficient for timestamp queries, as search degenerates into a static spatial window query for which R-trees are very efficient. Their disadvantage is extensive duplication of objects (even if they do not move) which leads to huge space requirements for most typical applications. As a side effect of this fact, their performance on interval queries is very poor.

3DR-trees [Vazirgiannis, 1998]: Another technique is based on 3-dimensional R-trees where the third dimension corresponds to time. An object which does not change its position during a certain period of time is modelled as a cube, bounding both its spatial and temporal attributes. A moving object can be modelled by multiple cubes, each corresponding to a different version. The strength of 3DR-trees is that the temporal attribute is integrated tightly with the spatial attributes, so that interval queries can be answered efficiently. Another advantage is its economical space usage as redundant duplication is avoided. The most serious problem of this structure is its poor performance on timestamp queries. The query time no longer depends on the live entries at the query timestamp, but on the total number of entries in history. Since all objects are indexed by a single tree, the size and height of the tree is expected to be larger than that of the corresponding HR-tree at the query timestamp.

MV3R-trees [Tao, 2001]: MV3R-trees include a small auxiliary 3DR-tree on the leaf nodes (not on the actual objects) and, as it was demonstrated, they usually outperform traditional 3DR-trees on interval queries while their performance does not deteriorate significantly when time evolves (as is the case with regular 3D R-trees).

SEST-trees [Gutierrez, 2005]: SEST-Index, which combines snapshots and events, uses an R-tree structure for storing snapshots and a log data structure for storing events that occur between consecutive snapshots. Experimental results that compare SEST-Index and HR-tree showed that SEST-Index outperforms HR-tree for interval queries. In addition, as SEST-Index is an event-oriented structure, event queries are efficiently answered.

Obviously, for time interval queries, the best access method is the MV3R-tree. Of course the initial query that we investigate thoroughly at this paper "Find and report all objects that appear in a given range of frames" derives from the general interval query "find objects that appear in the rectangle q at time slice $[t_i,t_{i+k}]$", if we resize the

rectangle q on it's maximum size. The latter means that we report all the objects appear on the xy-plane between t_i and t_{i+k} time-instants.

3 Preliminary data structures

This section is devoted to the interval tree. It allows us to store a set of *n* intervals in linear space such that intersection queries can be answered in logarithmic time.

Let $S = [x_i, y_i], 1 \le i \le n$ be a set of *n* closed intervals on the real line. Let $Q = q_1, \ldots, q_n$ be an enumeration, in ascending order, of all members of $\{x_i, y_i \mid 1 \le i \le n\}$, with duplicates eliminated. An interval tree *T* for *S* is a leaf-oriented search tree for *Q* where each node of the tree is augmented by additional information.

We define $xrange(v)$, where v is a node of the interval tree, as the interval $\lfloor q_{left}, q_{right} \rfloor$ such that: q_{left} is the leftmost leaf of the subtree rooted at *v* while q_{right} is the rightmost one.

The *node list* $NL(v)$ of node *v* is the set of intervals in *S* containing the split value of *v* but of no ancestor of *v:*

$$NL(v) = \{[x, y] \in S; split(v) \in [x, y] \subseteq xrange(v)\}$$

We store the node list of node *v* as two sorted sequences: the ordered list of left endpoints and the ordered list of right endpoints. Both sequences are stored in balanced search trees; furthermore, we provide pointers to the maximal (minimal) element of the sequence of right (left) endpoints.

The main power of interval trees stems from the node lists. The following lemma shows that interval trees use linear space, can be constructed efficiently, support insertions and deletions of intervals and answers intersection queries efficiently.

Lemma 1: Let *S* be a set of *n* intervals.

a) An interval tree for *S* uses space $O(n)$.

b) An interval tree for *S* of depth $O(\log n)$ can be constructed in time $O(n \log n)$.

c) Intervals can be inserted into an interval tree of depth $O(\log n)$ in time $O(\log n)$. The same holds for deletion.

d) Let *S* be a set of intervals and let $I = [x_0, y_0]$ be a query interval. Let $t = \{[x, y] \in S; [x, y] \cap [x_0, y_0] \neq \varnothing\}$ be the set of intervals in *S* intersecting *I*. Then, given an interval tree of height $O(\log n)$ for *S*, one can compute the intersection query *t* in time $O(\log n + t)$.

Proof. See [Mehlhorn, 1984].

4 The new algorithm

We store each frame segment associated to an object in the interval tree. We must note that it is not imperative to extend the tree to a full binary one since this leads to waste of space.

Each frame segment is inserted in the interval tree. It is stored only in one node and thus we obtain a logarithmic save in space. This frame segment is associated with an object. Namely, the object appears in the frame sequence defined by this frame segment. In this way, each segment is related to only one object and is stored only once.

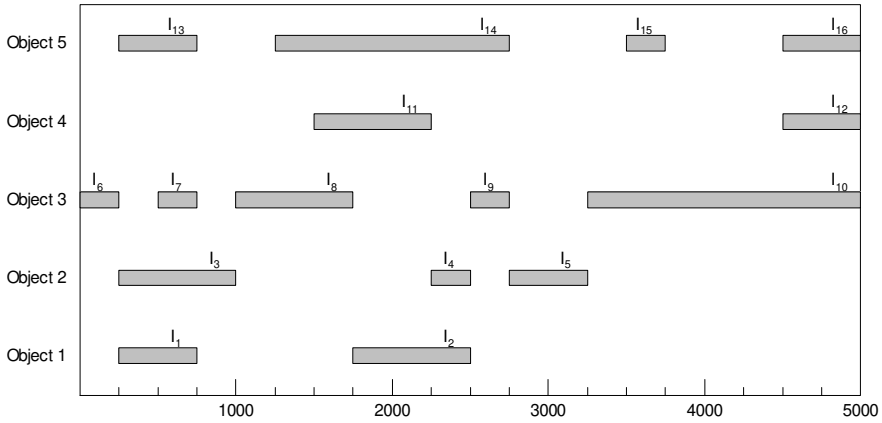

Figure 2: Example of the contents of a video

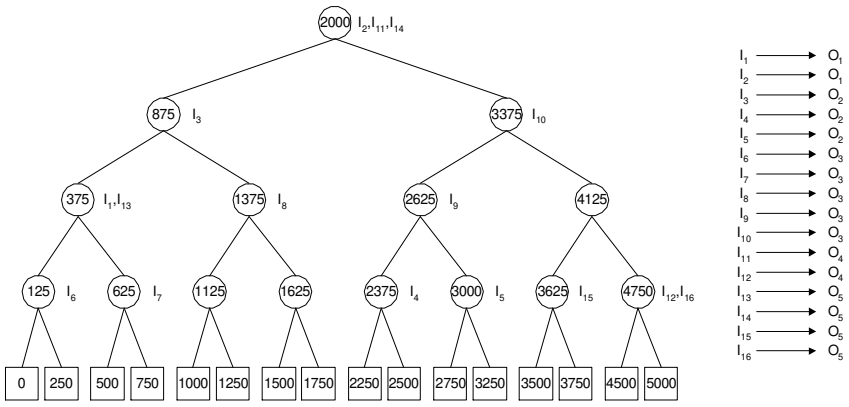

Figure 3: The frame interval tree with intervals associated with nodes and each interval associated with an object

