Polytechnic
UNIVERSITY
Brooklyn · Long Island · Westchester

INDEXING PROBLEMS IN SPATIOTEMPORAL DATABASES

George N. Kollios

Department of Computer and Information Science

Technical Report
TR-CIS-2000-05
06/23/2000

INDEXING PROBLEMS IN SPATIOTEMPORAL DATABASES

D I S S E R T A T I O N

for the Degree of

Doctor of Philosophy (Computer Science)

George N. Kollios

June 2000

INDEXING PROBLEMS IN SPATIOTEMPORAL DATABASES

DISSERTATION

Submitted in Partial Fulfillment

of the Requirements for the

Degree of

DOCTOR OF PHILOSOPHY (Computer Science)

at the

POLYTECHNIC UNIVERSITY

by

George N. Kollios

June 2000

Approved :

____

Department Head

Copy No. ________________ ____________________, 2000

Approved by the Guidance Committee :

Major : Computer Science

Alex Delis

Assistant Professor of
Computer Science

Vassilis Tsotras

Associate Professor of
Computer Science - UC Riverside

Phyllis Frankl

Associate Professor of
Computer Science

Lisa Hellerstein

Associate Professor of
Computer Science

Minor : Electrical Engineering

Shivendra Panwar

Associate Professor of
Electrical Engineering

Microfilm or other copies of this dissertation are obtainable from

UNIVERSITY MICROFILMS
300 N.Zeep Road
Ann Arbor, Michigan 48106

VITA

George N. Kollios was born in February 1972 in Preveza, Greece. He received his Diploma in Electrical and Computer Engineering from the National Technical University of Athens, Greece, in 1995 and his Master in Computer Science from Polytechnic University in Brooklyn, New York, in 1998. Since January 1998, he has been working towards his Ph. D. degree in Computer Science at Polytechnic University in Brooklyn, New York. His research interests are in spatial and temporal databases and data mining.

The work presented in this thesis was performed between September 1996 and May 2000 under the supervision of Professor Vassilis Tsotras of University of California at Riverside (Advisor) and Professor Alex Delis of Polytechnic University (Co-Advisor).

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to my thesis advisor Vassilis Tsotras and co-advisor Alex Delis. Without their assistance and guidance I would not been able to complete this dissertation. Vassilis has been very patient and generous with his time and a true source of inspiration throughout the course of this dissertation. Alex has been very supportive during my two years at Poly and especially in the last two years that I spent between Poly and UC Riverside.

I would like to extend my sincerest thanks to professor Dimitrios Gunopulos for his help and advice during my two years in Riverside. I would like to thank professors Phyllis Frankl, Lisa Hellerstein and Shivendra Panwar for serving on my thesis committee and for their valuable comments on my dissertation. I would also like to thank all faculty members in the CS departments at Polytechnic University and UC Rivreside, as well as my undergraduate professors Stathis Zachos and Timos Sellis from NTUA in Greece, for their support and help.

My colleagues in the Database Lab at Polytechnic University and in the Database Lab at UC Riverside always have been supportive and ready to help me. Special thanks to Michalis Loulakis and Yiannis Veremis for freely offering me their help and friendship. Also, I would like to thank Michalis Faloutsos, George Lapiotis, Symeon Pappavasileiou and Nick Koudas.

Finally, I would like to share a great deal of my achievement with my parents and my brother. Without their love, encouragement and support, this research could not have been completed. This dissertation is dedicated to them.

ABSTRACT

INDEXING PROBLEMS IN SPATIOTEMPORAL DATABASES

by

George N. Kollios

Advisor: Vassilis Tsotras

Co-Advisor: Alex Delis

Submitted in Partial Fulfillment of the Requirements for the
Degree of Doctor of Philosophy (Computer Science)

June 2000

Spatiotemporal databases manage spatial objects that change positions and/or extents over time. Examples include traffic surveillance data, climate and land cover data, demographic data and multimedia applications (animated movies). Since these databases are large in size, it is important to design efficient indexing schemes that can access and explore them.

We first study the problem of indexing spatial objects that have evolved in the past and the whole evolution of each object is known. We present methods to store these evolutions in external memory in order to answer efficiently range and nearest neighbor queries of the form: "find the objects that were in area S between time instants t_i and t_j", or "find the q nearest objects to a given position between time instants t_i and t_j". We reduce the problem to a partial-persistence one, that is, we use a 2-dimensional access method that is made partially persistent. We show that this approach leads to fast query time while still using space proportional to the total number of changes in the spatiotemporal evolution. What differentiates this problem from traditional temporal indexing approaches is that objects are allowed to move and/or change their extent continuously over time. We present novel methods to approximate such objects. We then, formulate the problem

as an optimization problem for which we provide an efficient and effective solution for the case where objects move linearly. An extensive experimental study demonstrates the advantages of our approach over other straightforward solutions.

While the above study concentrates on historical queries (past states) of spatiotemporal data, of interest are also queries about the future behavior of such data. Here, we assume that the objects movement/change functions are known. We show how to index mobile objects in one and two dimensions using efficient dynamic external memory data structures. Our approach is to employ methods to store the motion function of each object and answer range and nearest neighbor queries using these methods.

Finally, we present a solution to the temporal membership problem which is likely to occur in temporal and spatiotemporal databases. Consider a set of objects that evolves over time by adding and deleting objects and these changes are timestamped. A temporal membership query asks whether a given object was in the set at a specific time instant. We present methods that use partially persistent hashing schemes to answer efficiently this type of queries. The proposed methods have linear space and perform better than other approaches in practice.

Contents

List of Figures 6

List of Tables 9

1 Introduction 10

1.1 External Memory Model 12

1.2 Spatial and Temporal Indexes 12

1.2.1 R-Tree 12

1.2.2 The Snapshot Index 14

1.3 Organization of the Thesis 16

2 Indexing Historical Spatiotemporal Evolutions 18

2.1 Introduction 18

2.2 Background 22

2.2.1 Degenerate Case 22

2.2.2 Partially-Persistent R-Tree 24

2.3 General Case 26

2.3.1 A Greedy Algorithm 34

2.3.2 A Dynamic Programming Algorithm for the General Case 37

2.4 Performance Evaluation 37

2.4.1 Experimental Setup 38

2.4.2 Tuning the PPR-Tree 40

2.4.3 Degenerate Case 42

2.4.4 General Case 43

2.5 Related Work in Spatiotemporal Indexing 46

2.6 Summary 48

3 Indexing the Future Positions of Continuously Moving Objects 52

3.1 Introduction 52

3.2 Background 54

3.3 Indexing in one dimension 54

3.3.1 Space-time representation 55

3.3.2 The dual space-time representation. 56

3.3.3 Lower Bounds 59

3.3.4 An (Almost) Optimal Solution 60

3.3.5 Improving the average query time. 61

3.3.6 Achieving Logarithmic Query Time 65

3.4 Indexing in two dimensions 69

3.4.1 The 1.5-dimensional problem 69

3.4.2 The 2-dimensional problem 70

3.5 A Performance Study 72

3.6 Related Work in Indexing Continuously Moving Objects 73

3.7 Summary 74

4 Temporal Hashing in Temporal and Spatiotemporal Databases 76

4.1 Introduction 76

4.2 Background 78

4.2.1 Linear Hashing 79

4.3 Partially Persistent Linear Hashing 81

4.3.1 The Evolving-Set Approach 81

4.3.2 The Evolving-List Approach 88

4.4 Performance Evaluation 90

4.4.1 Experimental Setup 90

4.4.2 Experiments 94

4.4.3 Summary 99

5 Summary and Future Research **107**

Bibliography **110**

List of Figures

1.1 An example of a 2 dimensional R-Tree 13

2.1 A conceptual view of a spatiotemporal evolution. 18

2.2 A degenerate spatiotemporal evolution of four objects. 23

2.3 *A spatiotemporal object.* 27

2.4 A split operation of a 1-dimensional object (interval) that moved continuously from t_1 to t_2. 28

2.5 A 1-dimensional moving object with one and two splits. 31

2.6 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. 32

2.7 The spatiotemporal object created by a 1-dimensional moving point and the gain after one and two splits. 33

2.8 The Optimal GREEDY algorithm 35

2.9 Query performance for different merging/splitting policies. 40

2.10 Space consumption for different merging/splitting policies. 40

2.11 Query performance for small/ snapshot queries and DG datasets. 42

2.12 Query performance for medium/ snapshot queries and DG datasets. 42

2.13 Query performance for large/ snapshot queries and DG dataset. 43

2.14 Query performance for time period queries and DG dataset. 43

2.15 Space consumption for DG datasets. 44

2.16 Query performance for small/ snapshot queries and MV datasets. 44

2.17 Query performance for medium/ snapshot queries and MV datasets. 45

2.18 Query performance for large/ snapshot queries and MV datasets. 45

2.19 Space consumption for MV datasets. 46

2.20 Query performance for time period queries and MV datasets. 46

2.21 Query performance of the Greedy-PPR-tree for snapshot queries and different number of splits and MV datasets. 47

2.22 Query performance for small/ snapshot queries and GN datasets. 47

2.23 Query performance for medium/ snapshot queries and GN datasets. 48

2.24 Query performance for large/ snapshot queries and GN datasets. 48

2.25 Query performance of the Greedy-PPR-Tree for snapshot queries and different number of splits and GN datasets. 49

2.26 Space consumption for the GN datasets. 49

2.27 Query performance for time period queries and GN dataset. 50

2.28 Nearest Neighbor query performance for GN datasets. 50

2.29 Nearest Neighbor query performance for different time periods and GN datasets. . 50

2.30 Construction cost for MV dataset. 51

2.31 Construction cost for GN dataset. 51

3.1 Trajectories and query in (t, y) plane. 55

3.2 Query in the dual Hough-X plane. 57

3.3 Data regions for R-tree like and kd-tree like methods 63

3.4 Query in the dual Hough-Y plane. 63

3.5 Trajectories and query in (x, y, t) plane. 70

3.6 Query Performance for 10% Queries. 71

3.7 Query Performance for 1% Queries. 71

3.8 Space Consumption. 73

3.9 Update Performance. 73

4.1 Two instants in the evolution of an ephemeral hashing scheme. (a) Until time t = 20, no split has occurred and p = 0. (b) At t = 21, oid 8 is mapped to bucket 3 and causes a controlled split. Bucket 0 is rehashed using h_1 and p = 1. 83

4.2 The detailed evolution for set S until time t = 25 (a "+/-" denotes addition/deletion respectively). Changes assigned to the histories of three buckets are shown. The hashing scheme of Figure 4.1 is assumed. Addition of oid 8 in S at t = 21, causes the first split. Moving oid 15 from bucket 0 to bucket 5 is seen as a deletion and an addition respectively. The records stored in each bucket's history are also shown. For example, at t=25, oid 10 is deleted from set S. This updates the lifespan of this oid's corresponding record in bucket 0's history from $< 10, [1, now) >$ to $< 10, [1, 25) >$. 101

4.3 (a) Query, (b) Update, and, (c) Space performance for all implementations on a uniform workload with 8K oids, $n \sim 0.5M$ and $\bar{NB} \sim 30$. 102

4.4 (a) Query, (b) Update, and, (c) Space performance for ALH, PPLH-s, PPLH-l, MVBT and R_i methods using the exponential, step, normal and poisson workloads with 8K oids, $n \sim 0.5M$ and $\bar{NB} \sim 30$. 103

4.5 (a) Query, (b) Update, and, (c) Space performance for ALH, PPLH-s, PPLH-l, MVBT and Ri methods using various uniform workloads with varying $\bar{NB}$. . . . 104

4.6 (a) Query, (b) Update, and, (c) Space performance for ALH, PPLH-s, PPLH-l, MVBT and Ri methods using various uniform workloads with varying $|U|$. 105

4.7 (a) Query, (b) Update, and, (c) Space performance for PPLH-s on a uniform workload with varying values of the usefulness parameter u. 106

List of Tables

2.1 The datasets used for testing the index structures. 39

4.1 Uniform Datasets. 97

4.2 Datasets with different number of oids. 98

4.3 Performance comparison of PPLH-s (controlled splits) versus PPEH-s (uncontrolled splits). 99

Chapter 1

Introduction

Database systems that manage spatial and temporal objects have received increasing interest in recent years. Databases that store spatial objects that change their extent and/or their position over time are called *spatiotemporal databases.* In these databases, the current, the past, as well as the anticipated future positions and extents of the objects are frequently of interest. Applications that have to deal with spatiotemporal objects include global change (climate or land cover data), transportation (traffic surveillance), social (demographic, health) and multimedia (animated movies). Next we describe in more detail two specific examples of spatiotemporal applications.

In the first example, we consider a database that manages moving vehicles (cars) in a highway system. Recent advances in communications and GPS (Global Positioning System) technology allow people and vehicles to locate themselves at any position on earth, with high accuracy. Consider a database that (using this technology) stores the current positions of moving vehicles, as well as their direction and their speed. A number of interesting queries can be directed to this database. For example, a person may want to find the closest hotel to her car for the next 10 minutes. Also a company that manages trucks, may need to find the closest truck to a specific warehouse. Or, in case of a traffic accident we may want to find the closest ambulances and the closest hospital to the accident.

Another example of a spatiotemporal application comes from multimedia databases and in

particular from systems that store animated movies. A 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.) 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. Thus, a user may be interested in asking 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 objects that appear closest to a given position A during frames f_i and f_j". S and A are part of the 2-dimensional frame screen. For example the movie editor may want to find all the objects that are inside a given window region in a sequence of consecutive frames.

Due to the time component, spatiotemporal databases need to manage large amounts of data accumulated over long period of time. A user asks queries over this data and the straightforward solution to find the answer is to read all objects in the database and return the objects that belong to the answer. However this approach is inefficient due to the size of the database. A better solution is to construct indexes over the data and answer a query by reading only a small part of the database. In general, an index is a way to organize a dataset in disk pages in order to answer efficiently a specific type of queries, by reading only a small number of disk pages.

In this thesis, we present methods to answer efficiently range, nearest neighbor and membership queries in spatiotemporal databases. In a range query, we specify a region and a time interval and the system has to find the objects that overlap or contained inside the region during the specified time interval. Similarly, in a nearest neighbor query, we specify a location is space and we are interested in the objects that are close to this location during a time interval. Finally, in a membership query, we specify an object and a time instant and we would like to know if the object is in the database at the specified time instant.

1.1 External Memory Model

We consider our problems in the standard external memory model of computation[AV88]. In this model each disk access (an I/O) transmits in a single operation B units of data. We call B the page capacity. We measure the efficiency of an algorithm in terms of the number of I/O's to perform an operation. If N is the number of the objects in the database and K is the number of objects reported by a query, then the minimum number of pages to store the database is $n = \lceil \frac{N}{B} \rceil$ and the minimum number of I/O's to report the answer is $k = \lceil \frac{K}{B} \rceil$. We say that an algorithm uses linear space, if it uses $O(n)$ disk pages, and that it uses logarithmic time to answer a query if it needs to execute $O(\log_B n + k)$ I/O's. Note that $\log_B n$ is for the external memory model different than $\log_2 n$ since B is not a constant but a problem variable.

Our goal is to minimize the number of pages that are used to store a database and at the same time minimize the number of pages that we need to access in order to answer a query.

1.2 Spatial and Temporal Indexes

During the last two decades, a large number of index methods for spatial and temporal databases have been proposed. For an excellent survey in spatial access methods we refer to [GG98] and for temporal indices to [ST99]. Next we describe two very important access methods in spatial and temporal databases. In particular, we present the R-tree, an external memory spatial access method and the Snapshot Index, a very efficient temporal index.

1.2.1 R-Tree

An R-tree is a hierarchical, height-balanced external memory data structure proposed by Gutman in [Gut84]. 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). An MBR of an object is the smallest rectangle that encloses the object and has its sides parallel to the axes.

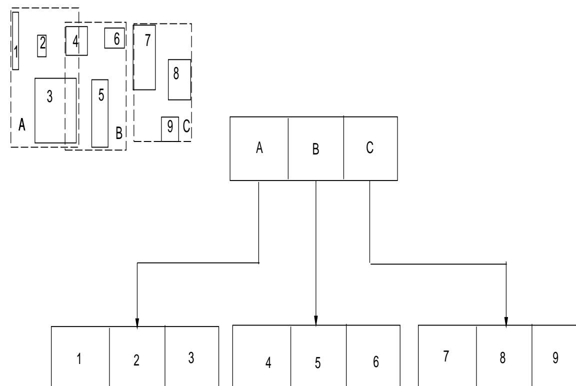


Figure 1.1: An example of a 2 dimensional R-Tree .

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 (See Figure 1.1).

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[KF94] and the R*-tree[BKS+90]. Another variation (R+-tree[SRF87]) does not allow overlapping among the MBRs of the same level of the tree.

1.2.2 The Snapshot Index

Before we describe the Snapshot Index, we present a model of a temporal evolution. Consider for simplicity an initially empty set S. As time proceeds, objects can be added to or deleted from this set. When an object is added to S and until (if ever) is deleted from S, it is called "alive". This is represented by associating with each object a semi-closed interval, or lifespan, of the form: $[start_time, end_time)$ [1]. While an object is alive it cannot be re-added in S, i.e. S contains no duplicates. Deletions can be applied to alive objects. When an object is added at t, its start_time is t but its end_time is yet unknown. Thus its lifespan interval is initiated as $[t, now)$, where now is a variable representing the always increasing current time. If this object is later deleted from S, its end_time is updated from now to the object's deletion time. Since an object can be added and deleted many times, objects with the same oid may exist but with non-intersecting lifespan intervals (i.e., such objects were alive at different times). The state of the set at a given time t, namely $S(t)$, is the collection of all alive objects at time t.

The Snapshot Index [TK95] solves the pure-snapshot problem where a time instant t_q is specified and the state of the set $S(t_q)$ has to be retrieved. It uses three basic structures: a balanced tree (time-tree) that indexes data pages by time, a pointer structure (access-forest) among the data pages and an ephemeral hashing scheme. The time-tree and the access-forest enable fast query response while the hashing scheme is used for fast updates.

We first discuss updates. Objects are stored sequentially in data pages in the same order as they are added on set S. When a new object with oid k is added at time t, a new record of the form $< k, [t, now) >$ is created and is appended in a data page. At any given instant there is only one data page that stores (accepts) records, the acceptor (data) page. When the current acceptor page becomes full, a new one is created. The time when an acceptor page was created along with its page address are stored in the time-tree. As acceptor pages are created sequentially the time-tree is easily maintained (amortized $O(1)$ I/O for indexing each new acceptor page). For object additions, the sequence of all data pages resembles a regular log but with two main differences: (1) deletion updates are handled differently, and (2) additional links (pointers) among the data pages exist.

[1] We use brackets ([]) to denote closed intervals and parentheses (()) for open intervals.

Object deletions are not added sequentially; rather they are in-place updates. When object k is deleted at time t', its record is first located and then updated from $< k, [t, now) >$ to $< k, [t, t') >$. Object records are found using their oids through the hashing scheme. When an object is added in S, its oid and the address of the page that stores the object's record are inserted in the hashing scheme. If this object is deleted the hashing scheme is consulted, the object's record is located and its interval is updated. Then this object's oid is removed from the hashing scheme.

Storing only one record for each object suggests that the records of the objects in $S(t)$ may be dispersed in various data pages. Accessing all pages with alive objects at t, would require too much I/O (O(a) pages). To achieve good record clustering a "controlled" copying technique is used that keeps the total space linear to the number of updates. The copying procedure is based on the concept of page usefulness.

Consider the number of "alive" records a page contains after it becomes full. For all times that this page contains uB alive records it is called useful. For these times t the page contains a good part of the answer for $S(t)$. Answering a pure-snapshot query about some time t needs only locate the useful pages at that time; each such page will contribute at least uB objects to the answer. The usefulness parameter u is a constant that tunes the behavior of the Snapshot Index.

Acceptor pages are special. While a page is the acceptor page it may contain fewer than uB alive records. By definition a page is also called useful for as long as it is the acceptor page. Such a page may not give enough answer to justify accessing it but it must still be accessed. Since for each time instant there exists exactly one acceptor page, this does not affect query performance.

Let $[u.start_time, u.end_time)$ denote a page's usefulness period; $u.start_time$ is the time the page started being the acceptor page. When the page gets full it either continues to be useful (and for as long as the page has at least uB alive records) or it becomes non-useful (if at the time it became full the page had less than uB alive records). The next step is to cluster the alive records for each t among the useful pages at t. When a page becomes non-useful, an artificial copy occurs that copies the alive records of this page to the current acceptor page (as in a timesplit [LS89]). The non-useful page behaves as if all its objects are marked as deleted but copies of its alive records can still be found from the acceptor page. Copies of the same record contain subsequent non- overlapping

intervals of the object's lifespan. The copying procedure reduces the original problem of finding the alive objects at t into finding the useful pages at t. The solution of the reduced problem is facilitated through the access-forest.

The access-forest is a pointer structure that creates a logical "forest of trees" among the data pages. Each new acceptor page is appended at the end of a doubly-linked list and remains in the list for as long as it is useful. When a data page d becomes non-useful: (a) it is removed from the list and (b) it becomes the next child page under the page c preceding it in the list (i.e., c was the left sibling of d in the list when d became non-useful). As time proceeds, this process will create trees of non-useful data pages rooted under the useful data pages of the list. The doubly-linked list and the child lists are implemented by using four pointers per page. Hence, each page has one pointer that points to the next page, one pointer to the previous page, one to the first page of its child list and one to the last page of its child list. The next, previous and last child pointers are updated so that they always point to the current pages in the corresponding positions.

The access-forest has a number of properties that enable fast querying. In [TK95] it was shown that starting from the acceptor page at t, all useful pages at t can be found in at most twice as many I/O's (in practice much fewer I/O's are needed). Finding the acceptor page at t requires searching the balanced time-tree (this corresponds to the logarithmic part of the query time). In practice this search is very fast as the height of the balanced tree is small (it stores only one entry per acceptor page which is clearly $O(n/B)$). The main part of the query time is finding the useful pages. The performance of the Snapshot Index can be fine tuned by changing parameter u. Large u implies that acceptor pages become non-useful faster, thus more copies are created which increases the space but also clusters the answer into smaller number of pages, i.e., less query I/O.

1.3 Organization of the Thesis

In this thesis we extend previous spatial and temporal access methods and we propose new methods to index spatiotemporal databases. In the next chapter we present access methods to index historical spatiotemporal data. In chapter 3 we present methods to index objects that move

in one and two dimensions. The goal is to answer efficiently queries about the future positions of the objects provided assumptions about their movement behavior. Finally, in chapter 4 we describe methods to answer the temporal membership query.

Chapter 2

Indexing Historical Spatiotemporal Evolutions

2.1 Introduction

In this chapter we present indexing structures for historical (past) data gathered from a spatiotemporal evolution. A simple type of an evolution is when objects retain their extent and position from the time they are inserted in the dataset to the time they are deleted. We call this type of evolution degenerate and changes in this evolution refer simply to objects additions and deletions.

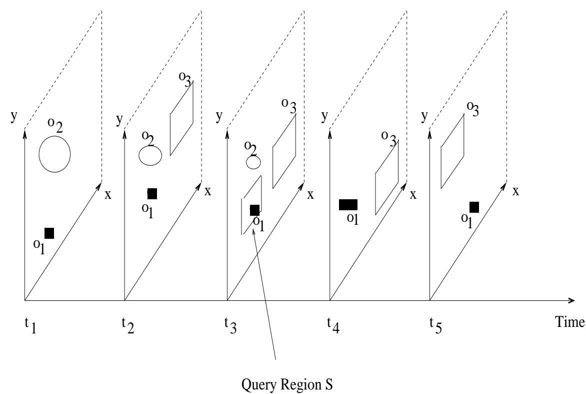


Figure 2.1: A conceptual view of a spatiotemporal evolution.

More interesting (and realistic) is the general case where objects are allowed to move and grow/shrink during their lifetime (Figure 2.1). However, in the general case it is not obvious how position and extent changes can be quantified as object insertions/deletions. Consider for example an object that moves from position A in time instant t_i to a new position C in the next time instant t_{i+1}. The simplest way to represent such movement is to delete the object from position A at t_{i+1} and reinsert it at position C at the same time instant. This creates two records for this object, one record storing position A and whose lifetime ends at t_{i+1} and one record with position C and lifetime starting at t_{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 time as it creates a large number of artificial insertions and thus it increases the index storage space.

A better way is to store the functions describing how objects move or vary their extents [EGS+98]. Even though general functions can be used, for simplicity we assume that objects can move or grow/shrink through a simple (for example *linear*) function of time. Then a new record is inserted only when the parameters describing an object's movement or extent functions change. The new record will maintain the object's lifetime under the new movement/extent function. Then in the general case the number of insertions N 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 time instant t_i, its deletion time is not yet known, so its lifetime is initiated as $[t_i, now)$ where *now* is a variable representing the (ever increasing) current time. If this object gets deleted at a later instant t_j, its lifetime interval is updated to $[t_i, t_j)$. Instead, in the Off-Line mode, we know in advance for each object its insertion and deletion times as well as its positions and extents in between. Obviously, in the Off-Line mode, the constructed index is expected to be more efficient since we have more information about the data. In this chapter we concentrate on the Off-Line mode, since this is the case in most spatiotemporal applications. Note however, that there are some 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.

One straightforward way to index moving/changing objects, is to consider time as another index dimension. Then each object can be stored as a 3-dimensional rectangle in a traditional spatial index (e.g. an R-Tree [Gut84] or its variations [KF94, SRF87, BKS+90]) 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. Since this approach uses an "off-the-self" spatial index, it is straightforward to implement. However, it does not take advantage of the specific properties of the time dimension. Objects remaining unchanged for many time instants will have long lifetimes and thus, they will be stored as long rectangles. A long-lived rectangle determines the length of the time range associated with the page in which it resides. This makes the clustering of objects into pages very challenging and leads to decreased query performance.

In an attempt to overcome the above problems with storing intervals, Kolovson and Stonebraker proposed the Segment R-Tree (SR-Tree) [KS91] which is a variation of the R-Tree. The SR-Tree combines properties of the R-Tree and the Segment Tree, a binary tree data structure that stores line segments [Sam90]. In the SR-Tree, intervals can be stored in both leaf and non-leaf nodes. An interval I is placed to the highest level node X of the tree such that I spans at least one of the intervals represented by X's child nodes. If I does not span X, then I 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, thus the SR-Tree tends to decrease the overlapping in leaf nodes (in the regular R-Tree, a long interval stored in a leaf node will "elongate" the area of this node thus exacerbating the overlap problem). Interval fragmentation implies storing fragments of the same interval in many places. At worst, the space of the SR-Tree is no longer linear (a logarithmic factor is added).

In contrast, we propose to use a different approach in indexing spatiotemporal objects which combines a spatial index (R-Tree) with the partially persistent methodology. A data structure is called *persistent* [DSS+86] 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 case of the problem we address. This is because in the degenerate case an update simply corresponds to an object addition/deletion. Thus, the partially-persistent R-Tree can be easily extended to index the degenerate case of spatiotemporal objects.

However, the general case where objects change continuously is far more challenging to address. One approach is to represent an object's movement or extent change by the largest 2-dimensional MBR that the object obtained at any time instant during its evolution (*maxMBR*). For example, in Figure 2.1 the largest MBR in the evolution of object o_2 occurs at time instant t_1. Then the evolution of o_2 can be represented by the insertion of this MBR at time instant t_1 and the deletion of the *same* MBR at t_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 time advances, it is still represented by the larger MBR. Empty space in R-Trees is known to deteriorate query time.

To reduce 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. In order to maintain the index storage space linear to the number of real evolution insertions 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 time instants the artificial updates are created. We formulate these questions as an optimization problem for which we provide a greedy algorithm that optimally finds the artificial updates for the case where 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, however it serves as a good heuristic that performs very well in practice.

Section 2.2 provides background on the partially persistent R-Tree and the degenerate case. Section 2.3 discusses the general case of animated objects as well as the greedy algorithm. Section 2.4 contains experimental results. Section 2.5 presents related work while 2.6 concludes the chapter.

2.2 Background

There are two obvious but inefficient ways to address topological queries in spatiotemporal evolutions. The first is to store in the database snapshots of the evolution at each time instant. This "snapshot" approach provides fast access to the time of interest, but extra work is needed to locate the objects in the query area S. The main disadvantage however is the high storage space redundancy. Many objects that do not change will be stored several times. At worst, the space can become quadratic to the number of updates in the evolution, i.e., $O(\frac{N^2}{B})$.

The second straightforward approach is to store the changes between time instants in a "log". This approach uses minimal space $O(\frac{N}{B})$, but the query time is rather large as the time of interest has to be reconstructed starting from the beginning of the log (at worst the whole log must be read resulting to $O(\frac{N}{B})$ query time). An intermediate approach is to store a number of snapshots and the sequences of changes between successive snapshots (similar idea as in MPEG[PG97]). However, this approach has the following disadvantages: (i) it is not obvious how often to keep snapshots (frequent snapshots increase storage space, 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 first with a discussion of the degenerate case. We then show how a Partially Persistent R-Tree can be extended to efficiently index this case.

2.2.1 Degenerate Case

In the degenerate case, an object is inserted at some point instant and remains as is until the time it is deleted. A solution for this kind of spatiotemporal evolution has been proposed in [VTS98], that utilizes a 3-dimensional R-Tree. The time dimension is considered as another dimension along with the spatial ones, and an object is represented by its 3-dimensional MBR. Figure 2.2 shows the MBRs of four objects in a degenerate evolution. As mentioned before, with this approach objects that remain unchanged over long time will be stored as records with long lifetimes. The R-Tree will attempt to store these records as long rectangles (like object o_3 in Figure 2.2). causing a lot

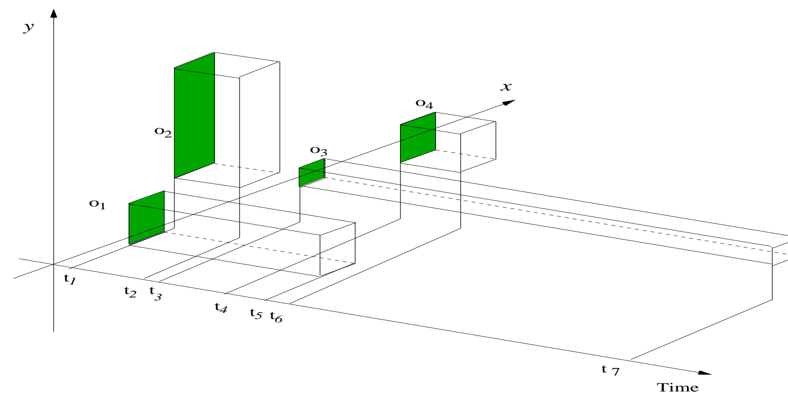


Figure 2.2: A degenerate spatiotemporal evolution of four objects.

of overlapping between the nodes of the R-Tree. However, large overlapping decreases the R-Tree query performance.

The SR-Tree has been proposed as a remedy for storing intervals. Overlapping is decreased by placing long intervals in higher nodes of the structure. However large numbers of spanning records or fragments of spanning records are stored high up in the tree and this decreases the fan-out of the index as there is less room for pointers to children. In [KS91] is suggested 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 adds some page accesses to the search cost.

As with the R-tree, if an interval is inserted at a leaf (because it did not span anything) the boundaries of the MBR covered by the leaf node in which it is placed may be expanded. Expansions may be needed on the MBRs of all nodes on the path to the leaf which contains the new record. This may 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 space requirements for an SR-Tree is $O((N/B)log_B(N/B))$ [ST99]. To improve performance, Kolovson and Stonebraker have also proposed the use of a Skeleton SR-Tree, which is an SR-Tree which pre-partitions the entire domain into some number of regions [KS91]. This pre-partition is based on some initial as-

sumption on the distribution of data and the number of intervals to be inserted. Then the Skeleton SR-Tree is populated with data.

A better approach is to use partial persistence. Considering the degenerate evolution of Figure 2.2, assume that the objects at time t_1 are indexed by a 2-dimensional R-Tree. As the time advances, this 2D R-Tree evolves, by applying on it the updates (object additions/deletions) as they occur at the appropriate time instants. 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 disassociate the indexing requirements within a time instant from the evolution. More specifically, indexing the objects at a time instant is provided from the properties of the ephemeral 2D R-Tree while the time 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 space that is linear in the number of updates in the degenerate spatiotemporal evolution.

2.2.2 Partially-Persistent R-Tree

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 indices; we use a 2D R-Tree for simplicity.

Kumar et al. presents a partially persistent R-Tree (called the Bitemporal R-Tree) [KTF98] following an approach similar to [BGO+96] (which shows how to make a B-tree partially persistent). In temporal applications it is assumed that updates arrive in order. Hence we assume that all updates in the evolution are provided in a sequence ordered by time (the "update sequence"). For simplicity of exposition, assume at most one update per time instant (in practice many updates happen per time instant).

The partially-persistent R-Tree (PPR-Tree) records the evolution of an ephemeral R-Tree on which the above update sequence is applied. However, it does not store snapshots of all the

versions of the ephemeral R-Tree. Instead it records the evolution updates efficiently so that the storage space remains linear, while still providing fast query time. The PPR-Tree is actually a directed acyclic graph of nodes (each node is again corresponding 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's evolution. Data records in the PPR-Tree leaf nodes maintain the evolution of the ephemeral R-Tree data objects. Each data record is thus extended to include the two lifetime fields: *insertion-time* and *deletion-time.* Similarly, index records in the directory nodes of the PPR-Tree maintain the evolution of the corresponding index records of the ephemeral R-Tree and are also augmented with insertion-time and deletion-time fields.