In each node of the interval tree there may be many segments stored to its node list. These segments are stored sorted in each of these node lists. Attached to each such

segment is the object to which the specific segment belongs. The attachment is expressed by a pointer to a list of objects corresponding to the specific node. In this way, when we find a segment, which intersects the query segment *I*, we immediately report the respective object.

The search for all the segments intersecting the query segment is described below. We start from the root and search for the two endpoints of the query segment. The two paths from the root to the two leaves of the interval tree comprise the set *P*. These two paths coincide until they reach a node, called *split node*. The leftmost and rightmost leaves of its subtree are the endpoints of the query segment *I*. Let P_{Left} and P_{right} be the two paths from split node to the left and right endpoints of *I* respectively. Similarly, the set *C* is defined as the set of nodes *v*, where $xrange(v) \subseteq I$.

Intuitively, the nodes that belong to the set C are defined by the right (left) subtrees of the nodes in set P_{Left} (P_{Right}). Thus, when we are reporting the answer corresponding to a query segment we may just append the list of objects associated to a node, belonging to set *C,* to the answer. In nodes of set *P* we are obliged to search inside the node lists. These node lists, as mentioned in the preceding section, are organised as binary trees whose leaves are connected in a linked list and whose root has pointers to the smallest and to the largest element (endpoint). In each node list of a node in a set *P* some segments may intersect *I* and some others may not. Because of the fact that the segments in the node lists are stored sorted we are able to report the answer in these in constant time per object.

The largest or smallest element is accessed in constant time through the respective root pointers. We then traverse the linked list of leaves until we reach a segment that does not intersect the query segment. If the node under consideration lies on the path to the left endpoint we begin the traversal of the linked list from the largest element (rightmost leaf). If the node lies on the right we begin from the smallest element (leftmost leaf). If the node lies on both of them, that is, on the path from root to the split node of the two paths, we compare the endpoints of the query segment to the largest and smallest elements and we proceed analogously.

For better comprehension of the above method we give a simple example. Assume that we have a video and we are interested for only 5 objects. Each object appears in the video in a sequence of series of frames defining frame segments. Figure 2 depicts objects and their location in a video consisting of 5000 frames. Figure 3 depicts the frame interval tree.

5 Reduction to Dominance

In the preceding sections we associated the problem of querying video content with that of finding segments that intersect a specific segment.

In this section we outline a different approach to this problem. Specifically, we show how to reduce the problem of querying video content to the dominance problem.

The dominance problem is a pure geometrical problem.

The *d*-dimensional dominance problem is defined as follows: given a set Q of *d*-dimensional points and a query point *p*, report all points $p' \in Q$ such that p' is dominated by *p*. A point $p' = (p'_1, p'_2, \ldots, p'_d)$ is dominated by $p = (p_1, p_2, \ldots, p_d)$ if and only if $p'_i \leq p_i, \forall i$. Here we consider the 2-dimensional dominance problem.

Let $f = [f_1, f_2]$ be a given frame segment. A query segment $q = [q_1, q_2]$ intersects *f* if and only if one of the following conditions holds:

(i) *f* contains *q*, that is $f_1 \leq q_1 \leq q_2 \leq f_2$ (see figure 4(a)).

(ii) *f* partially intersects *q*, that is $f_1 \leq q_1 \leq f_2$ (*f* intersects *q* on the left) or $f_1 \leq q_2 \leq f_2$ (f intersects q on the right) (see figure 4(b) for the first case).

(iii) *f* is contained in *q* that is $q_1 \leq f_1 \leq f_2 \leq q_2$ (see figure 4(c)).

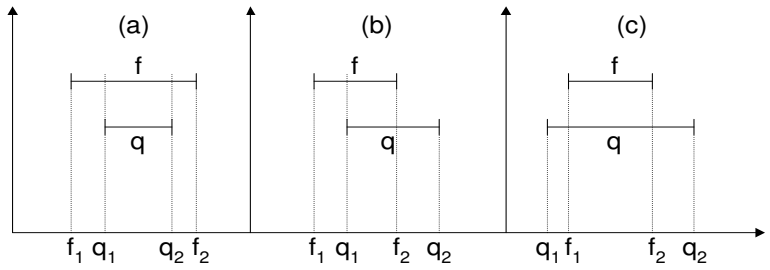

Figure 4: Query segment q compared to a frame segment f, (a) q is contained in f, (b) q's left endpoint is contained in f and (c) q contains f.

From the above cases it is trivial to see that it suffices to store the frame segments in a two-dimensional dominance-searching problem. This structure stores each frame segment $f = [f_1, f_2]$ as a two-dimensional point (f_1, f_2). Then, given a query segment $q = [q_1, q_2]$ we find all segments intersecting *q* by querying the structure with $(q_1, -q_2)$ (case (i)), $(q_1, -q_1)$ (case (ii) left partial intersection), $(q_2, -q_2)$ (case (ii) right partial intersection) and $(-q_1, q_2)$ (case (iii)).

The two-dimensional dominance-searching problem has an optimal RAM dynamic solution using linear space, exhibiting query time $O(\log n / \log \log n + k)$ and update time $O(\log n / \log \log n)$, where *k* is the size of the output (number of reported objects in our case) ([Willard, 1992]).

The static counterpart of this problem has an optimal RAM solution of linear space, and $O((\log n / \log \log n)^{1/2} + k)$ query time (simple combination of persistence and the search structure of [Beam, 1999]).