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

An update leads to a *structural* change if at least one new node is created. *Non-structural* are those updates which are handled within an existing node. An insertion update triggers a structural change if the target leaf node already has B records. A deletion update triggers a structural change if the resulting node ends up having fewer 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* [BGO+96]. 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 proposal [LS89] and the page copying concept proposed in [TK95]. The split on a node x at t, is performed by copying to a new node y the records alive in node x at t. Node x is considered *dead* after time instant t. (We can assume that

the deletion-time field of all x's alive records is changed to t even though this is not needed in practice). Then the resulting new node has to be incorporated in the structure (for details we refer to [KTF98, VV97, BGO+96]).

Answering a range query about region S and time t has two parts. First, the root alive at t is found. This part is conceptually equivalent to accessing the root of ephemeral R-Tree which indexes the data objects alive at t. Second, the objects intersecting S are found by searching this tree in a top-down fashion as in a regular R-Tree. The lifetime interval of every record traversed should contain the time instant t, while the record's MBR should intersect the region S. Answering a query that specifies a time interval $[t, t')$ is similar. First all roots with lifetime interval intersecting the time 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 [RKV95] and later refined in [CF98]. The query consists of a point or object and a time interval. The answer contains the q nearest objects that are closest to the query object during the specified time interval. The algorithm proposed in [RKV95] can be used directly; the only difference is on the way distances are computed. All objects that are not alive during the query time interval have infinite distance to the query object. On the other hand for the objects that have lifetimes intersecting the query time interval, the distance is computed using their extent dimensions. The algorithm visits first 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.

2.3 General 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 that we introduce artificial deletions and re-insertions of objects. Here we assume that objects move with a linear function over time and later we discuss the case where objects move and change extent in a more complex way. We proceed with some definitions.

Definition 1 *Consider a 2-dimensional spatial object o that moves and/or changes its extent during its lifetime interval L. This evolution creates a spatiotemporal object O^L which is the 3-dimensional volume occupied by o during its lifetime L.*

In the rest, 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. We define the function* **Empty:** $G \rightarrow R$ *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 2.3 shows the movement of object o_1 from time t_1 to t_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.

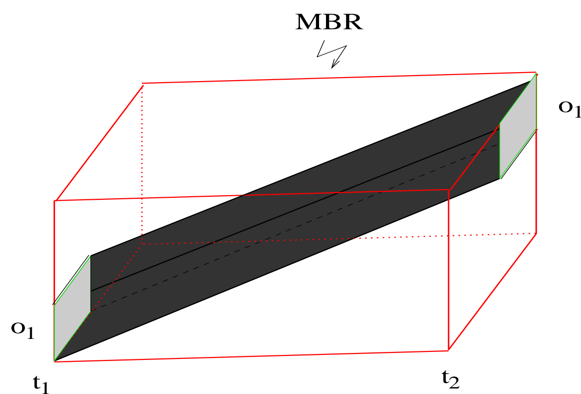


Figure 2.3: *A spatiotemporal object.*

Next we define the (artificial) split operation. Consider the spatiotemporal object created by the evolution of object o from t_i to t_j. A split operation at the time t_s, where $t_i < t_s < t_j$,

artificially deletes object o at t_s and reinserts it at the same time instant with the same extent at the same position. As a result, the original spatiotemporal object $O^{[t_i,t_j)}$ is replaced by two new spatiotemporal objects, namely $O^{[t_i,t_s)}$ and $O^{[t_s,t_j)}$. 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 details.

Figure 2.4 shows the result of a split operation performed on the object evolution of Figure 2.3. The view from the $x - axis$ is depicted (i.e., the spatial object is simply an interval that moved along the $y - axis$ from time t_1 to t_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 on object with interval y_1 with lifetime $[t_1, t_s)$ and object t_2 with lifetime $[t_s, t_2)$. Without the artificial split, we had an object y_{tot} with lifetime $[t_1, t_2)$. Using the split operation we can decrease the empty space, and consequently the possibility of overlapping among the nodes in the ephemeral R-Tree. Thus the query performance of the index is improved with the expense of course of using more space, since every time we perform a split, we increase the number of the indexed objects by one.

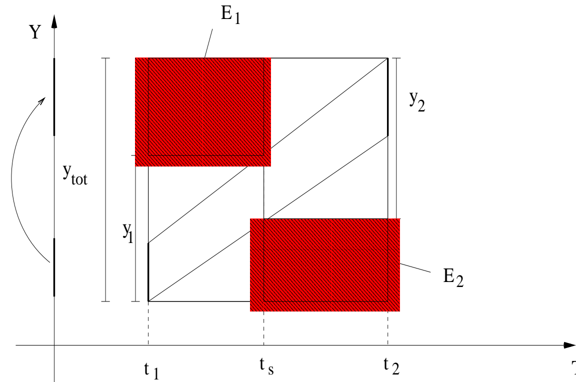


Figure 2.4: A split operation of a 1-dimensional object (interval) that moved continuously from t_1 to t_2.

The more general split operation allows a spatiotemporal object to be split many times.

Definition 3 *Consider again a spatiotemporal object $O^{[t_i,t_j)}$. Then the* **Split-k(O)** *operation partitions $O^{[t_i,t_j)}$ into $k + 1$ spatiotemporal objects using sp_l splitting points, where $t_i \leq sp_l \leq t_j, l = 1, ..., k$.*

The splitting points can be any time instants between the start time and the end time of the lifespan of a spatiotemporal object. In the case that objects 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. Note that this is true only for objects that move linearly (while retaining the same extent). It is also true for objects that change one of their extent dimensions linearly. However, it is not the optimal choice for objects that change both their extent dimensions. There are ways to compute the best splitting points even in this scenario and we discuss this case in the next section.

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 the 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 space 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:** $G \times \mathcal{N} \rightarrow \mathcal{R}$*, 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 the objects that generated after applying the operation split-k(O).

For example, in the 1-dimensional case that is shown in Figure 2.4, we have $gain(O,1) = E_1 + E_2$. Next, 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 1 *Let O be a spatiotemporal object created by a linear movement. Then for any i, j, with $i < j$:*

$$gain(O,i) - gain(O,i-1) \geq gain(O,j) - gain(O,j-1)$$

Proof: The position change is described by an equation of the form: $f(\bar{t}) = \alpha\bar{t} + \beta$. We initially 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 over 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 2.5 depicts an 1-dimensional object O that is split one and two times. 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.)

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

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

For example, 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

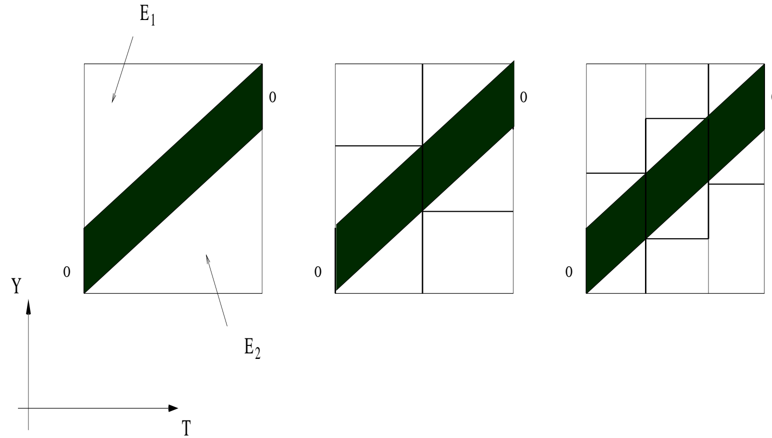


Figure 2.5: A 1-dimensional moving object with one and two splits.

extent. After k splits, we get $k+1$ spatiotemporal objects, that 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:

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

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 refers to volume.

If the initial and final positions have different x and y coordinates, (see Figure 2.6), the gain formula involves also the spatial extent of the object. Using the same arguments as for point objects it can be shown that:

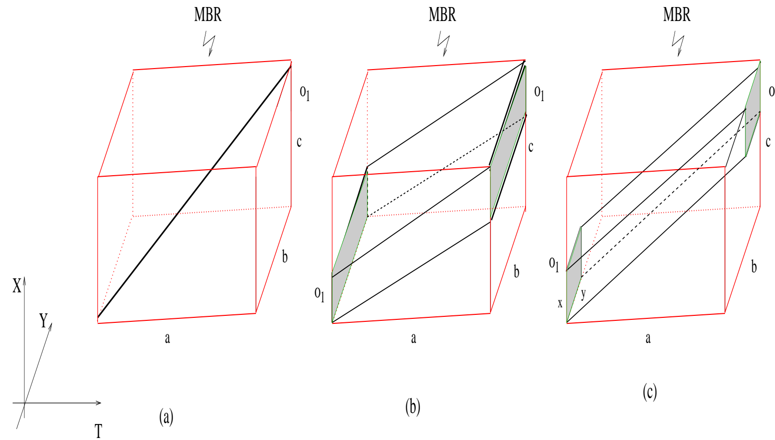


Figure 2.6: Three cases for 2-dimensional moving objects, (a) point, (b) moving rectangle with the same starting and ending x-coordinates and (c) moving rectangle with different starting and ending x and y coordinates.

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

where a is the lifespan of the object, x and y the lengths of the two sides of the object's MBR and b and c the displacement of the object in Y and X axes respectively.

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.,

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

■

It should be noted that the above property does not hold for spatiotemporal objects created by non-linear movement functions. For example, Figure 2.7, shows the case of a 1-dimensional moving object, where the difference in gain between the first split and no split is smaller that the difference between the second and the first split. Note that in that case we don't use equidistant splits, since the motion is not linear. Similar examples exist for 2-dimensional objects.

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

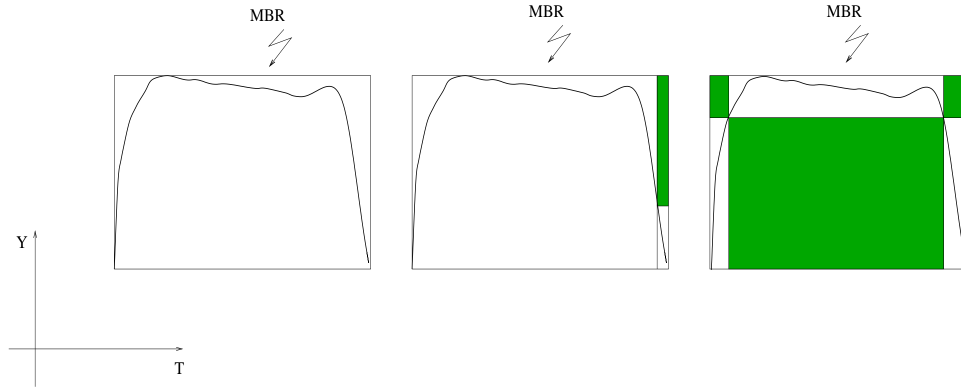


Figure 2.7: The spatiotemporal object created by a 1-dimensional moving point and the gain after one and two splits.

empty space, but after some point the gain in empty space will be small.

As presented 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 will improve.

On the other hand, the 3D R-Tree is not expected to be significantly affected by the splits. To justify this, we use the results presented in [TS96]. 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, since a split operation decreases the density of the dataset (D), but at the same time increases the number of indexed objects (m).

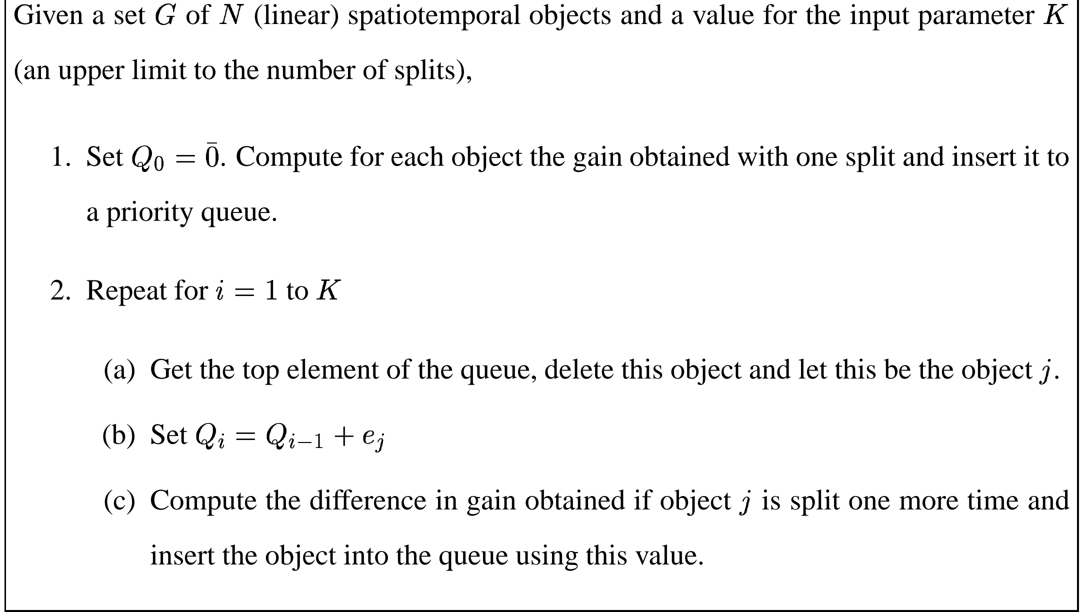
2.3.1 A Greedy Algorithm

In this subsection we introduce a greedy algorithm for the MES problem which optimally finds the spatiotemporal objects to split for linearly moving objects. We also discuss possible implementation methods of the algorithm.

Figure 2.8 describes 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 $\bar{0} = (0, ..., 0)$. Then we find the optimal solutions 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 one object one more time. The vector e_j has zero values to all position except the position j whose value is one (1). Thus we choose from all possible objects, the one that gives the higher gain in empty space.

A naive implementation of this idea will give an algorithm with complexity $O(KN)$ operations in main memory. Note however, 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 $gain(o_j, l + 1) - gain(o_j, l)$.

Next we state and prove the following theorem:



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 the 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 2.8: The Optimal GREEDY algorithm

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 the 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.

Clearly, fro $k = 1$ the best solution is to split the object that gives the highest reduction in empty space. 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 one k_2 and so on. Also we have that $\sum_{i=1}^{N} k_i = k$ and $k > 1$. 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 $i \in \{1, ..., N\}$.

Let assume that this is not true and that the optimal solution for $k+1$ splits has the form:

$Q_{k+1'} = \{k_1, ..., k_i + 2, ..., k_j - 1..., k_N\}$ 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 1 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 $Q_{k+1'}$.

Thus, the optimal solution for $k + 1$ splits can be derived by the optimal solution with k splits and the algorithm in Figure 2.8 does exactly this.

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 [CLR90]. 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 [Arg95]. The basic idea is to perform operations (insertions and deletions) off-line in such a way that the

amortized complexity of each operation is $O(\frac{1}{B} log_{\frac{M}{B}} \frac{N}{B})$ [Arg97]. 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 attributes linearly. As mentioned earlier it works as an approximation otherwise.

2.3.2 A Dynamic Programming Algorithm for the General Case

For objects that move and/or change with a complex non-linear function over time, the above greedy algorithm is suboptimal. To find the optimal solution here, we can use a dynamic programming algorithm for objects that change with any complex function.

Let assume that we have N spatiotemporal objects and K possible splits. Also, we assume that given an object and a number of splits, we can find the optimal way to split these objects with the given splits. Let $gain(i, j)$ denote the gain in empty space that we get by splitting object i using at most j splits and SP(i,j) the total gain in empty space by splitting the set of the first i objects using at most j splits. Then we have that:

$$SP(i, j) = \min_{0<=t<=j} \{SP(i-1, j-t) + gain(i, t)\}$$

and we need to find the SP(N,K) in order to solve our problem. The running time for an implementation of this algorithm in main memory is $O(NK^2)$ and assuming $K = O(N)$, then it becomes $O(N^3)$. Given that the number of the spatiotemporal objects is large (in order of millions or billions of objects) the above algorithm is not very practical. Note however that we can use the same algorithm to find the best splitting points for a single object given a number of splits and this can be used with the previous greedy algorithm.

2.4 Performance Evaluation

We first describe the datasets and outline the workloads used in our experimental evaluation. Then, subsection 2.4.2 discusses the performance characteristics of various implementations

(tuning) of the PPR-Trees that we have developed. Finally, we present experimental results for both types of object evolution namely, the degenerate (subsection 2.4.3) and the general (subsection 2.4.4) cases.

2.4.1 Experimental Setup

For all methods 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 deletion operations a buffer of 10 pages was used with a LRU replacement policy. In all methods, during the query phase the buffer was invalidated before a new query is executed (so that strengths and weaknesses of the various implementations are revealed). For the 3D R-Tree method, we used an implementation of the R*-tree [BKS+90]. We implemented the Skeleton SR-Tree based on the description in [KS91]. In our implementation we allowed index nodes to have larger pages (starting from the leaf nodes the page size doubles as the level reaches the root). At a given index page, one third is allocated to storing spanning segments while the rest is for 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 overflow pages (i.e., the reported SR-Tree query times are underestimates of the actual ones).

We generated various spatiotemporal datasets to compare the performance of the different methods. The datasets for the degenerate case were similar to the spatiotemporal datasets described in [VTS98]. Objects in a given time instant were approximated by their 2-dimensional MBRs and the size of the data space was 1.0×1.0 (unit square). Moreover, 70% of the objects were small rectangles with small lifetimes. The length of each rectangle in the x and y axes was uniformly chosen from the interval (0, 0.04] and the centers of the rectangles were uniformly distributed in the unit square. The lifetime of each object followed a Poisson distribution with mean value equal to 50. Another 15% of the objects were large rectangles with small lifetimes. Here the length of each rectangle in spatial dimensions was uniformly chosen from (0, 0.6] and the lifetimes were the same as above. The remaining 15% objects were small rectangles with large lifetimes. The lifetimes for these objects were uniformly chosen between 250 and 500 time instants. For each object a

number of lifetimes between 0 and 10000 was generated, and the time intervals between subsequent lifetimes had the same characteristics as the lifetimes. We generated five different datasets with objects per time instant ranging from 250 to 2500. We call this type of datasets DG (degenerate).

For the general case we created two different types of datasets. First we had 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 position. Each set had one-third of "slowly" moving rectangles whose sides were uniformly chosen in $(0, 0.02]$, and speeds between 0 and 0.001. Another third had 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 change their center positions. The lifetime of each object had mean value 50. Again the average number of objects per time instant ranged between 500 and 2500.

We also generated a collection of datasets that was 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. Another third are moving objects and the rest are objects that change position and extent over time. To generate some of the above datasets we used the GSTD generator [TSN99]. In Table 2.1, we give the main characteristics of the datasets.

Table 2.1: The datasets used for testing the index structures.

Dataset	Avg Number of Objects per Time Instant	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

It should be noted that to insert objects into a PPR-Tree, we first sort the dataset over the object insertion and deletion times. Then the dataset is processed sequentially until the end of the evolution. For the 3D R-Tree the dataset is first sorted on the object insertion times and objects are inserted in that order. For the Skeleton SR-Tree inserting the spatiotemporal objects according to insertion time 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 various query workloads were generated (separate workloads were created for range and nearest neighbor queries). A query workload consists of 1000 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 time instant, and, "period" queries where each query specified a time 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 inside the data space. The temporal part was a "period" selected randomly, with length between 0 and 100.

2.4.2 Tuning the PPR-Tree

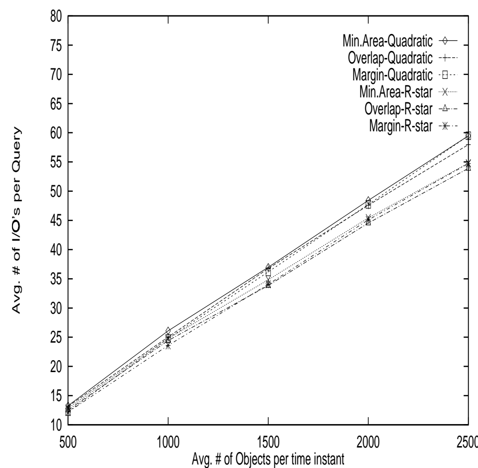


Figure 2.9: Query performance for different merging/splitting policies.

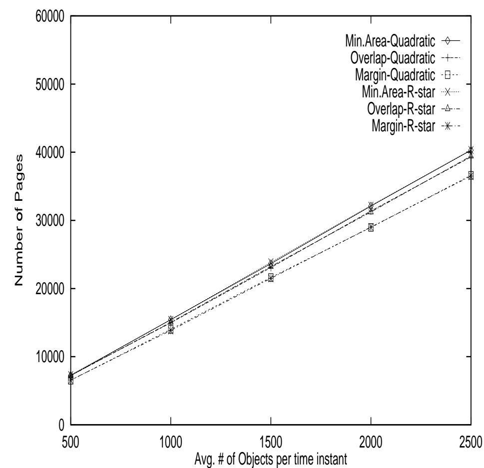


Figure 2.10: Space consumption for different merging/splitting policies.

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. Note that the merging policies of

the partially persistent B-tree are not applicable. The reason is that in a B-tree there is a single order among indexed objects; moreover, data is kept in pages according to this order. Consequently, for each underutilized page there are at most two possible sibling pages that it can merge with. On the other hand, a PPR-Tree stores multidimensional objects and for each page there are many possible sibling pages (all pages that have the same parent with the underutilized page are candidates). 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, which is the sum of the lengths of all sides of the bounding rectangle.

For the splitting policies, we use two methods. On is called *Quadratic* and it has been proposed in the original R-Tree paper. The other one (*R-star*) is the policy that is used by the R*-tree. The first policy assigns objects in two groups, initializing these groups by picking the pair of objects that would waste the most area if put 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.

In Figure 2.9 we plot 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 space consumption of the PPR-Tree is depicted on Figure 2.10. Here the important factor is the merging policy and the Margin policy gives the best results. As a result, for the rest of our experiments we implemented the PPR-Tree using the R-star splitting policy and the Margin policy for merging nodes.

2.4.3 Degenerate Case

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 2.11-2.13 depict the results for snapshot queries with Small, Medium and Large size (in spatial extent) respectively. The average lifetime of the objects is 50 time instants. In all cases the partially persistence methodology outperforms the 3D R-Tree. The difference is higher for smaller queries. Figure 2.14 shows the results for Small/Period queries. Here the query time period ranged from 0 to 100. The difference between the two methods is decreasing as the size of the query time period increases. Since the PPR-Tree (and partial persistence) is optimized towards snapshot queries, a query involving a large period (many subsequent snapshots) will overlap with many object copies thus decreasing query performance.

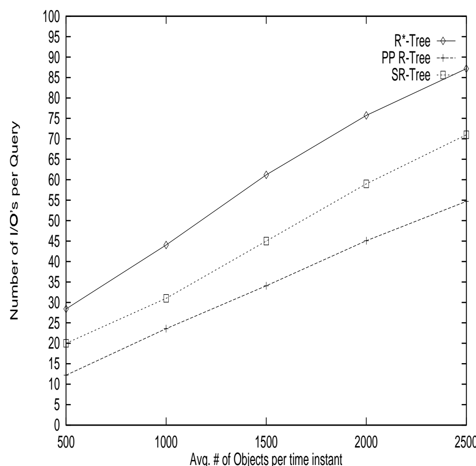


Figure 2.11: Query performance for small/ snapshot queries and DG datasets.

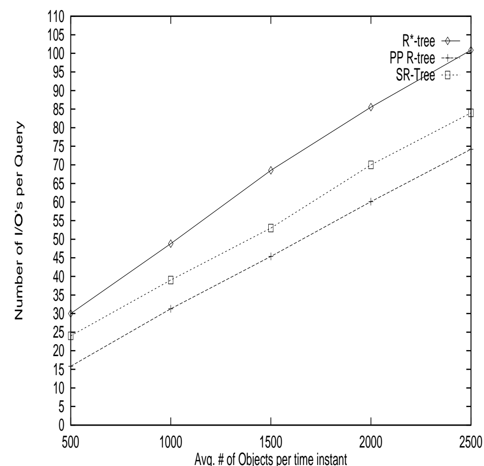


Figure 2.12: Query performance for medium/ snapshot queries and DG datasets.

Figure 2.15 depicts the space consumption of both methods, for DG datasets. As expected the space consumption of the PPR-Tree is higher than the 3D R-Tree. Note though that the space overhead remains linear to the number of objects and it is about 2.5 times more than the space used

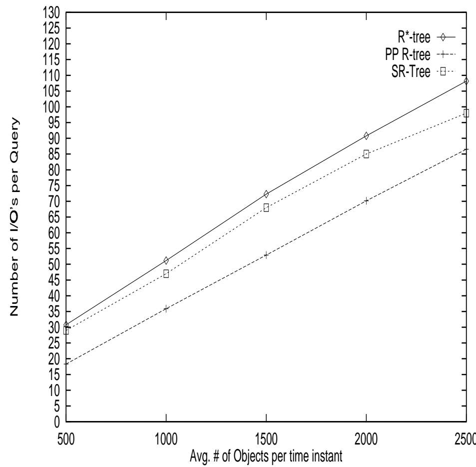


Figure 2.13: Query performance for large/ snapshot queries and DG dataset.

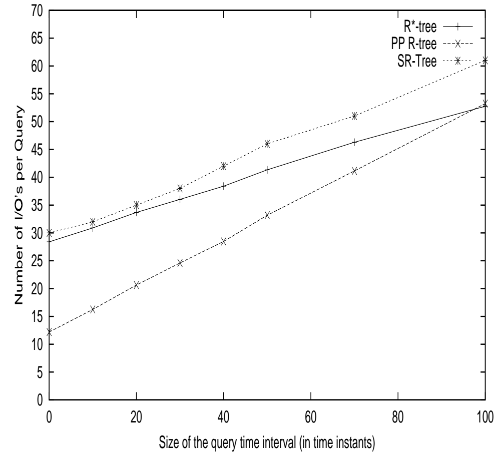


Figure 2.14: Query performance for time period queries and DG dataset.

by the 3D R-Tree.

2.4.4 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 gains 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 these two approaches against the simple approach were no artificial split is considered. That is, we used a 3-dimensional MBR around a spatiotemporal object and indexed them using a plain 3D R-tree. Similarly we used the $maxMBR$ approach for the PPR-Tree (maxMBR-PPR-Tree). Unless otherwise stated, the number of artificial splits were about half of the number of original spatiotemporal objects ($1.5N$).

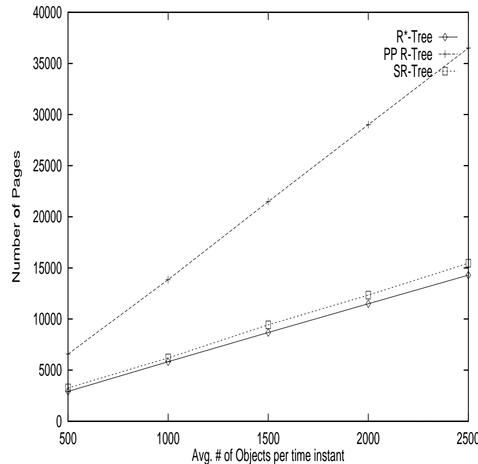


Figure 2.15: Space consumption for DG datasets.

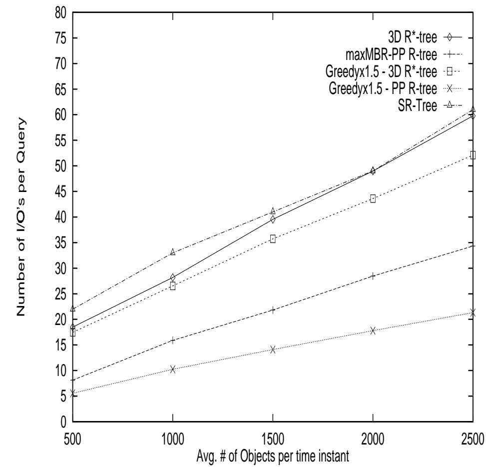


Figure 2.16: Query performance for small/ snapshot queries and MV datasets.

We used this number of splits because it provides good query performance with a small increase in space overhead.

Figures 2.16-2.18 depict the results for snapshot queries and MV datasets. The greedy algorithm combined with the PPR-Tree provides the best query performance. Next is the simple PPR-Tree with 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 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 space for a method that uses the greedy approach is about 1.5 times the space of the same non-greedy method (Figure 2.19). Time period queries appear in Figure 2.20 using a dataset with 1000 objects per time instant. 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 3D R-tree. This is because the splits introduced by the greedy approach introduce copies that the R-tree considers as separate objects.

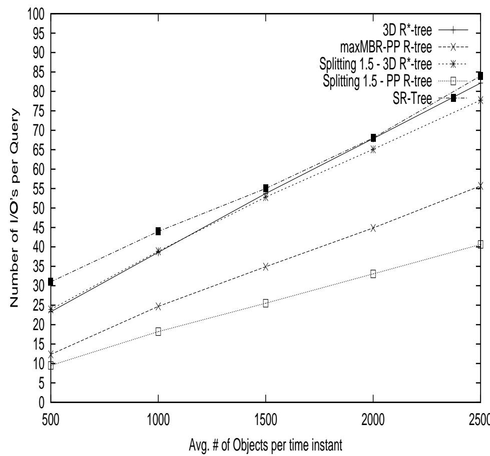


Figure 2.17: Query performance for medium/ snapshot queries and MV datasets.

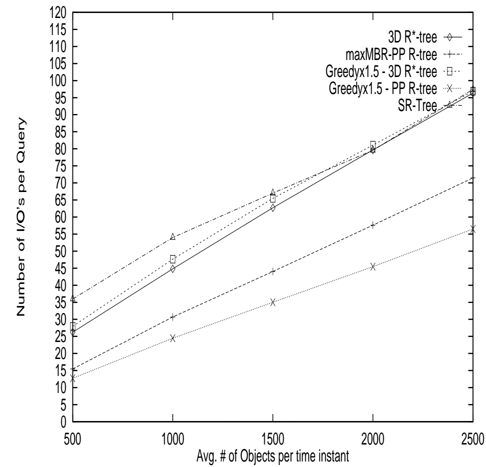


Figure 2.18: Query performance for large/ snapshot queries and MV datasets.

The effect of using different number of splits is examined in Figure 2.21. The query performance is shown for three MV snapshot query workloads with 1000 objects per time instant and different number of splits. For brevity only the Greedy-PPR-Tree is presented. The dataset was a MV dataset with 1000 objects per time instant. Clearly increasing the number of splits improves the query performance. The space is also increasing proportionally to the split percentage increase (and is thus not depicted).

The performance comparisons for the general datasets (that include mixtures of moving/static/extending objects) appear in Figures 2.22 to 2.27. 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. We report results for the general datasets (GN), but the same trend was observed for the other datasets as well. In Figure 2.28 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 time period was 20. Figure 2.29 reports

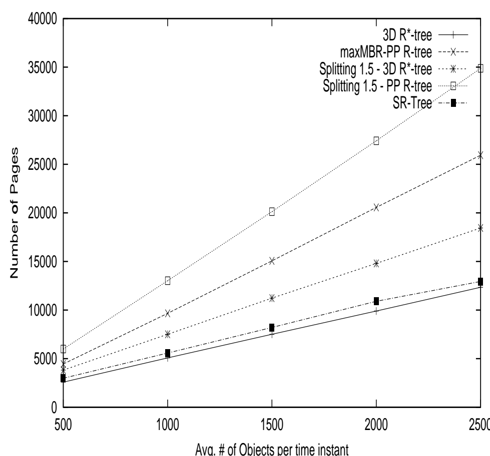


Figure 2.19: Space consumption for MV datasets.

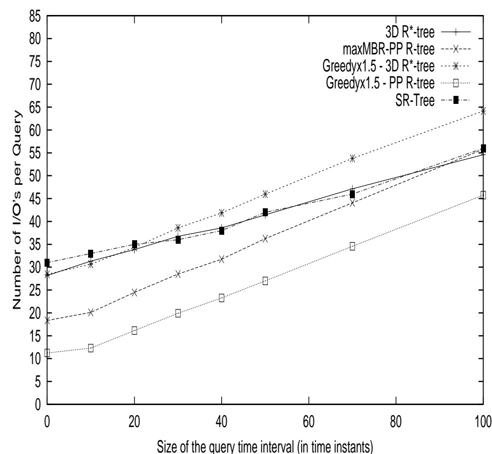


Figure 2.20: Query performance for time period queries and MV datasets.

results for nearest neighbor queries with different time periods. The Greedy-PPR-tree has again the best query performance.

Finally, in Figures 2.30 and 2.31 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 we can decrease the construction time considerably. The 3D R-Trees have lower construction time that 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 time and again at the deletion time. 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 the querying, i.e., the update cost is not that critical.

2.5 Related Work in Spatiotemporal Indexing

Research in the area of spatiotemporal database indexing is limited. In particular, Theodoridis et al. [TSP+98] summarize the issues that a spatiotemporal index needs to address. In an early

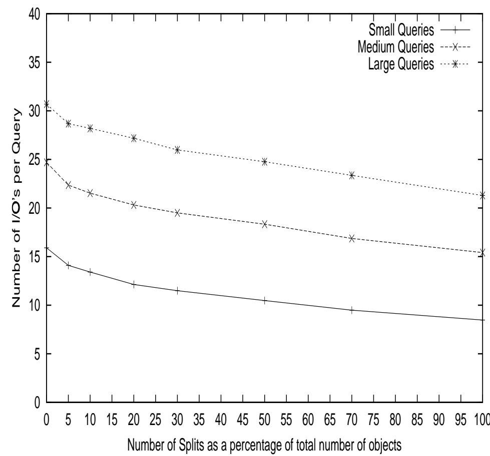


Figure 2.21: Query performance of the Greedy-PPR-tree for snapshot queries and different number of splits and MV datasets.