Finally, in secondary memory an optimal dynamic structure has been proposed recently ([Arge, 1999]) that occupies $O(n/B)$ disk pages (*B* is the size of a page),

supports insertions and deletions in $O(\log_B n)$ I/Os and answers queries in $O(\log_B n + k/B)$ I/O's.

6 The fusion tree solution

First, we will briefly review the data structures used in this solution.

6.1 The fusion tree

Let *S* be an ordered set of *n* *w*-bit keys. The *fusion tree* [Willard, 1992] is a dynamic data structure that supports $O(\log n/\log\log n)$ amortized time queries in linear space. This structure is a two-level data structure where the upper level consists of an ordinary *B*-tree while the lower level consists of weighted-balanced trees. The amortized cost of searches and updates is $O(\log n/\log b + \log b)$ for any $b = O(w^{1/6})$. The first term corresponds to the number of *B*-tree levels and the second to the height of the weighted-balanced trees.

The main advantage of the fusion technique is that we can decide in constant time in which subtree to continue the search by compressing the *b*-keys of every *B*-tree node using *w*-bit words.

6.2 The exponential search tree

The *exponential search tree* [Anderson, 1996] answers queries in one-dimensional space. It is a multi-way tree where the degrees of the internal nodes decrease exponentially as we traverse the levels of the tree starting from the root. Auxiliary information is stored in each node to support efficient search queries. The exponential search tree has the following properties:

1. Its root has degree $\Theta(n^{1/5})$.
2. The keys of the root are stored in a local data structure. During a search procedure, the local data structure is used to determine in which subtree of a node the search is to be continued.
3. The subtrees are exponential search trees of size $\Theta(n^{4/5})$.
4. The local data structure of each node of the tree is a combination of van Emde Boas trees and perfect hashing. As a result we achieve $O(\log w \log\log n)$ worst-case time cost for a search query.

Anderson, by using an exponential search tree in the place of *B*-trees in the fusion tree structure, avoids the need for weight-balanced trees at the bottom while at the same time improves the complexity for large word sizes. This structure is a significant improvement on linear space deterministic sorting and searching. On a unit-cost RAM with word size *w*, an ordered set of *n* *w*-bit keys (viewed as binary strings or integers) can be maintained in $O\left(\min\left\{\sqrt{\log n}, \log n/\log w + \log\log n, \log w \log\log n\right\}\right)$ time per operation, including *insert*, *delete*, *member search* and *neighbour search*. The cost for searching is worst–case while the cost of updates is amortized. For range queries there

is an additional cost of reporting the found keys. As an application, n keys can be sorted in linear space at a worst-case time cost of $O\left(n\sqrt{\log n}\right)$. The best previous method for deterministic sorting and searching in linear space has been the fusion tree, which supports search queries in $O(\log n/\log\log n)$ amortized time and sorting in $O(n\log n/\log\log n)$ worst–case time.

6.3 The fusion interval tree

Let T be a B-ary tree, that is a tree for which each node has B sons. We set the branching factor $B=O\left(\sqrt{\log n}\right)$. Each node v in ordinary interval trees such that $v=nca(x_1,x_2)$ (nca=nearest common ancestor), stores the value $range(v)=[x_1,x_2]$. Thus, in each node v such that $v=nca(x_1,x_2)$ we store the following B slabs:

$sl_1=(x_1,k_1],sl_2=(k_1,k_2],\ldots,sl_B=(k_{B-1},x_2]$

We define the structure of each node list $NL(v)$ as follows: $NL(v)=\{s\in S;\ s$ spans a slab of v and s is included in a slab of $parent(v)\}$.

If $s\in NL(v)$, then s spans a continuous set of slabs $sl_i,\ldots,sl_{i+k}$ and s cuts the two bound-slabs sl_{i-1},sl_{i+k+1}. In the slabs $sl_i,\ldots,sl_{i+k}$, s is stored in an unordered list but in sl_{i-1},sl_{i+k+1} s is stored in an ordered list (see figure 5) of endpoints that is organized as an exponential search tree [Anderson, 1996]. Thus, the total required space for the node lists is $O(nB)$.

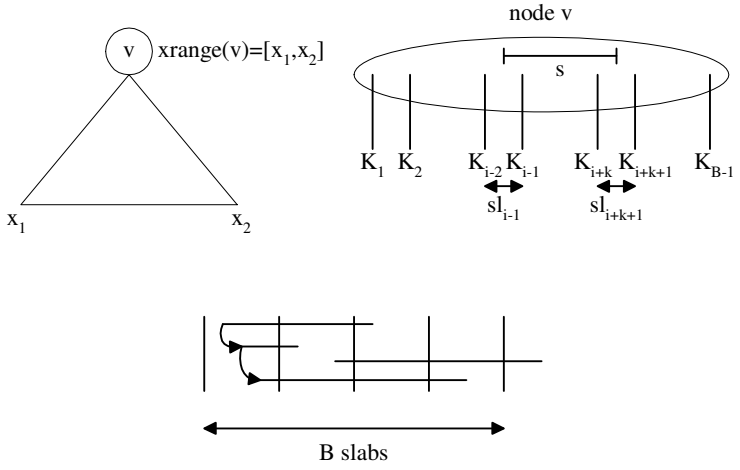

Figure 5: Depiction of the xrange of a node, of the slabs and their order inside ordered lists.

Lemma 2: Let $U \subseteq \Re$ be an ordered finite universe and let *S* be a set of *n* intervals with left endpoints in *U*. To simplify our applications we assume that $|S| = |U| = n$.

a) The fusion interval tree *T* for *S* requires $O(nB)$ space.

b) The fusion interval tree *T* for *S* can be constructed in time $O(nB)$.

c) Intervals (with left endpoint in *U*) can be inserted into *T* in time $O(B)$. Deletions are completely symmetric.

d) The intersection query *t* can be solved in $O(\log n/\log\log n + t)$ time.