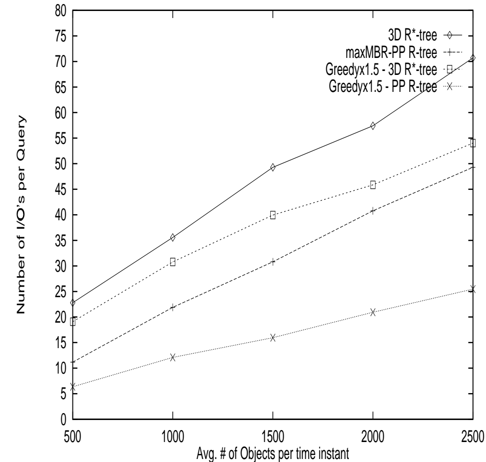


Figure 2.22: Query performance for small/ snapshot queries and GN datasets.

paper [XHL90], 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) at every time instant that this 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 [NS98, TVM98, XHL90, NST99] the idea of overlapping trees is used to make an index partially persistent. Different indices are created for each time instant, but to save space, common paths are maintained only once since they are shared among the structure. However the overlapping method has a logarithmic space 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 [NST99] the overlapping R-Tree (HR-Tree) has an order of magnitude higher space overhead that 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, Saltenis and Jensen [SJ99], an R-Tree is ex-

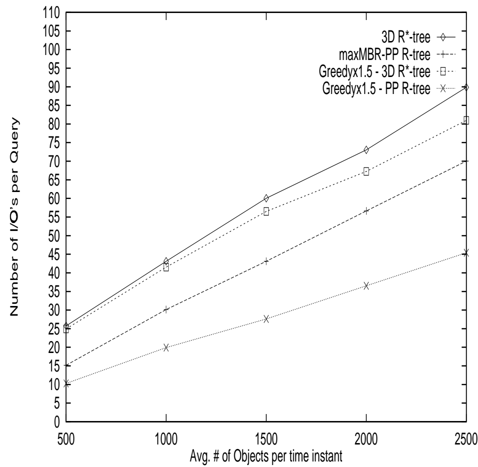


Figure 2.23: Query performance for medium/ snapshot queries and GN datasets.

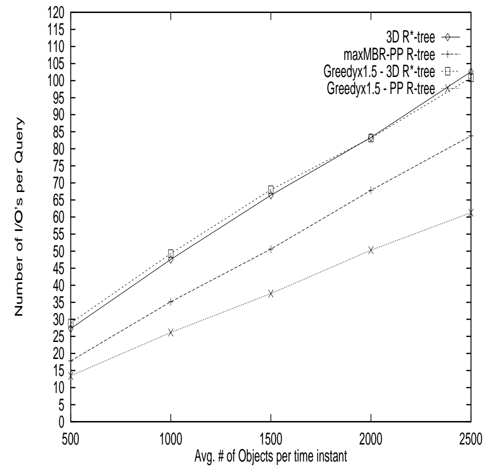


Figure 2.24: Query performance for large/ snapshot queries and GN datasets.

tended to support transaction and valid time. However, this work concentrates on the combination of degenerate evolutions and bitemporal datasets.

2.6 Summary

We have examined the problem of indexing objects in spatiotemporal evolutions, by using the idea of partially persistent data structures. However, the partial persistence approach considers only objects that remain unchanged during their evolution. This is not realistic in many real life applications where objects can change their extent/position over time. 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. The presented approach provides very fast query time at the expense of some extra space, which however is linear to the number of changes in the evolution. We have shown the merit of our method by comparing it with an approach that sees time as another dimension and uses (i) a regular 3D R-Tree,

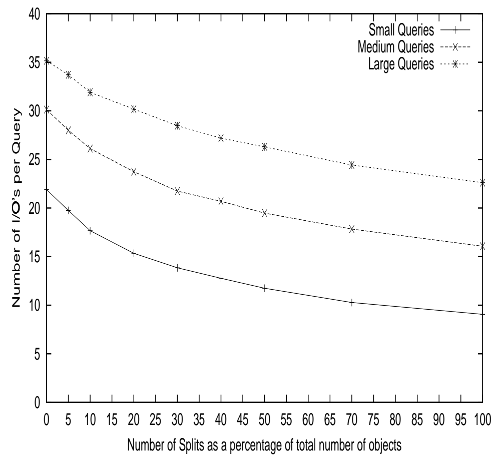


Figure 2.25: Query performance of the Greedy-PPR-Tree for snapshot queries and different number of splits and GN datasets.

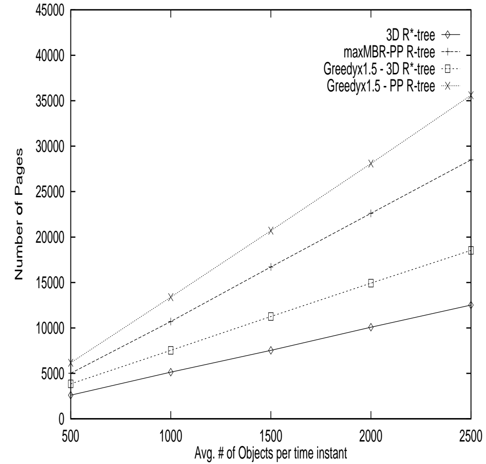


Figure 2.26: Space consumption for the GN datasets.

or, (ii) a Skeleton Segment R-Tree.

While in this chapter we considered range and nearest neighbor queries that refer to the past, in the next chapter we investigate the problem of answering queries that refer to the future assuming that objects move linearly over time.

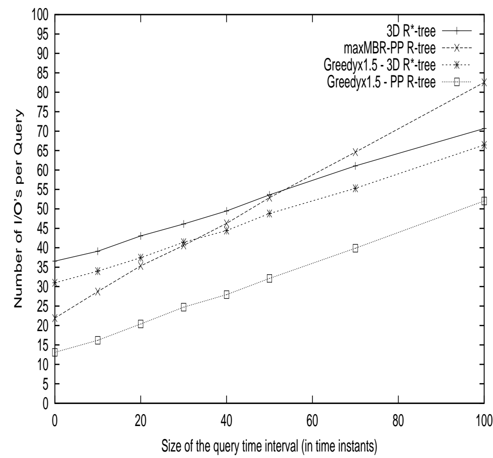


Figure 2.27: Query performance for time period queries and GN dataset.

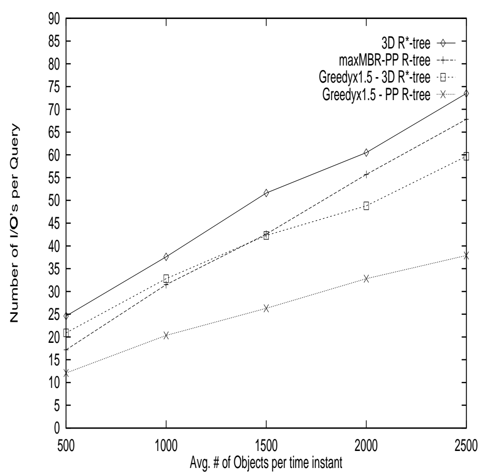


Figure 2.28: Nearest Neighbor query performance for GN datasets.

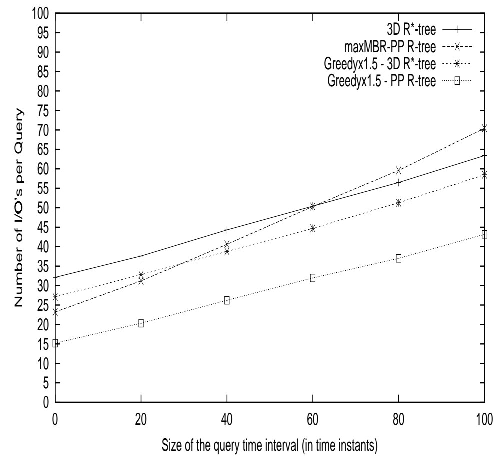


Figure 2.29: Nearest Neighbor query performance for different time periods and GN datasets.

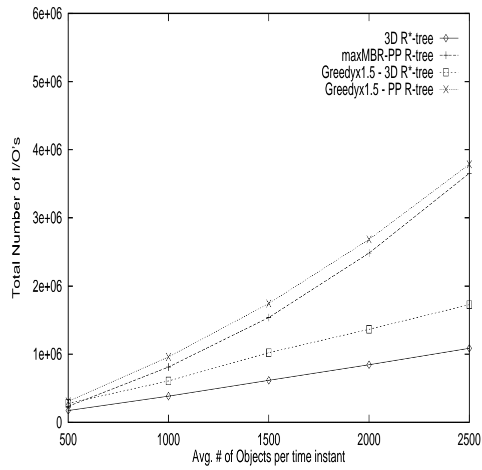


Figure 2.30: Construction cost for MV dataset.

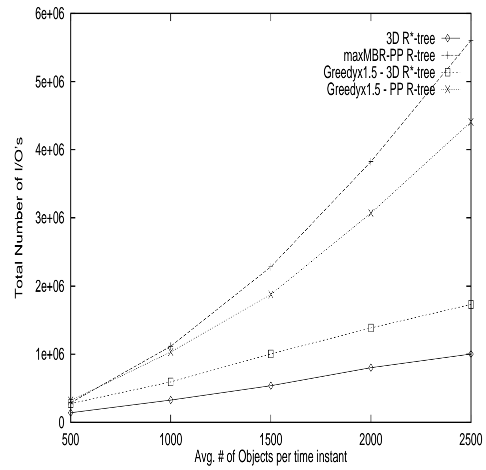


Figure 2.31: Construction cost for GN dataset.

Chapter 3

Indexing the Future Positions of Continuously Moving Objects

3.1 Introduction

Traditional database management systems assume that data stored in the database remain constant until explicitly changed through an update. While this model serves well many applications where data changes in discrete steps, it is not appropriate for applications with continuously changing data. One such application is a "motion" database that stores the location of mobile objects (e.g. cars). Since objects change location continuously, one would have to update the database at every unit of time. This is clearly an inefficient and infeasible solution considering the prohibitively large update overhead.

A better approach is to abstract each object's location as a function of time $f(t)$, and update the database only when the parameters of f change (for example when the speed or the direction of a car changes). Using $f(t)$ the database can compute the location of the mobile object at any time in the future. While this approach minimizes the update overhead, it introduces a variety of novel problems (such as the need for appropriate data models, query languages and query processing and optimization techniques) since the database is not directly storing data values but

functions to compute these values. Motion database problems have recently attracted the interest of the research community: ([SWC+97, WCD+98, WXC+98]) present the Moving Objects Spatio-Temporal (MOST) model and a language (FTL) for querying the current and future locations of mobile objects; ([EGS+98]) proposes a model that tracks and queries the history (past routes) of mobile objects, based on new spatio-temporal data types. Another spatiotemporal model appears in [CR99]. Spatio-temporal queries about mobile objects have important applications in traffic monitoring, intelligent navigation and mobile communications domains. For example in databases that track cars in a highway system, we can detect future congestion areas. In mobile communications, we can allocate more bandwidth for areas where high concentration of mobile phones is approaching. There is already a GIS system [ARC98] that supports tracking and querying of mobile objects.

In this chapter we focus on the problem of indexing mobile objects. In particular we examine how to efficiently address range queries over the object locations into the future. An example of such a spatio-temporal query is: "Report all the objects that will be inside a query region P 10 minutes from now". Note that the answer to this query is tentative in the sense that it is computed based on the current knowledge stored in the database about the mobile objects' location functions. In the near future this knowledge may change, which implies that the same query could have a different answer. As the number of mobile objects in the applications we consider (traffic monitoring, mobile communications, etc.) can be rather large, we are interested in external memory solutions.

While in general an object could move anywhere in the 3-dimensional space using some rather complex motion, we limit our treatment to objects moving in 1- and 2-dimensional spaces and whose location is described by a linear function of time. There is a strong motivation for such an approach based on the real-world applications we have in mind: straight lines are usually the faster way to get from one point to another; cars move in networks of highways which can be approximated by connected straight line segments on a plane; this is also true for routes taken by airplanes or ships. In addition, solving these simpler 1- and 2-dimensional problems may provide intuition for addressing the more difficult problem of indexing general multidimensional functions.

3.2 Background

We consider a database that keeps track of mobile objects moving in one and two dimensions. We model the objects as points that move with a constant velocity starting from a specific location at a specific time instant. Using this information we can compute the location of an object at any time in the future for as long as its movement characteristics remain the same. In one dimension, an object started from location y_0 at time t_0 with a velocity v (v can be positive or negative) will be in location $y_0 + v(t - t_0)$ at time $t > t_0$. Similarly for objects moving in two dimensions. Objects are responsible for updating their motion information, each time their speed or direction changes. Also, we assume that the objects can move inside a finite terrain (a line segment in one dimension or a rectangle in two). Thus when an object has reached the limits, it has to issue an update (either because it is deleted or it is reflected). Finally, we allow insertion of a new object or deletion of an old one, eg. the system is dynamic.

We would like to answer efficiently proximity queries among the mobile objects. In particular, we are interested in answering queries of the form: "Report the objects that reside inside the interval $[y_{1q}, y_{2q}]$ (or the rectangle $[x_{1q}, x_{2q}] \times [y_{1q}, y_{2q}]$ in two dimensions) at the time instants between time t_{1q} and t_{2q}, (where $t_{now} \leq t_{1q} \leq t_{2q}$), given the current motion information of all objects". We call this type of query a *one dimensional MOR (Moving Objects Range) query* for objects moving in one dimension and a *two dimensional MOR query* for objects moving in two dimensions.

3.3 Indexing in one dimension

We begin with the simpler problem of objects moving on a 1-dimensional line. We partition the mobile objects into two categories, the objects with low speed $v \approx 0$ and the objects with speed between a minimum v_{min} and maximum speed v_{max}. We consider here the "moving" objects, eg. the objects with speed greater than v_{min}. We discuss the case of slowly moving objects later.

We assume that the objects move on the $y-$axis between 0 and y_{max} and that an object can update its motion information whenever it changes. We treat an update as a deletion of the old

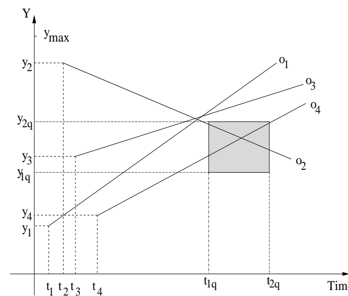


Figure 3.1: Trajectories and query in (t, y) plane.

information and an insertion of the new one. Next we give different geometric representations of the problem and for each one we discuss access structures to efficiently address MOR queries.

3.3.1 Space-time representation

In this representation we plot the trajectories of the mobile objects as lines in the time-location (t, y) plane. The equation of each line is $y(t) = vt + a$ where v is the slope (the velocity in our case) and a is the intercept, that can be computed by the motion information. In fact a trajectory is not a line but a semi-line starting from the point (y_i, t_i). However since we ask queries for the present or for the future, assuming that the trajectory is a line does not affect the correctness of the answer. Figure 3.1 shows a number of trajectories in the plane.

The query is expressed as a 2-dimensional interval $[(y_{1q}, y_{2q}), (t_{1q}, t_{2q})]$. The answer is the set of objects that correspond to lines that intersect the query rectangle.

While the space-time representation is quite intuitive, it leads to indexing long lines, a situation that causes significant shortcomings to traditional indexing techniques.

One way is to index the lines using a Spatial Access Method (SAM). Then each line is approximated by a minimum bounding rectangle (MBR) which is then indexed using an R-tree[Gut84] or an R*-tree[BKS+90]. However, this approach is problematic because: (i) an MBR assigns to the moving object a much larger area than a line has, (ii) since objects retain their trajectory until being updated, all lines in figure 3.1 extend to ”infinity”, i.e. a common ending on the time dimension. Mapping a line segment as a point in four dimensions, by taking the coordinates of the end points,

will also not work. Even if we partition the time dimension in time intervals ("sessions") of length ΔT (as in [TUW98]) and index the part of each trajectory that falls in the current session, we still have segments with a common endpoint (the end time of the current session). Another shortcoming is that the SAM can only address queries until the end of the current session.

A different approach is to decompose the data space into disjoint cells and store with each cell the set of lines that intersect it. Indexes that follow this approach are the R+–tree[SRF87], the cell–tree[Gun89] and the PMR–quadtree[Sam90]. The main drawback for these methods is that every line will have many copies; this becomes worse in our environment since lines are large. Storing many copies affects both the update performance (when an object changes its trajectory, its previous route has to be deleted from all cells it was contained), as well as space. [1]

In [Jag90] a method is proposed to index line segments based on the dual transformation. The use of dual transformation to index mobile objects is also proposed in [WXC+98]. In the next section we consider this approach in our setting, namely using the dual transformation to index mobile objects.

3.3.2 The dual space-time representation.

Duality is a powerful and useful transform frequently used in the computational geometry literature; in general it maps a hyper-plane h from R^d to a point in R^d and vice-versa. The duality transform is useful because it allows formulation of a problem in a more intuitive manner.

In our case we can map a line from the *primal* plane (t, y) to a point in the *dual* plane. There is no unique duality transform, but a class of transforms with similar properties. Sometimes one transform is more convenient than another.

Consider a dual plane where one axis represents the slope of an object's trajectory and the other axis its intercept. Thus the line with equation $y(t) = vt + a$ is represented by the point (v, a) in the dual space (this is called the Hough-X transform in [Jag90]). While the values of v are between $-v_{max}$ and v_{max}, the values of the intercept are dependent on the current time. If the

[1] [TUW98] uses a method based on the PMR-quadtree; their experiments show that even for a small number of mobile objects (50K) the number of copies can become quite large (about 250 copies/object).

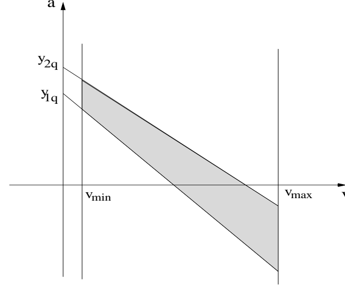


Figure 3.2: Query in the dual Hough-X plane.

current time is t_{now} then the range for a is $[-v_{max} \times t_{now}, y_{max} + v_{max} \times t_{now}]$.

The query is transformed in a polygon in the dual space. We can express this polygon using a linear constraint query [GRS+97].

Proposition 1 *The one dimensional MOR query is expressed in the dual Hough-X plane as follows:*

- *For $v > 0$ the query is : $Q = C_1 \wedge C_2 \wedge C_3 \wedge C_4$, where: $C_1 = v \geq v_{min}$, $C_2 = v \leq v_{max}$, $C_3 = a + t_{2q}v \geq y_{1q}$ and $C_4 = a + t_{1q}v \leq y_{2q}$.*
- *For $v < 0$ the query is : $Q = D_1 \wedge D_2 \wedge D_3 \wedge D_4$, where: $D_1 = v \leq -v_{min}$, $D_2 = v \geq -v_{max}$, $D_3 = a + t_{1q}v \geq y_{1q}$ and $D_4 = a + t_{2q}v \leq y_{2q}$.*

Proof: Let $v > 0$. The C_1 and C_2 constraints are straightforward by the problem setting. We will show that the other two are also necessary. Let a be a particular $a \leq y_{2q}$. Then a line with intercept a can intersect the query rectangle iff the slope has values:

$$\tan(\phi_1) \leq v \leq \tan(\phi_2) \Rightarrow \frac{y_{1q} - a}{t_{2q}} \leq v \leq \frac{y_{2q} - a}{t_{1q}}$$

From the above inequalities we get the two other constraints. Similarly for $v < 0$. ■

Since the query is different for positive and negative slopes, we can use two structures to store the dual points. It is easy to see that the range of the a's values is now $[-v_{max} \times t_{now}, y_{max} - v_{min} \times t_{now}]$.

However since time is monotonically increasing, the values of the intercept are not bounded. If the value of the maximum speed is significant, the values of the intercept can become very large and this potentially can be a problem (i.e., representing unbounded ranges of real numbers).

To solve this problem we use our assumption that when an object crosses a border it issues an update (i.e. it is deleted or reflected). Combining this assumption with the minimal speed, we can assure that all objects have updated their motion information at least once during the last T_{period} time instants, where $T_{period} = \frac{y_{max}}{v_{min}}$. We can then use two distinct index structures. The first index stores all objects that have issued their last update in the period $[0, T_{period}]$. The second index stores objects that have issued their last update in the interval $[T_{period}, 2T_{period}]$. Each object is stored only once, either in the first or in the second index. Before time T_{period}, all objects are stored in the first index. However, every object that issues an update after T_{period} is deleted from the first index and it is inserted in the second index. The intercept of the first index is computed by using the line $t = 0$ and for the second index using the line $t = T_{period}$. Thus we are sure that the intercept will always have values between 0 and $v_{max} \times T_{period}$. To query the database we use both indices. After time $2T_{period}$ we know that the first index is empty and all objects are stored in the second index, since every object have issued at least one update from T_{period} to $2T_{period}$. At that time we remove the empty index and we initiate a new one with time period $[2T_{period}, 3T_{period}]$. We continue in the same way and every T_{period} time instants we initiate a new index and we remove an empty one. Using this method the intercept is bounded while the performance of the index structures remain asymptotically the same as if we had only one structure.

Another way to represent a line $y = vt + a$, is to write the equation as $t = \frac{1}{v}y - \frac{a}{v}$. Then we can map this line to a point in the dual plane with coordinates $n = \frac{1}{v}$ and the $b = -\frac{a}{v}$ (Hough-Y in [Jag90]). Note that b is the point where the given line intersects the line $y = 0$. Note also that this transform cannot represent horizontal lines (similarly, the Hough-X transform cannot represent vertical lines). However, this is not a problem since our lines have a minimum and a maximum slope.

3.3.3 Lower Bounds

The dual space-time representation transforms the problem of indexing mobile objects on a line to the problem of *simplex* range searching in two dimensions.

In simplex range searching we are given a set S of 2-dimensional points, and we want to answer efficiently queries of the following form: given a set of linear constraints $a\dot{x} \leq b$, find all the points in S that satisfy all the constraints. Geometrically, the constraints form a polygon on the plane, and we want to find the points in the interior of the polygon. This problem has been extensively studied before in the static, main-memory setting (see for example the excellent survey in [AE98] and the related work section).

The only known lower bound for simplex range searching, if we want to report all the points that fall in the query region rather than their number, is due to Chazelle and Rosenberg ([CR92]). They show that simplex reporting in d-dimensions with a query time of $O(N^{\delta} + K)$, where N is the number of points, K is the number of the reported points and $0 < \delta \leq 1$, requires space $\Omega(N^{d(1-\delta)-\epsilon})$, for any fixed ϵ. This result is shown for the pointer machine model of computation [CR92]. The bound holds for the static case, even if the query region is the intersection of just two hyper-planes. Since ϵ can be arbitrary small, any algorithm that uses linear space for d-dimensional simplex range searching has worst case query time $\Omega(N^{(d-1)/d} + K)$.

Here we show that a similar bound holds for the input-output complexity of simplex searching. Following the approach in [SR95] we use the external memory pointer machine as our model of computation. This is a generalization of the pointer machine suitable for analyzing external memory algorithms. In this model, a data structure is modeled as a directed graph $G = (V, E)$, with a source w. Each node of the graph represents a disk block and is therefore allowed to have B data and pointer fields. The points are stored in the nodes of G. Given a query, the algorithm traverses G starting from w, examining the points at the nodes it visits. The algorithm can only visit nodes that are neighbors of already visited nodes (with the exception of the root) and, when it terminates the answer to the query must be contained in the set of visited nodes. The running time of the algorithm is the number of nodes it visits.

Theorem 2 *Simplex reporting in d-dimensions with a query time of $O(n^\delta + k)$ I/O's, where N is the number of points, $n = N/B$, K is the number of the reported points, $k = K/B$, and $0 < \delta \leq 1$, requires $\Omega(n^{d(1-\delta)-\epsilon})$ disc blocks, for any fixed ϵ.*

Proof: To prove the lower bound we show that, given δ, there exists a set of N points, and a set of $\Omega(n^{d(1-\delta)-\delta-\epsilon})$ queries such that each query has $\Theta(Bn^\delta)$ points, and the intersection of any pair of query results is small. To answer a query with $\Theta(Bn^\delta)$ points, the answering algorithm must visit $\Omega(n^\delta)$ nodes. To answer this query in $O(n^\delta)$ I/O's, at least a constant fraction of that many blocks have a constant fraction of their points in the answer of the query. But if the set of the queries has small intersection, it follows that to answer each query in the set in time $O(n^\delta)$ at least $\Theta(n^\delta) \times \Omega(n^{d(1-\delta)-\delta-\epsilon}) = \Omega(n^{d(1-\delta)-\epsilon})$ nodes have to be visited. It remains to show that such a set of queries exist. To do so we simply modify the existing construction by Chazelle and Rosenberg [CR92] by replacing each point in their point set by B copies. ■

A corollary of the theorem is that in the worst case a data structure that uses linear space to answer the 2-dimensional simplex range query and thus the one dimensional MOR query, requires $\Omega(\sqrt{n} + k)$ I/O's. Next we present a dynamic, external-memory algorithm that achieves almost optimal query time with linear space. As we shall see however this algorithm is not practical so we also consider faster algorithms to approximate the queries. Finally, we give a worst case logarithmic query time algorithm for a restricted but practical version of our problem.

3.3.4 An (Almost) Optimal Solution

Matousek ([Mat92]) gave an almost optimal algorithm for simplex range searching, given a static set of points. This main memory algorithm is based on the idea of simplicial partitions.

We briefly describe this approach here. For a set S of N points, a simplicial partition of S is a set $\{(S_1, \Delta_1), \ldots, (S_r, \Delta_r)\}$ where $\{S_1, \ldots, S_r\}$ is a partitioning of S, and Δ_i is a triangle that contains all the points in S_i. If $\max_i |S_i| < 2 \min_i |S_i|$, we say that the partition is balanced. Matousek ([Mat92]) shows that, given a set S of N points, and a parameter s (where $0 < s < N/2$), we can construct in linear time, a balanced simplicial partition for S of size $O(s)$ such that any line

crosses at most $O(\sqrt{s})$ triangles in the partition.

This construction can be used recursively to construct a partition tree for S. The root of the tree contains the whole set S, and a triangle that contains all the points. We find a balanced simplicial partition of S of size $\sqrt{|S|}$. Each of the children of the root are associated with a set S_i from the simplicial partition, and the triangle Δ_i that contains the points in S_i. For each of the S_i's we find simplicial partitions of size $\sqrt{|S_i|}$, and continue until each leaf contains a constant number of points. The construction time is $O(N \log_2 N)$.

To answer a simplex range query, we start at the root. We take each of the triangles in the simplicial partition at the root and check if it is inside the query region, outside the query region, or intersects one of the lines that define the query. In the first case all points inside the triangle are reported, in the second case the triangle is discarded, and in the third case we recurse on the triangle. The number of triangles that the query can cross is bounded however, since each line crosses at most $O(|S|^{\frac{1}{4}})$ triangles at the root. The query time is $O(N^{\frac{1}{2}+\epsilon} + K)$, with the constant factor depending on the choice of ϵ.

Agarwal et. al. [AAE+98] give an external memory version of static partition trees that answers queries in $O(n^{\frac{1}{2}+\epsilon} + k)$ I/Os. To adapt this structure to our environment, we have to make it dynamic. Using a standard technique by Overmars ([Ove83]) for decomposable problems we can show that we can insert or delete points in a partition tree in $O(\log_2^2 N)$ I/Os, and answer simplex queries in $O(n^{\frac{1}{2}+\epsilon} + k)$ I/Os.

3.3.5 Improving the average query time.

Partition trees are not very useful in practice because the query time is $O(n^{\frac{1}{2}+\epsilon} + k)$ and the hidden constant factor becomes large if we chose a small ϵ. In this section we present two different approaches that are designed to improve the average query time.

Using Point Access Methods

There are a large number of access methods that have been proposed to index point data[GG98]. All these structures were designed to address *orthogonal* range queries, eg. a query expressed as a multidimensional hyper-rectangle. However, most of them can be easily modified to address non-orthogonal queries like simplex queries.

Recently, Goldstein at al. [GRS+97] presented an algorithm to answer simplex range queries using R-trees. The idea is to change the search procedure of the tree. In particular they gave efficient methods to test whether a linear constraint query region and a hyper-rectangle overlap. As mentioned in [GRS+97] this method is not only applicable to the R-tree family, but to other access methods as well.

We use this approach to answer the one dimensional MOR query in the dual Hough-X space (Figure 3.2). However it is not clear what structure would be more suitable here, given that the distribution of points in the dual space is highly skewed. We argue that an index structure based on kd-trees (like the LSD-tree [HSW89] and the hB^{Π}-tree [ELS95]) is more suitable than a method based on R-trees. The reason is that since R-trees try to cluster data points into squarish regions [KF93], they will split using only one dimension (the intercept). On the other hand a kd-tree based method will use both dimensions to split (see Figure 3.3). Thus it is expected to have better performance for the MOR query.

A Query Approximation Algorithm.

A different approach is based on a *query approximation* idea using the Hough-Y dual plane. In general, the b coordinate can be computed at different horizontal ($y = y_r$) lines. The query region is described by the intersection of two half-plane queries (Figure 3.4). The one line intersects the line $n = \frac{1}{v_{max}}$ at the point $(t_{1q} - \frac{y_{2q}-y_r}{v_{max}}, \frac{1}{v_{max}})$ and the line $n = \frac{1}{v_{min}}$ at the point $(t_{1q} - \frac{y_{2q}-y_r}{v_{min}}, \frac{1}{v_{min}})$. Similarly the other line that defines the query intersects the horizontal lines at $(t_{2q} - \frac{y_{1q}-y_r}{v_{max}}, \frac{1}{v_{max}})$ and $(t_{2q} - \frac{y_{1q}-y_r}{v_{min}}, \frac{1}{v_{min}})$.

Since access methods are more efficient for rectangle queries, suppose that we approxi-

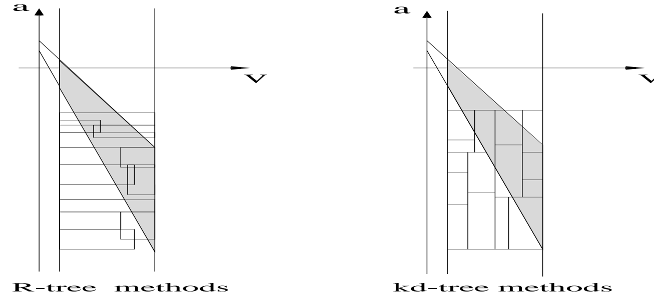


Figure 3.3: Data regions for R-tree like and kd-tree like methods .

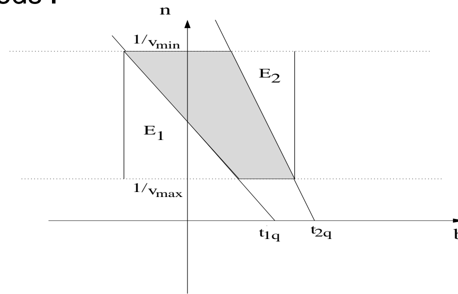


Figure 3.4: Query in the dual Hough-Y plane.

mate the simplex query with a rectangular one. In Figure 3.4 the query rectangle will be $[(t_{1q} - \frac{y_{2q}-y_r}{v_{min}}, t_{2q} - \frac{y_{1q}-y_r}{v_{max}}), (\frac{1}{v_{max}}, \frac{1}{v_{min}})]$. Note that the query area is enlarged by the area $E = E_1 + E_2$ which is computed as:

$$E = \frac{1}{2}(\frac{v_{max} - v_{min}}{v_{min} \times v_{max}})^2(|y_{2q} - y_r| + |y_{1q} - y_r|) \tag{3.1}$$

We are interested in minimizing E since it represents a measure of the extra I/O's that an access method will have to perform for solving an one dimensional MOR query. E is based on both y_r (i.e. where the b coordinate is computed) and the query interval (y_{1q}, y_{2q}) which is unknown. Hence, we propose to keep c indices (where c is a small constant) at equidistant y_r's. All c indices contain the same information about the objects, but use different y_r's. The i-th index stores the b coordinates of the data points using $y = \frac{y_{max}}{c} \times i$, $i = 0,..,c-1$. Conceptually,

y_i serves as an "observation" element, and its corresponding index stores the data as observed from position y_i. We call the area between subsequent "observation" elements, a *subterrain*. A given one dimensional MOR query will be forwarded to the index(es) that minimize E. Since all 2-dimensional approximate queries have the same rectangle side $(\frac{1}{v_{max}}, \frac{1}{v_{min}})$ (Figure 3.4) the rectangle range search is equivalent to a simple range search on the b coordinate axis. Thus each of the c "observation" indices can simply be a B+-tree [Com79].

To process a general query interval $[y_{1q}, y_{2q}]$ we consider two cases depending on whether the query interval covers a subterrain:

(i) $y_{2q} - y_{1q} \leq \frac{y_{max}}{c}$: then it can be easily shown that area E is bounded by:

$$E \leq \frac{1}{2}(\frac{v_{max} - v_{min}}{v_{min} \times v_{max}})^2(\frac{y_{max}}{c}) \tag{3.2}$$

The query is processed at the index that minimizes $|y_{2q} - y_r| + |y_{1q} - y_r|$.

(ii) $y_{2q} - y_{1q} > \frac{y_{max}}{c}$: the query interval contains one or more subterrains, which implies that if a query is executed at a single observation index, area E becomes large. To bound E we index each subterrain, too. Each of the c subterrain indices records the time interval when a moving object was in the subterrain. Then the query is decomposed into a collection of smaller subqueries: one subquery per subterrain fully contained by the original query interval, and one subquery for each of the original query's endpoints. The subqueries at the endpoints fall to case (i) above, thus they can be answered with bounded E using an appropriate "observation" index. To index the intervals in each subterrain we could use an external memory Interval tree[AV96] which will answer a subterrain query optimally (i.e., $E = 0$). As a result, the original query can be answered with bounded E. Thus we have the following lemma:

Lemma 2 *The one dimensional MOR query can be answered in time $O(\log_B n + (K + K')/B)$, where K' is the approximation error, that is the number of objects that we retrieve and do not belong in the answer of the original query. The space used is $O(cn)$ where c is a small constant, and the update is $O(c \log_B n)$.*

Note that assuming that the points are distributed uniformly over the b-axis, then the approximation

error is bounded by $1/c$, eg. $K' = O(1/c)$.