Proof:

a) A *B*-tree for *U* clearly uses linear space $O(|U|) = O(n)$. Furthermore, the total space required for the node lists is $O(nB)$ since every interval in *S* is stored in the *B* slabs. The space complexity follows.

b) A *B*-tree can be built in time $O(|U|) = O(n)$ and has depth: $O(\log_B |U|) = O(\log n/\log B) = O(\log n/\log\log n)$ It remains to construct the node lists. We show how to insert an interval $s = [x_1, x_2]$ in $O(B)$ time. Let *v* be the nearest common ancestor of the endpoints of *s*. This node can be computed in constant time [Schieber, 1988]. It remains to insert the interval *s* in the *B* slabs of node *v*. Assuming that *s* spans a continuous set of slabs $sl_i, \ldots, sl_{i+k}$ and cuts the two bound-slabs sl_{i-1}, sl_{i+k+1} we can insert the interval *s* in the slabs $sl_i, \ldots, sl_{i+k}$, in an unordered list with $O(B)$ cost. However, we have to insert interval *s* in the ordered lists of sl_{i-1}, sl_{i+k+1}. This incurs $O(\sqrt{\log n})$ cost since the ordered list of each slab is organized as an exponential search tree [Anderson, 1996]. Thus, the sequence of insertions of *n* intervals require $O(nB)$ time and as a result the total construction time is $O(n + nB) = O(nB)$.

c) It is obvious from the construction above, that we can insert an interval $s = [x_1, x_2]$ in $O(B)$ time. The same holds for deletions.

d) We define sets *P* and *C* in the same way as in section 4. *P* consists of the nodes on the search paths to *x* and *y* and *C* is the set of nodes between these paths. Let *t* the output of intersection query *I*. Since $[x, y] \in NL(v)$ and $[x, y] \cap I \neq \varnothing$ implies $xrange(v) \cap I \neq \varnothing$, *t* is defined as follows: $t = \sum_{v \in C} NL(v) \cup \sum_{v \in P} \{(x, y) \in NL(v); [x, y] \cap I \neq \varnothing\}$. In addition, the fact that $v \in C$ clearly implies that $NL(v) \subseteq t$. Now consider nodes *v* such that $v \in P$. Recall that we organized $NL(v)$ as two ordered lists, the list of left endpoints and the list of right endpoints. Let $x_1 \leq x_2 \leq \ldots \leq x_k$ be the former list and let $y_1 \leq y_2 \leq \ldots \leq y_k$ be the latter list. We have to discuss three cases, two of which are symmetric. Suppose first

that $xrange(v) \subseteq I$. Then $NL(v) \subseteq A$, since we know that $xrange(v) \subseteq [x, y]$ for all $[x, y] \in NL(v)$.

Suppose that $xrange(v) \subseteq [x_1, x_2] \not\subset I$ and $x_1 \leq x_0$ (the other case is symmetric). Then, interval $[x_i, y_i] \in NL(v)$ intersects I if $x_0 \leq y_j$. We can thus find all such intervals by inspecting $y_k, y_{k-1}, \ldots$ in turn as long as they are at least as large as x_0 . Hence we can determine $NL(v) \cap t$ in time proportional to $|NL(v) \cap t|$.

Thus, the time required for the computation of *t* is $O(|P| + |C| + t)$. For the case of a "small" universe *U*, which means that *U* contains only endpoints of intervals in *S*, it holds that $|C| = O(t)$ since all leaves in *C* are endpoints of intervals in *t*. Since $|P| \leq 2h$, where $h = O(\log n / \log \log n)$, the time bound follows.

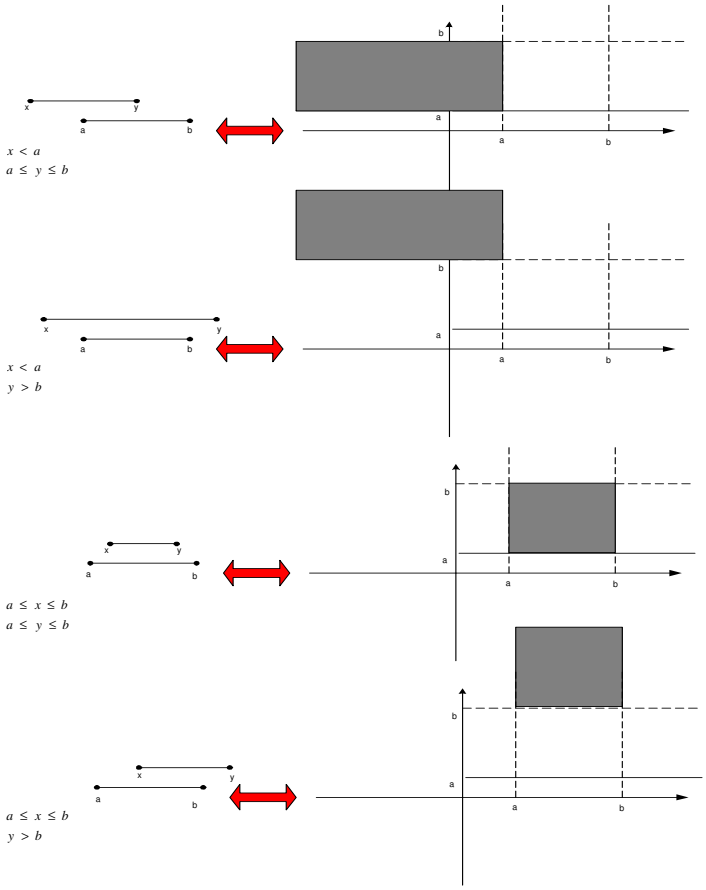

Figure 6: The 2-D transformation of the four possible cases

7 Reduction to Quadrant Range-Searching Problem

In this section we will show how we can transform the video content query to a static quadrant range-searching problem giving two optimal linear space solutions in main memory with O(k) query time, where k the size of the answer.

Considering the intersection between a video object (arbitrary interval [x,y]) and a frame interval query (reference interval [a,b]) there are four possible cases as they are depicted on the left side of the figure 6 above.

We can view each of the intersections above (between [x,y] and [a,b] intervals) as the relation of a point (x,y) instead of an interval [x,y] with respect to a window query $[a,b] \times [a,b]$ instead of reference interval [a,b]. According to this 2-D transformation it's obvious that the interval intersection query can be transformed to the quadrant range query $(-\infty, b] \times [a, +\infty)$ as the result of the geometric union of the 2-d transformations (right side of figure 6) of the four possible cases depicted in the figure above. We answer the quadrant range query (or 2-sided range query) in optimal O(k) time using either the Cartesian Tree ([Gabow, 1984]) or the Modified Priority Search Tree (MPST) ([Kitsios, 2000]). As we will prove theoretically the solution presented in [Kitsios, 2000] is better. The theoretical comparison between the Cartesian Tree and the Modified Priority Search Tree assumes that the dominant operation is a table lookup. This is a valid assumption since both solutions given in [Gabow, 1984] and [Kitsios, 2000] make heavy use of table lookups.

7.1 1st Solution (Cartesian Tree)

Lemma 3: *The Cartesian tree is an optimal quadrant (and three-sided) searching data structure and requires* $\boldsymbol{T(k)=2+2kt_{lca}}$ *table lookups, where k the size of the answer and* $\boldsymbol{t_{lca}}$ is the number of table lookups in order to perform a lowest common ancestor computation.

Proof: Let $S=\{(x_1,y_1), (x_2,y_2), \ldots,(x_N,y_N)\}$ be the set of stored points , with $x_1 \leq x_2 \leq \ldots \leq x_N$. A Cartesian tree T for S is a binary tree of N nodes with each node being labeled by a point in S. The tree T is defined recursively as follows: the root is labeled with (x_m,y_m), where $y_m=\min\{y_i | i=1,\ldots,N\}$. Its left subtree is a Cartesian Tree for $\{(x_i,y_i) | i=1,\ldots,m-1\}$ and its right subtree is a Cartesian Tree for $\{(x_i,y_i) | i=m+1,\ldots,N\}$. Let $p(v)$ be the point labeling the node v of T, $p_x(v)$ be its x-coordinate and $p_y(v)$ be its y-coordinate. The structure makes also use of an array A of size M (M denotes the universe size), which stores pointers to the nodes of T. Specifically $A[i]$ contains a pointer to the node of T which is labeled with the point in S having maximum x-coordinate smaller or equal to i. The crucial property of the above construction is that given two nodes u,w of T with $p_x(u)=x_i$, $p_x(w)=x_j$, $i<j$, then $p_y(\text{nca}(u,w))=\min\{y_t \mid i \leq t \leq j\}$.

So given a query range of the form $(-\infty, b] \times [a, +\infty)$ we firstly use the array A to locate the two nodes u,w of T with $A[-\infty]=u$ *(that means the leftmost leaf)* and $A[b]=w$. As a consequence we need 2 table lookups. Let $z=\text{lca}(u,w)$ with $p(z)=(x_i,y_i)$. If $y_i>c$ then the query algorithm halts, otherwise we report the point (x_i,y_i) and we continue recursively the same way, by probing $\text{lca}(u,u_1)$ and $\text{lca}(u_2,w)$ consuming other two table lookups, where u_1 is the predecessor (in symmetric order) of z in T (that is the point that is stored in u_1 is $p(u_1)=(x_{i-1},y_{i-1})$) and u_2 is the successor (in

symmetric order) of z in T (that is the point that is stored in u_2 is $p(u_2)=(x_{i+1},y_{i+1})$). Thus, the time complexity of the procedure above (in terms of table lookups) is $\boldsymbol{2+2kt_{lca}}$. In order to compute the lowest common ancestor of two nodes in T, we will use the algorithm described by Schieber and Vishkin ([Gusfield, 1994],[Schieber, 1988]) (this algorithm simplifies the approach described in Harel and Tarjan ([Harel, 1984])).

7.2 2nd Solution ([Kitsios, 2000])

Let's remember the most basic components of the Modified Priority Search Tree (MPST) (see fig.7) presented in [Kitsios, 2000]. We used an array A of size M, which stores pointers to the leaves of a classic priority search tree of McCreight T. Specifically $A[i]$ contains a pointer to the leaf of T with maximum x-coordinate smaller or equal to i. With this array we can determine in $O(1)$ time the leaf of any search path P_i for i (in our case the search path P_b for b). In each leaf u of the tree with x-coordinate i we store the following lists $L(u)$, $P_L(u)$: The list $L(u)$ stores the points of the nodes of L_i *(Left children of nodes belonging on* P_i *)*. The list $P_L(u)$ stores the points of the nodes of P_i which have x-coordinate smaller or equal to i. Both lists also contain pointers to the nodes of T that contain these points. Lists $L(u)$ and $P_L(u)$, stores its nodes in increasing y-coordinate of their points.

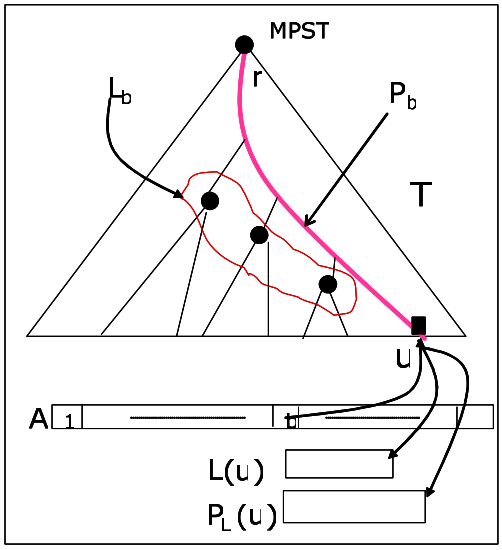

Figure 7: The Modified Priority Search Tree

To answer a query with a range $(-\infty, b] \times (-\infty, c]$ we find in $O(1)$ time the leaf u of the search path P_b for b. Then we traverse the list $P_L(u)$ and report its points until we find a point with y-coordinate greater than c. We traverse the list $L(u)$ in the same manner and find the nodes of L_b whose points have y-coordinate less than or equal to c. For each such node we report its point and then we continue searching further in its subtree as long as we find points inside the range. The size of each list is $O(IogN)$ and

the space of *T* is *O(NlogN).* The space of the whole structure is *O(M+NlogN)* because of the size of the array *A* but making the lists partially persistent (for details see [Kitsios, 2000]) the space becomes linear O(M+N).

Lemma 4: *The MPST structure answers quadrant range queries of the form (-∞, b] × [a, +∞) consuming* ***T(k)=3+k table*** *lookups.*

Proof: Our query algorithm firstly locates the leaf *u* that corresponds to the value *b (*1 table lookup) and then traverses the persistent lists $P_L(u)$ *(*1 table lookup) and *L(u)* (1 table lookup) in order to extract a number of k_1 nodes whose points have *y*-coordinate less than or equal to *c* and for each such node continues searching further in its subtree as long as it finds k_2 points inside the range,where $k_1+k_2=k$ the size of the answer. The theorem follows.

8 The External Modified Priority Search Tree (EMPST)

In this subsection we explain how we can convert the 2nd main memory solution presented in previous section (fig.7) in external memory model of computation (fig.8).