3.3.6 Achieving Logarithmic Query Time

For many applications, the relative positions of the moving objects do not change often. Consider for example the case where objects are moving very slowly, or with approximately the same velocity. In this case the lines in the time-space plane do not cross until well forward in the future. If we restrict our queries to occur before the first time that a point overtakes (passes) another, the original problem is equivalent to 1-dimensional range searching.

This is one of our motivations to consider a restricted version of the original problem, namely, to index mobile objects in a bounded time interval T in the future. As we have seen, there exist lower bounds for the original problem that show that we cannot achieve query time better than $\Omega(\sqrt{n})$ given linear space. However, using the above restriction, we achieve a logarithmic query time, with space that can be quadratic in the worst case but is expected to be linear in practice.

Formally, the problem we are considering in this section is the following: given a set of objects that are moving on a line, and a time limit T, find all the objects that lie in the segment $[y_l, y_r]$ at time t_q (where $t_0 \leq t_q \leq t_0 + T$). Equivalently, this is a standard one dimensional MOR query where $t_{1_q} = t_{2_q}$. We will call it an one dimensional MOR1 query.

Our method is to find all the times when an object overtakes another. These events correspond to line segment crosses in the time-space plane. Note that between two consecutive crossing events the relative ordering of the objects on the plane remains the same.

First we show the following lemma:

Lemma 3 *If we have the relative ordering of all the objects at time t_q, the position of the objects at time t_c that corresponds to the closest crossing event before t_q, and the speed of the objects, we can find the objects that are in $[y_l, y_r]$ in $O(\log_2 N + K)$ time.*

Proof: Assume that the objects are $\{p_1, p_2, \ldots, p_N\}$, where p_i has a position y_i at time t_c and a velocity v_i. Without loss of generality, assume that, at time t_q, the relative order of the objects

from left to right is $p_1, p_2, \ldots, p_N$.

Store the objects in a binary tree, sorted by their original positions. The root of the tree, point p_i, is going to be at position $y_i + v_i \times t_q$ at time t_q. Since the objects in the binary tree are stored by order at the time t_q, if $y_i + v_i \times t_q < y_l$ then this is also true for all the objects to the left child of the root, in which case we eliminate the left child and recurse on the right child. Otherwise we recurse on the left child of the tree. Thus in $O(\log_2 N)$ time we can find the positions of y_l and y_r relative to the objects at time t_q, and we report the objects that lie between. ■

The following lemma finds all the crossings of points efficiently.

Lemma 4 *We can find all the crossings of objects in time* $O(N \log_2 N + M \log_2 M)$*, where* M *is the number of crosses in the time period* $[0, T]$.

Proof: Let $\{p_1, \ldots, p_N\}$ be the ordering of the N objects at time 0. At time T, the position of object i is $y_i + v_i \times T$. To find the ordering of the objects at time T we have to sort their positions. Let $\{p_{t(1)}, \ldots, p_{t(N)}\}$ be the ordering of the same N objects at time T. Then objects i and j cross if and only if $t(j) < t(i)$.

Keep the objects in a linked list, in the same order they were at time $t = 0$. Scan the sorted list of objects at time T. Find object $p_{t(1)}$ in the list. This object crosses all the objects ahead of it in the list. After reporting these crosses, we remove it from the list, and repeat this process with the next point. This procedure reports all the crossings in $O(N + M)$ time. After all the crossings are reported we can find when each occurs and sort them on their time attributes. ■

These M crosses define M ordered lists of the N objects. Each two consecutive lists differ in exactly two positions, the positions that correspond to the objects that cross. The total sum of the differences between consecutive lists is therefore $O(M)$. In the next lemma we show how we can efficiently store and search these lists in external memory.

Lemma 5 *We can store the* $O(M)$ *ordered lists of* N *objects in* $O(n + m)$ *blocks and perform a search on any list in* $O(\log_B (n + m))$ *I/O's, where* $n = \frac{N}{B}$ *and* $m = \frac{M}{B}$.

Proof: Let $L(t)$ be the list of objects at time t. Consider CS = t_1,..., t_M the ordered sequence of the time instants where crossings occur during the interval (0,T). The problem of storing the M ordered lists $L(t_1)$ through $L(t_M)$ can be "visualized" as storing the history of a list $L(t)$ that evolves over time, i.e., a partial persistence problem [DSS+86]. That is, list $L(t)$ starts from an initial state $L(0)$ and then evolves through consecutive states $L(t_1)$, $L(t_2)$,..., $L(t_M)$, where $L(t_{i+1})$ is produced from $L(t_i)$ by applying the crossing that occurred at t_{i+1} (i=0,...,M-1, and t_0=0).

A common characteristic in the list evolution is that each $L(t)$ has exactly N positions, namely positions 1 through N, where position j stores the $j-th$ element of $L(t)$. To perform a binary search on a given $L(t)$ we could implement it using a binary tree with N nodes, where each node is numbered by a position (the root node corresponds to the middle position in the list and so on) and holds the element of $L(t)$ at that position. One obvious solution to the problem would be to store the binary tree of the original list $L(0)$ and the binary tree of each $L(t_i)$ for all t_i in CS. Then, a query about list $L(t)$ is addressed by using the binary tree of $L(t_i)$, where t_i is the largest instant in CS that is less or equal to t. While this achieves $O(\log_2(N+M))$ query time, it uses $O(MN)$ space.

To reduce the space to $O(N+M)$ we must take advantage of the fact that subsequent lists do not differ much. A main-memory solution to this problem appears in [Col86]. Here we present an efficient external memory solution. In particular, we first embed the binary tree structure inside a B-tree. This is easily done since the structure of the list (and its corresponding binary tree) does not change over time. Consider for example tree $B(0)$ that corresponds to the initial list $L(0)$. Tree $B(0)$ uses $O(n)$ nodes where each node can hold B entries. An entry is now a record: (*position, occupant, pointer, t*), where *position* corresponds to a position in the list, *occupant* contains the element at that position, $pointer$ points to a child node and t corresponds to the time this element was at that position, in this case t=0.

Conceptually, each B-tree node is permanently assigned B positions and is responsible for storing the occupants of these positions. Consider the evolution of such a node s through trees $B(0)$,$B(t_1)$,$B(t_2)$,...,$B(t_M)$. An obvious way to store this evolution is to store a copy of $s(0)$ and a "log" of changes that happen on the occupants of node s at later times. A change is simply

another record that stores the position where a change occurred, the new occupant and the time of the change. To achieve fast access to $s(t)$ we do not allow the "log" to get too large. Every $O(B)$ changes (in practice when the log fills one or two pages) we store a new, current copy of s. If we consider the history of node s independently, we can have an auxiliary array with records $(time, pointer)$ that point to the various copies of node s. Locating the appropriate node $s(t)$ takes $O(\log_B m)$ time (first find the record in the auxiliary array with the largest timestamp that is less or equal to t and then we access the appropriate copy of s and probably a (constant) number of "log" pages). The space remains $O(n+m)$ since every new node copy is amortized over the $O(B)$ changes in the "log".

While this solution works nicely for the history of a given B-tree node, it would lead to $O(\log_B n \times \log_B m)$ search (since finding the appropriate version of a child node, when searching the B-tree, requires $O(\log_B m)$ search in the child node's history). Instead of using the auxiliary array to index the copies of node s we post such entries as changes in the history of the parent node p. Assume that node s is pointed to by the record at position l in node p. When a new copy of node s is created, a new record is added on the "log" of p that has the same position l, but a pointer to the new copy of s and the current time. Since new node copies are added after $O(B)$ changes, the overall space remains $O(n+m)$. The query time is reduced to $O(\log_B(n+m))$ since performing a binary search on list $L(t)$ is equivalent to searching a path of $B(t)$; locating the root of $B(t)$ takes $O(\log_B m)$ (searching the history of the B-tree root node) while all other nodes of $B(t)$ are found in $O(\log_B n)$ using the appropriate parent to child pointers. ■

The following theorem follows from the previous lemmas:

Theorem 3 *Given N objects and a time limit T, a one dimensional MOR1 query can be answered in time $\log_B(n+m)$ using space $O(n+m)$, where $m = \frac{M}{B}$ and M is the number of crosses of objects in the time limit T.*

To solve the problem of answering queries within a time interval T into the future, we stagger the construction of our data structure. Thus, at time t_0 we construct a data structure that will answer queries in the time interval $[t_0, t_0 + 2T]$, and at time $t_0 + iT$ we construct a data structure that will answer queries in the time interval $[t_0 + (i+1)T, t_0 + (i+2)T]$

Our approach works for any value of T. If the time limit is set too large however, all pairs of objects may cross, in which case the size of the data structure will be quadratic. It is therefore important to set the time limit appropriately so that only approximately a linear number of crossings occur. Fortunately, in practice it is often true that many objects move with approximately equal speeds (one example is cars on a highway) and therefore do not cross very often.

3.4 Indexing in two dimensions

In this section we consider the problem of mobile objects in the plane. Again we consider only "moving" objects, namely objects with a speed between v_{min} and v_{max}. We assume that objects move in the (x, y) plane inside the finite terrain $[(0, x_{max})(0, y_{max})]$. The initial location of the object o_i is (x_{i_0}, y_{i_0}) and its velocity is a vector $\vec{v} = (v_x, v_y)$.

We distinguish two important cases. The first considers objects moving in the plane but their movement is restricted on using a given collection of routes (roads) on the finite terrain. Due to its restriction, we call this case the 1.5-dimensional problem. There is a strong motivation for such an environment; for the applications we have in mind, objects (cars, airplanes etc.) move on a network of specific routes (highways, airways). In the second case the objects move anywhere in the plane.

3.4.1 The 1.5-dimensional problem

The 1.5-dimensional problem can be reduced to a number of 1-dimensional queries. In particular, we propose representing each predefined route as a sequence of connected (straight) line segments. The positions of these line segments on the terrain are indexed by a standard SAM. (Maintaining this SAM does not introduce a large overhead since for most practical applications: (a) the number of routes is much smaller than the number of objects moving on them, (b) each route can be approximated by a small number of straight lines, and, (c) new routes are added rather infrequently.) Indexing the objects moving on a given route is an 1-dimensional model and will use techniques from the previous section.

Given a two dimensional MOR query, the above SAM identifies the intersection of the routes with the query's spatial predicate (the rectangle $[x_{1q}, x_{2q}] \times [y_{1q}, y_{2q}]$). Since each route is modeled as a sequence of line segments, the intersection of the route and the query's spatial predicate is also a set of line segments, possibly disconnected. Each such intersection corresponds to the spatial predicate of an 1-dimensional query for this route. In this setting we assume that when routes intersect, objects remain in the route previously traveled (otherwise an update is issued).

3.4.2 The 2-dimensional problem

The full 2-dimensional problem (i.e., allowing objects to move anywhere on the finite terrain) is more difficult. As with the 1-dimensional case, we discuss different representations of the problem and we propose methods to address the two dimensional MOR query.

In the space-time representation the trajectories of the mobile objects are lines in the space. The lines can be computed by the motion information of each object. The two dimensional MOR query is expressed as a cube in the 3-dimensional (x, y, t) space and the answer is the set of objects with lines that cross the query cube.

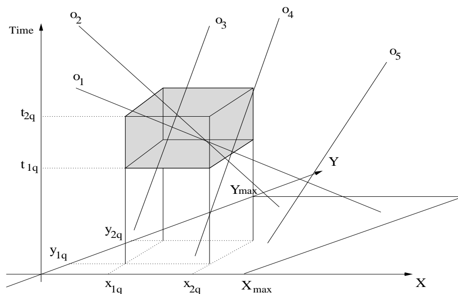


Figure 3.5: Trajectories and query in (x, y, t) plane.

Algorithms that are applied directly to the time-space representation do not work well in one-dimension, so the performance is likely to be even worse in two dimensions. Unfortunately we cannot use directly the dual transformations of the previous section, since these transforms map a hyper-plane in the space into a point and vice-versa, where here we have lines. We point out that the problems with lines in the space are much harder than lines in the plane. The reason is that a

line in space has 4 degrees of freedom and therefore taking the dual we jump to a 4-dimensional space. To get the dual we project the lines on the (x, t) and (y, t) planes and then take the duals for the two lines on these planes. Thus now a line can be represented by the 4-dimensional point (v_x, a_x, v_y, a_y), where the v_x and v_y are the slopes of the lines on the (x, t) and (y, t) planes and the a_x and a_y are the intercepts respectively.

The two dimensional MOR query is mapped to a simplex query in the dual space. This query is the intersection of four 3-d hyper-planes and the projection of the query to (t, x) and to (t, y) planes are wedges, as in the 1-dimensional case. Thus we can use a 4-dimensional partition tree (section 3.4) and answer the MOR query in $O(n^{0.75+\epsilon} + k)$ I/O's. A simple approach to solve the 4-dimensional problem is to use an index based on the $kd-tree$. An alternative approach is to decompose the motion of the object into two independent motions, one in the x-axis and the other in the y-axis. For each axis we can use the methods for the 1-dimensional case and answer two 1-dimensional MOR queries. We must then take the intersection of the two answers to find the answer to the initial query. This method allows us to use the algorithms for the 1-dimensional case.

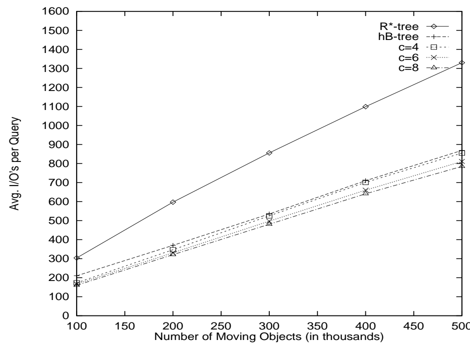


Figure 3.6: Query Performance for 10% Queries.

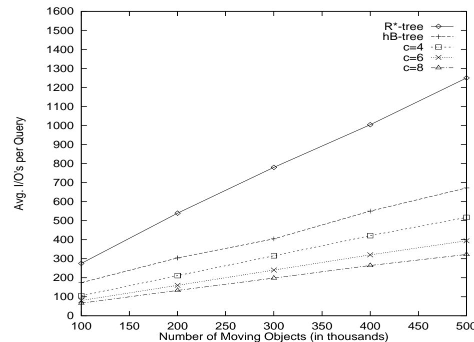


Figure 3.7: Query Performance for 1% Queries.

3.5 A Performance Study

We present results for the one dimensional MOR query, comparing our query approximation approach, the kd-tree method and a traditional R-tree based approach. First we describe the way experimental data is generated. At time $t = 0$ we generated the initial locations of N mobile objects uniformly distributed on the terrain $[0, 1000]$. We varied N from $100k$ to $500k$. The speeds were generated uniformly from $v_{min} = 0.16$ to $v_{max} = 1.66$ and the direction randomly positive or negative.[2] Then objects start moving. When an object reaches a border simply it changes its direction. At each time instant we choose 200 objects randomly and we randomly change their speed and/or direction. We generate 10 different time instants that represent the times when queries are executed. At each such time instant we execute 200 random queries, where the length of the y-range is chosen uniformly between 0 and $YQMAX$ and the length of the time range between 0 and TW. We actually generated two sets of queries. One set of large queries with $YQMAX = 150$ and $TW = 60$ and one set of small queries with $YQMAX = 10$ and $TW = 20$. The first set of queries has average cardinality almost 10% and the second one close to 1%. We run this scenario using a particular access method for 2000 time instants.

To verify that indexing mobile objects as line segments is not efficient, we stored the trajectories in an R*-tree. We fixed the page size to 4096 bytes. To represent a line segment in an R*-tree we used four 4-byte numbers (the two end points) and one more number as a pointer to the real object, resulting in a page capacity of $B = 204$. For the B+-tree we used one 4-byte number to represent the b-coordinate, one number for the speed and another one for the pointer, so the page capacity was $B = 341$. We index each record using only the b-coordinate but using the speed of each object we can identify the objects that correspond to the real answer and report only these objects. The same page capacity used and for the hB^{Π}-tree. However, each hB^{Π}-tree page reserves some space for internal structural data. We consider a simple buffering scheme for the results we present here. For each tree we buffer the path from the root to a leaf node, thus the buffer size is only 3 or 4 pages. For the queries we always clear the buffer pool before we run a query. An update is performed when the motion information of an object changes.

[2] Note that 0.16 miles/min is equal to 10 miles/hour and 1.66 miles/min is equal to 100 miles/hour.