Each point of S is stored in a leaf of T and the points are in sorted x-order from left to right. The x-ordered points are organized in a block fashion.

Each internal node v of T has O(B) fan-out, where B is the block size, and stores a point p(v) of S. The point p(v) is the point with the minimum y-coordinate amongst the points stored in the leaves of T_v.

Each node v is equipped with a secondary list S(v). S(v) contains the points stored in the leaves of T_v in increasing y-coordinate and in a block fashion.

In a similar and straight forward way of that presented in figure 7, we enhance the base virtual tree T with the Array A, the List L(u) and the list $P_L(u)$. Due to external memory model of computation we have to organize the array A [0,..M-1] and the corresponding lists for the leaf_block u, L(u) [0,..,O(logN)] and $P_L(u)$ [0,..,O(logN)] respectively, with consecutive blocks of size B. Thus, the length of the array A becomes M/B and the lengths of the lists become O(logN/B). Due to the fact that we have O(N/B) leaf blocks, the space of the whole structure becomes $O((N/B)\log_B N+M/B)$. The $O(N/B \log_B N)$ term in the space bound is due to the size of the lists $P_L(u)$ and *L(u).* Note that the height of T is $O(\log_B N)$. We can reduce the total space of these lists to *O(N/B)* by making them partially persistent. We show how to implement the lists $P_L(u)$ using a partially persistent list. Let *u* be a leaf_block in *T* and let *w* be its predecessor (the leaf_block on the left of block_*u*). We denote by x_u the left-most *x*-coordinate of block *u* and by x_w, the left-most *x*-coordinate of block *w.* The two root-to-leaf paths Px_u, Px_w, share the nodes from the root of *T* to the nearest common ancestor of blocks *u* and *w*. So we can create $P_L(u)$ by updating $P_L(w)$ in the following way: First we delete from list $P_L(w)$ the points that don't lie on Px_u. Then we insert the points of Px_u which have *x*-coordinate smaller or equal to x_u. In this way we can construct all lists as versions of a persistent list: we begin from the leftmost leaf and construct the list $P_L(u)$ of each leaf *u* by updating the one of its predecessor (see Fig.9). The total number of insertions and deletions is *O(N/B)* because each point

is inserted and deleted only once. Therefore the space of all the lists is *O(N/B)*. In the same way we can construct the lists *L(u)* for all leaves in *O(N/B)* space.

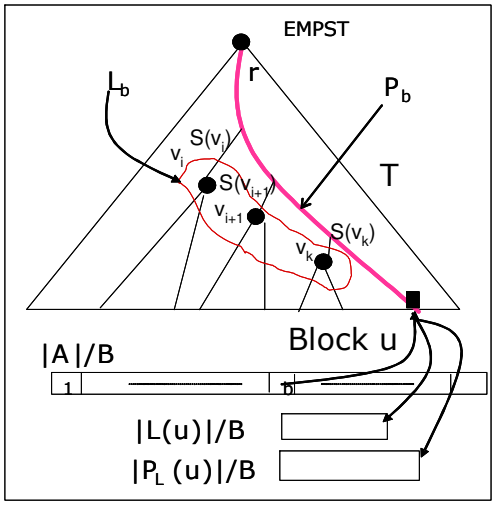

Figure 8: The External Modified Priority Search Tree equipped with the appropriate external arrays and lists

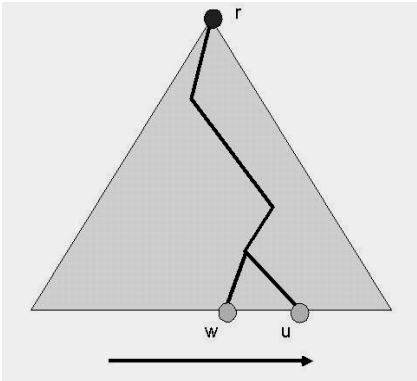

Figure 9: We implement the lists $P_L(u)$ and L(u) as partially persistent lists by performing a sweep from left to right

The query algorithm finds the k_1 points of nodes of P_b that lie inside the query-range in $O(k_1/B)$ time and the k_2 points of *nodes v of* L_b that lie inside the query-range in $O(k_2/B)$ time by simple traversals of the blocks of the y-ordered lists $P_L(u)$ *and* $L(u)$ respectively. The search in the corresponding subtrees T_v*'s of* L_b takes $O(k_3/B)$ additional time for reporting $k=k_1+k_2+k_3$ points in total, by simple traversals of the blocks of the secondary lists S(v)'s. Therefore the query algorithm takes $O(k/B)$ time and more precisely 3+k/B block transfers according to Lemma 4.

9 Experimental Evaluation

We have conducted an experimental study making the customary assumption that the page size is 4096 bytes, the length of each key is 8 bytes, and the length of each pointer is 4 bytes. Consequently, each block contains B=341 elements. We considered data sets of size N∈ $[10^3,10^5]$, where N the number of multimedia objects. We generated synthetic occurrences of multimedia objects. In particular, for each object the time occurrences (the ends of intervals in time axis) draw a Gaussian distribution. In a pre-processing step we transformed each occurrence of a multimedia object (time interval) to a 2-dimensional point. Then we associated each point with the respective multimedia object. We conducted a variety of video function queries FindOinV (v, F_i ,F_j), where the parameters F_i and, F_j draw a Gaussian distribution too. The specific variety of the parameters above generated queries where the mean value of its respective answer K varies between 1000 and 10000 objects.

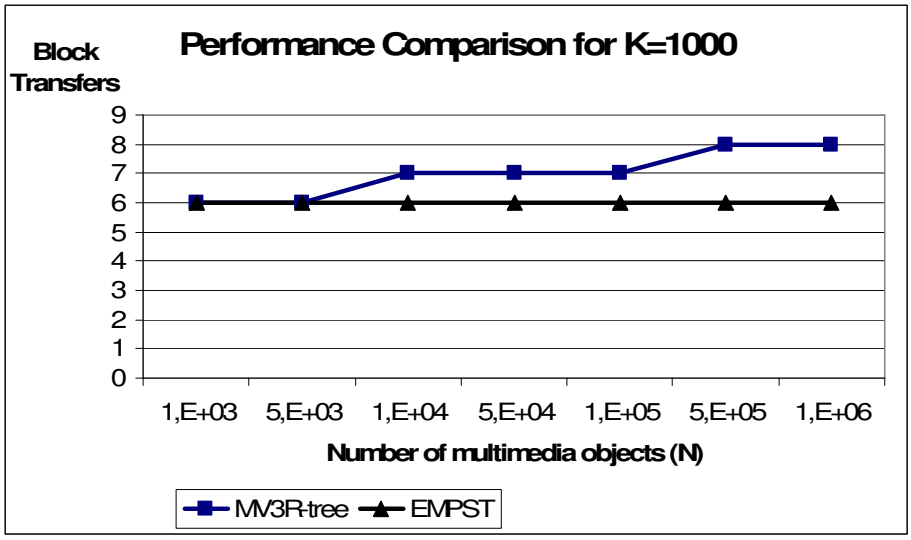

Figure 10: Performance comparison between MV3R-tree and EMPST-tree for K=1000

We compared the implementation of our solution (EMPST) with that of the best current solution MV3R-tree on the same data sets. The implementation of EMPST was carried out in C++ including particular libraries from LEDA v4.1. The source code of MV3R-tree was found in the following URL: http://www.rtreeportal.org/code.html.

Our main concern was to measure the performance, simulated block transfers (I/Os), of video function queries. Our solution outperforms the best current solution, since EMPST requires a constant number of 3 I/Os for locating the appropriate array and list positions instead of a logarithmic number of block transfers required for

locating the appropriate node paths of MV3R-tree. Both solutions require additionally (K/B) I/Os for the total answer selection process.

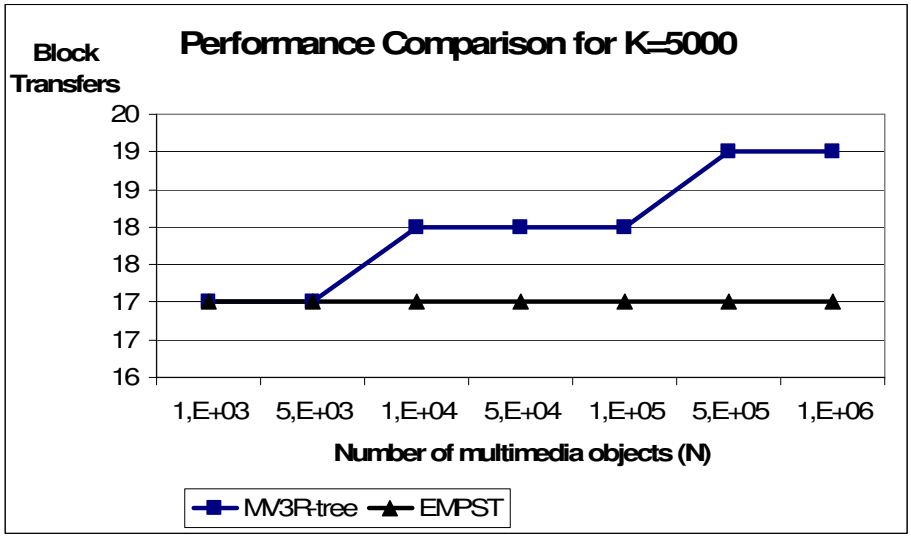

Figure 11: Performance comparison between MV3R-tree and EMPST-tree for K=5000

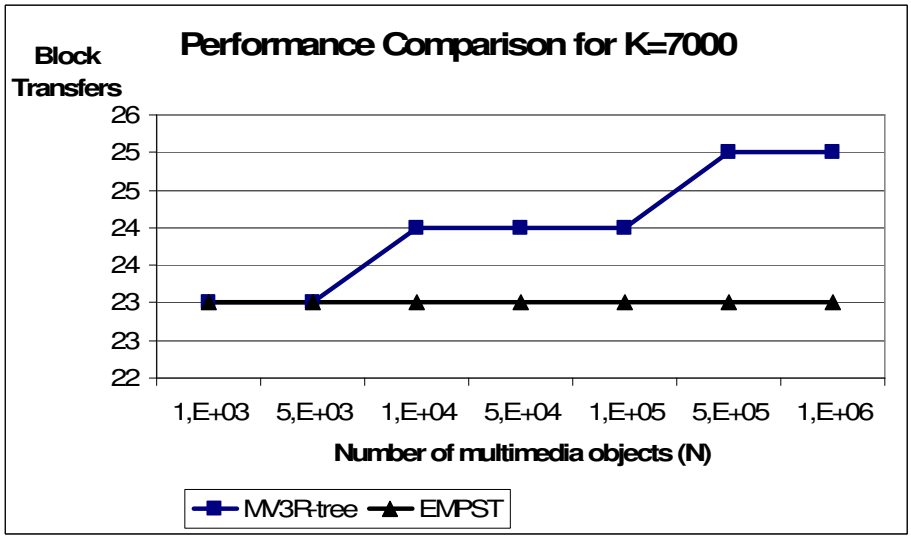

Figure 12: Performance comparison between MV3R-tree and EMPST-tree for K=7000

Figures 10, 11 and 12 depict that EMPST access method requires a constant number of block transfers (6, 17 and 23 I/O's respectively) while the performance of MV3R-tree is increased logarithmically. Figure 13 depicts the number of required Block Transfers as a function of answer size (K). In particular, we conducted a big variety of video function queries where the parameter K varies between 1000 and 10000. As a conclusion, concerning time interval queries our method outperforms the best previous access method.

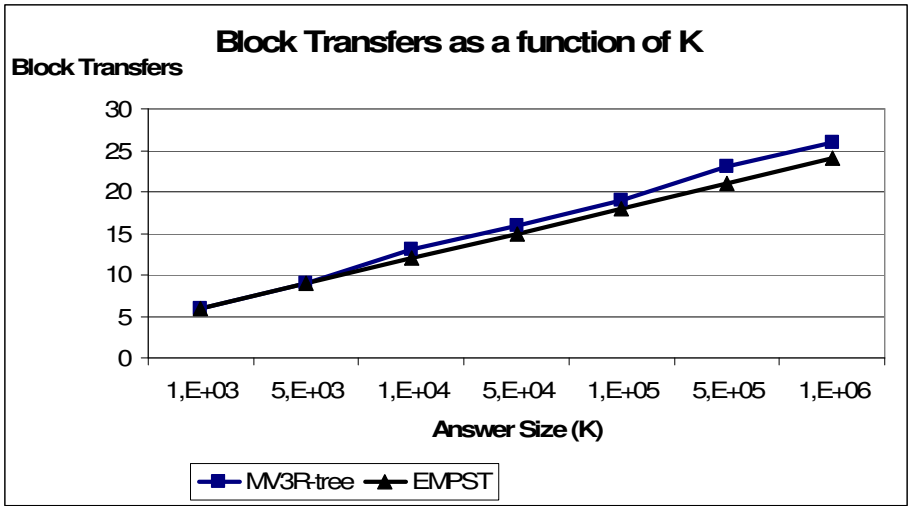

Figure 13: Block Transfers as a function of answer size K

9 Conclusion

We have discussed in this paper a way to store video metadata in order to perform efficiently time interval queries. Metadata specifies either objects or activities in the video. The solutions we suggest refer to both main and secondary memory model of computation and perform efficiently special video functions involving time interval queries. About external memory solution, an experimental evaluation is also included that shows the performance, scalability and efficiency of our method.

References

[Anderson, 1996] Anderson, A. Faster deterministic sorting and searching in linear space. *In Proc. of 37th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, 1996.

[Arge, 1999] L. Arge, V. Samoladas, J.S. Vitter. Two dimensional indexability and optimal range search indexing. *In Proceedings of the ACM Symposium Principles of Database Systems (PODS)*, 1999.

[Beam, 1999] Paul Beam and Faith Fich. Optimal Bounds for the Predecessor Problem. *In Proceedings of the Thirty First Annual ACM Symposium on Theory of Computing (STOC)*, Atlanta, GA, May 1999.

[Bentley, 1977] Bentley J.L. *Solution to Klee's Rectangle Problem.* Carnegie-Mellon Univ., Dept. of Computer Science, unpublished notes, 1977.

[Dimitrova, 2002] Dimitrova, N., Zhang, H.J. Shahraray, B., Sezan, I., Huang, T., Zakhor, A. Applications of Video-Content Analysis and Retrieval, IEEE MultiMedia, 09(3): 42-55, 2002.

[Gabow, 1984] H. N. Gabow, J. L. Bentley, and R. E. Tarjan, Scaling and related techniques for geometry problems, *in Proceedings, l6th Annual ACM Symp. on Theory of Computing, 1984, pp. 135-143.*

[Gusfield, 1994] D.Gusfield, Algorithms on Strings, Trees and Sequences, *Computer Science and Computational Biology*, Cambridge University Press,1994.

[Gutierrez, 2005], Gilberto A. Gutierrez, Gonzalo Navarro, Andrea Rodrıguez, Alejandro Gonzalez, Jose Orellana, A Spatio-Temporal Access Method based on Shnapsots and Events, ACM GIS, November 4-5, 2005, Berlin, Germany.

[Guttman, 1984] A. Guttman. R-Trees: A Dynamic Index Structure for Spatial Searching. In Proc. of the ACM Intl. Conf. on Management of Data, SIGMOD, pages 47–57, June 1984.

[Harel, 1984] D. Harel, R. E. Tarjan, Fast algorithms for finding nearest common ancestor, *SIAM J. Comput.13 (1984), pp. 338-355.*

[Kitsios, 2000] Kitsios N., Makris C., Sioutas S. Tsakalidis A., Tsaknakis J., Vassiliadis B., 2-D Spatial Indexing in Optimal Time, in Proc. Current Issues in Database and Information Systems, ADBIS-DASFAA 2000, Springer Verlang

[Kosch, 2005] Kosch, H., Boszormenyi, L., Doller, M., Libsie, M., Schojer, P., Kofler, A., The Life Cycle of Multimedia Metadata," IEEE MultiMedia, 12(1): 80-86, 2005.

[Madhwacharyula, 2006] Madhwacharyula CL., Davis, M., Mulhem, P., Kankanhalli, M.S. Metadata handling: A video perspective. ACM Transactions on Multimedia Computing, Communications, and Applications (TOMCCAP), 2(4): 358 – 388, 2006.

[Manolopoulos, 1990] Manolopoulos, Y., Kapetanakis, G. Overlapping B+trees for Temporal Data. *JCIT*, 1990.

[Mark de Berg, 1997] Mark de Berg, Marc van Kreveld, Mark Overmars, Otfried Schwarzkopf. *Computational Geometry: Algorithms and Applications.* Springer-Verlag, Heidelberg, 1997.

[Mehlhorn, 1984] K. Mehlhorn. *Data Structures and Algorithms 3: Multi-dimensional Searching and Computational Geometry.* EATCS Monographs on Theoritical Computer Science, Springer-Verlag, 1984.

[Nack, 2005] Nack, F., van Ossenbruggen, J., Hardman, L. That Obscure Object of Desire: Multimedia Metadata on the Web, Part 2," IEEE MultiMedia, 12(1): 54-63, 2005.

[Nack, 2000] Nack, F. All Content Counts: The Future in Digital Media Computing is Meta, *IEEE MultiMedia,* 7(3): 10-13, 2000.

[Nascimento, 1998] Nascimento, M., Silva, J. Towards Historical R-trees. *ACM SAC*, 1998.

[Preparata, 1985]F.P. Preparata, M.I. Shamos. *Computational Geometry: An introduction,*. Springer-Verlag, New York, 1985.

[Samet, 1989] H. Samet. *The Design and Analysis of Spatial Data Stuctures.* MA: Addison-Wesley, 1989.

[Schieber, 1988] B. Schieber, U. Vishkin. On finding lowest common ancestors: simplifications and parallelization. *SIAM J. of Comput*., 17:1253-62, 1988.

[Subramanian, 1998] V.S. Subrahmanian. *Principles of Multimedia Database Systems.* Morgan Kaufmann Publishers Inc., pp. 179-213, 1998.

[Tao, 2001] Y. Tao, D. Papadias: MV3R-Tree: A Spatio-Temporal Access Method for Timestamp and Interval Queries. VLDB 2001: 431-440

[Willard, 1992] Dan E. Willard. Applications of the fusion tree method for Computational Geometry and searching. *In Proc. 3rd Symposium on Discrete Algorithms (SODA)*, pp. 286-296, 1992.

[Vazirgiannis, 1998] Vazirgiannis M., Theodoridis, Y., Sellis, T. Spatio•Temporal Composition and Indexing for Large Multimedia Applications. *ACM Multimedia Systems*, 6(5), 1998.

[Xu, 1990] Xu, X., Han, J., Lu, W. RT-tree: An Improved R-tree Index Structures for Spatiotemporal Data. *SDH*, 1990.