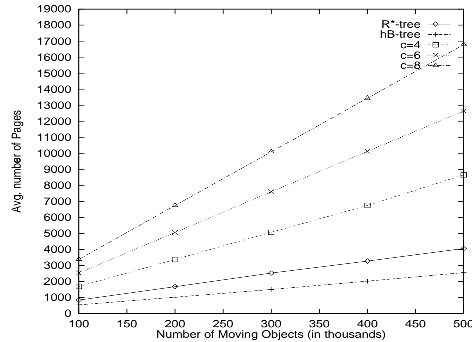


Figure 3.8: Space Consumption.

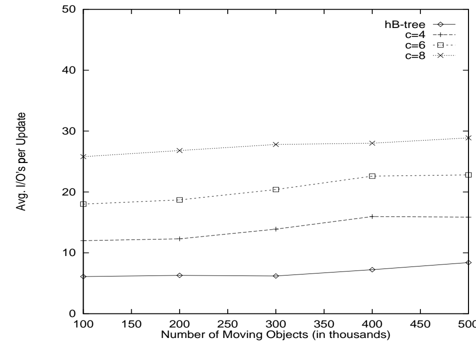


Figure 3.9: Update Performance.

In Figure 3.6 we present the results for the average number of I/O's per query for 10% queries and in figure 3.7 for 1% queries. The approximation method used $c = 4, 6$ and 8 B+-trees. As anticipated, the line segments method with R*-trees has the worst performance. Also, the approximation method outperforms the hB^{Π}-tree for small queries and it is slightly better for large queries.

In Figures 3.8,3.9 we plot the space consumption and the average number of I/O's per update respectively. We did not report the update performance for the R*-tree method because it was very high (more than 90 I/O's per update). The update and space performance of the hB^{Π}-tree is better than the other methods since its objects are stored only once and better clustered than the R*-tree. The update performance of the hB^{Π}-tree and the approximation approach remain constant for different number of mobile objects. The space of all methods is linear to the number of objects. The approximation approach uses more space due to the use of c observation indices. There is a tradeoff between c and the query/update performance.

3.6 Related Work in Indexing Continuously Moving Objects

The problem of indexing mobile objects is novel; we are not aware of any other related work except [TUW98] where a method to index mobile objects based on the PMR-quadtree is presented. However this approach has large space and update overhead.

Mobility in a geographic system is addressed in [SY91] where the aim is to map close points in space to adjacent disks so that collision detection queries are optimized.

The issue of mobility and maintenance of a number of configuration functions among continuously moving objects has been addressed by Basch et al. in [BGH97]. Such functions are the convex hull, the closest pair and the minimum spanning tree. They propose a framework to transform a static data structure into a *kinetic* data structure (KDS) that maintains an attribute of interest for a set of mobile objects and they give a number of criteria for the quality of such structures. The key structure is an event queue that contains events corresponding to times where the value of the configuration function (may) change. This event queue is the interface between the data structure and the mobile objects. All these structures are main memory data structures. The major results of this chapter are presented in [KGT99]. More recently, there are two other works that follow the ideas presented here to index moving objects. In [SJL+00] a structure based on the R-Tree is presented for indexing objects moving in two and three dimensions. The basic idea is to parameterize the index structure using the velocity vectors of the moving objects. Thus, instead of MBRs, each node in the tree stores "moving" MBRs and given a query, the state of the tree is computed on line by projecting the MBRs at the specified time instant in the future. In another work [AAE00], methods to index two dimensional moving objects are presented. They use external memory partition trees and kinetic range trees to answer range and nearest neighbor queries with good worst-case query performance.

3.7 Summary

Indexing mobile objects is a problem motivated by real life applications. We study the one and two dimensional versions of the problem. For the one dimensional case, we give a dynamic, external memory algorithm with guaranteed worst case performance and linear space. We also give a practical approximation algorithm also in the dynamic, external memory setting, which has linear space and expected logarithmic query time. Finally we give an algorithm with guaranteed logarithmic query time for a restricted version of the problem. We also extend some of our results into two dimensions. First we consider the case where objects move in 2–dimensional networks

of 1–dimensional routes. In this case we can effectively apply our 1–dimensional algorithms. We also consider objects that move on a plane, and we discuss extensions of our techniques to two dimensions.

Chapter 4

Temporal Hashing in Temporal and Spatiotemporal Databases

4.1 Introduction

In this chapter we discuss the problem of answering membership queries in a temporal or spatiotemporal environment. Conventional membership queries over a set of objects have been addressed through hashing. Hashing can be applied either as a main memory scheme (all data fits in main-memory [DKM+88, FNS+92]) or in database systems (where data is stored on disk [Lit80]). Its latter form is called external hashing [EN94, R97]. For every object in the set, a hashing function computes the bucket where the object is stored. Each bucket has initially the size of a page. Ideally, each distinct oid should be mapped to a separate bucket, however this is unrealistic as the universe of oids is usually much larger than the number of buckets allocated by the hashing scheme. If a bucket cannot accommodate the oids mapped to it in the space assigned to the bucket, an overflow occurs. Overflows are dealt in various ways, including rehashing (where another bucket is found using another hashing scheme) and/or chaining (create a chain of pages under the overflown bucket).

If no overflows are present, finding whether a given oid is in the set is trivial: simply compute the hashing function for the queried oid and visit the appropriate bucket. If the object is in

the set it should be in that bucket. Hence, if the hashing scheme is perfect, membership queries are answered in $O(1)$ steps (just one I/O to access the page of the bucket). Overflows however complicate the situation. If data is not known in advance, the worst case query performance of hashing is large. It is linear in the size of the set since all oids could be mapped to the same bucket if a bad hashing scheme is used. Nevertheless, practice has shown that in the absence of pathological data, good hashing schemes with few overflows and constant average case query performance (usually each bucket has size of one or two pages) exist.

Static hashing refers to schemes that use a predefined collection of buckets. This is inefficient if the set of objects is allowed to change (by adding or deleting objects from it). Instead, a dynamic hashing scheme has the property of allocating space proportional to the size of the set. Various external dynamic hashing schemes [Lar78, FNP+79, ED88] have been proposed, among which linear hashing [Lit80] appears to be the most commonly used.

Even if the set evolves, traditional dynamic hashing is ephemeral, i.e., it answers membership queries on the most current state of the set. The problem we discuss here is more general. We assume that changes of set S are timestamped by the time instant when they occurred. We are interested in answering membership queries for any state that set S exhibited. Let S(t) denote the state (collection of objects) set S had at time t. Then the membership query has a temporal predicate as in: "given oid k and time t find whether k was in S(t)". We term this problem as Temporal Hashing and the new query as temporal membership query.

Assume that for every time t when S(t) changes (by adding/deleting objects) we could have a good ephemeral dynamic hashing scheme h(t) that maps efficiently (with few overflows) the oids in S(t) into a collection of buckets b(t). One straightforward solution to the temporal hashing problem would be to separately store each collection of buckets b(t) for each t. Answering a temporal membership query for oid k and time t requires first applying h(t) on k and then accessing the appropriate bucket of b(t). This would provide an excellent query performance (as it uses hashing scheme h(t) for each t), but the space requirements are prohibitively large. If n denotes the number of changes in S's evolution, flashing each b(t) on the disk could easily create $O(n^2)$ space.

Instead we propose a solution that has similar query performance as above (with a small

overhead) but uses space linear in n. We call our solution partially persistent hashing as it reduces the original problem into a collection of partially persistent sub-problems. We apply two approaches for solving these sub-problems. The first approach ”sees” each sub-problem as an evolving subset of set S and is based on the Snapshot Index [TK95]. The second approach ”sees” each sub-problem as an evolving sublist. In both cases, the partially persistent hashing scheme ”observes” and stores the evolution of the ephemeral hashing h(t).

We compare partially persistent hashing with three other approaches. The first one uses a traditional dynamic hashing scheme to map all oids ever created during the evolution of S(t). This solution does not distinguish among the many copies of the same oid k that may have been created as time proceeds. A given oid k can be added and deleted to/from S many times, creating copies of k each associated with a different, disjoint time interval. Because all such copies will be hashed on the same bucket, bucket reorganizations will not solve the problem (this was also observed in [AS]). Large bucket sizes will eventually deteriorate performance especially as the number of copies increases. The second approach assumes that a B+ tree is used to index each S(t) and makes this B+tree partially persistent [BGO+96, VV97, LS89]. The third approach sees each oid-interval combination as a multidimensional object and uses an R*-tree for storing it. Our experiments show that the partially persistent hashing outperforms the other three competitors in membership query performance while having a minimal space overhead.

The rest of the chapter is organized as follows: section 4.2 describes the basics of Linear Hashing. The description of partially persistent linear hashing appears in section 4.3. Performance comparisons are presented in section 4.4, while conclusions appear in section 4.4.3.

4.2 Background

In chapter 1 we discussed the Snapshot index and we described how this method can be used to answer the pure-snaphsot query efficiently. Next we present the basics for Linear Hashing.

4.2.1 Linear Hashing

Linear Hashing (LH) is a dynamic hashing scheme that adjusts gracefully to object insertions and deletions. The scheme uses a collection of buckets that grows or shrinks one bucket at a time. Overflows are handled by creating a chain of pages under the overflowed bucket. The hashing function changes dynamically and at any given instant at most two hashing functions are used.

More specifically, let U be the universe of oids and $h_0 : U- > \{0, ..., M-1\}$ be the initial hashing function that is used to load set S into M buckets (for example: $h_0(oid) = oid \;\; mod \;\; M$). Assume that an empty page is initially assigned to each bucket. Insertions and deletions of oids are performed using h_0 until a bucket overflow happens. When the first overflow occurs (it can occur in any bucket), the first bucket in the LH file, bucket 0, is split (rehashed) into two buckets: the original bucket 0 and a new bucket M, which is attached at the end of the LH file. A new empty page is also added in the overflown bucket to accommodate the overflow. The oids originally mapped into bucket 0 (using function h_0) are now distributed between buckets 0 and M using a new hashing function $h_1(oid)$. The next bucket overflow triggers a new split that will attach a new bucket M+1 and the contents of bucket 1 will be distributed using h_1 between buckets 1 and M+1. A crucial property of h_1 is that any oids that were originally mapped by h_0 to bucket j should be remapped either to bucket j or bucket j+M. This is a necessary property for linear hashing to work. An example of such hashing function is: $h_1(oid) = oid \;\; mod \;\; 2M$.

Further bucket overflows will cause additional bucket splits in a linear bucket-number order. A variable p indicates which is the bucket to be split next. Conceptually the value of p denotes which of the two hashing functions that may be enabled at any given time, applies to what buckets. Initially p=0, which means that only one hashing function (h_0) is used and applies to all buckets in the LH file. After the first overflow in the above example, p=1 and h_1 is introduced. Suppose that an object with oid k is inserted after the second overflow (i.e., when p=2). First the older hashing function (h_0) is applied on k. If then the bucket $h_0(k)$ has not been split yet then k is stored in that bucket. Otherwise the bucket provided by h_0 has already been split and the newer hashing function (h_1) is used; oid k is stored in bucket $h_1(k)$. Searching for an oid is similar, that is, both hashing functions may be involved.

After enough overflows, all original M buckets will be split. This marks the end of splitting- round 0. During round 0, p went subsequently from bucket 0 to bucket M-1. At the end of round 0 the LH file has a total of 2M buckets. Hashing function h_0 is no longer needed as all 2M buckets can be addressed by hashing function h_1 (note: $h_1 : U-> \{0, ..., 2M - 1\}$). Variable p is reset to 0 and a new round, namely splitting-round 1, is started. The next overflow (in any of the 2M buckets) will introduce hashing function $h_2(oid) = oid \ \ mod \ \ 2^2M$. This round will last until bucket 2M-1 is split. In general, round i starts with p = 0, buckets $\{0, ..., 2^iM - 1\}$ and hashing functions $h_i(oid)$ and $h_{i+1}(oid)$. The round ends when all 2^iM buckets are split. For our purposes we use $h_i(oid) = oid \ \ mod \ \ 2^iM$. Functions h_j, $j = 1, ...,$ are called split functions of h_0. A split function h_j has the properties: (i) $h_j : U-> \{0, ..., 2^jM - 1\}$ and (ii) for any oid, either $h_j(oid) = h_{j-1}(oid)$ or $h_j(oid) = h_{j-1}(oid) + 2^{j-1}M$.

At any given time, the linear hashing scheme is completely identified by the round number and variable p. Given round i and variable p, searching for oid k is performed using h_i if $h_i(k) \geq p$; otherwise h_{i+1} is used. During round i the value of p is increased by one at each overflow; when $p = 2^iM$ the next round $i + 1$ starts and p is reset to 0.

A split performed whenever a bucket overflow occurs is an uncontrolled split. Let l denote the LH file's load factor, i.e., $l = |S|/(B|b|)$ where $|S|$ is the current size of set S and $|b|$ the current number of buckets in the LH file. The load factor achieved by uncontrolled splits is usually between 50-70%, depending on the page size and the oid distribution. In practice, a higher storage utilization is achieved if a split is triggered not by an overflow, but when the load factor l becomes greater than some upper threshold g. This is called a controlled split and can typically achieve 95% utilization. (Note that the split is now independent of bucket overflows. Other controlled schemes exist where a split is delayed until both the threshold condition holds and an overflow occurs). Deletions in set S will cause the LH file to shrink. Buckets that have been split can be recombined if the load factor falls below some lower threshold f. Then two buckets are merged together; this operation is the reverse of splitting and occurs in reverse linear order. Practical values for f and g are 0.7 and 0.9, respectively. In our experiments and analysis we use the controlled version of linear hashing.

4.3 Partially Persistent Linear Hashing

We will apply the partially persistent methodology to an ephemeral linear hashing scheme. We first describe the evolving-set approach which is based on the Snapshot Index; the evolving-list approach will follow.

4.3.1 The Evolving-Set Approach

Using partial persistence, the temporal hashing problem will be reduced into a number of sub- problems for which efficient solutions are known. Assume that an ephemeral linear hashing scheme (as the one described in the previous section) is used to map the objects of S(t). As S(t) evolves with time the hashing scheme is a function of time, too. Let LH(t) denote the linear hashing file as it is at time t. There are two basic time-dependent parameters that identify LH(t) for each t, namely i(t) and p(t). Parameter i(t) is the round number at time t. The value of parameter p(t) identifies the next bucket to be split.

An interesting property of linear hashing is that buckets are reused; when round i+1 starts it has double the number of buckets of round i but the first half of the bucket sequence is the same since new buckets are appended in the end of the file. Let b_{total} denote the longest sequence of buckets ever used during the evolution of S(t) and assume that b_{total} consists of buckets: $0, 1, 2, ..., 2^q M - 1$. The above observation implies that for all t, b(t) (the collection of buckets used at time t) is a prefix of b_{total}. In addition $i(t) \leq q, \forall q$.

Consider bucket b_j from the sequence b_{total} and observe the collection of objects that are stored in this bucket as time proceeds. The state of bucket b_j at time t, namely $b_j(t)$, is the set of oids stored in this bucket at t. Let $|b_j(t)|$ denote the number of oids in $b_j(t)$. If all states $b_j(t)$ can somehow be reconstructed for each bucket b_j, answering a temporal membership query for oid k at time t can be answered in two steps:

(1) find which bucket b_j, oid k would have been mapped by the hashing scheme at t, and,

(2) search through the contents of $b_j(t)$ until k is found.

The first step requires identifying the hashing scheme used at t. The evolution of the hashing scheme LH(t) is easily maintained if a record of the form $< t, i(t), p(t) >$ is appended to an array H, for those instants t where the values of i(t) and/or p(t) change. Given any t, the hashing function used at t is identified by simply locating t inside the time-ordered H in a logarithmic search.

The second step implies accessing $b_j(t)$. The obvious way would store each $b_j(t)$, for those times that $b_j(t)$ changed. As explained earlier this would easily create quadratic space requirements. The updating per change would also suffer since the I/O required for storing the current state of b_j would be proportional to the bucket's current size, namely $O(|b_j(t)|/B)$.

By observing the evolution of bucket b_j we note that its state changes as an evolving set by adding/deleting oids. Each such change can be timestamped with the time it occurred. At times the ephemeral linear hashing scheme may apply a rehashing procedure that remaps the current contents of bucket b_j to bucket b_j and some new bucket b_r. Assume that such rehashing occurred at time t' and its result is a move of v oids from b_j to b_r. For the evolution of b_j (b_r), this rehashing is viewed as a deletion (respectively addition) of the v oids at time t' , i.e., all such deletions (additions) are timestamped with the same time t' for the corresponding object's evolution.

Figure 4.1 shows an example of the ephemeral hashing scheme at two different time instants. For simplicity M = 5 and B = 2. Figure 4.2 shows the corresponding evolution of set S and the evolutions of various buckets. Here we assumed controlled splitting with an upper threshold g = 0.9. At time t = 21 the addition of oid 8 triggers a split since $|S| = 10$ oids and $M = 5, l > g$ (at the same time oid 8 causes an overflow on bucket 3). Thus the contents of bucket 0 are rehashed between bucket 0 and bucket 5. As a result oid 15 is moved to bucket 5. For bucket's 0 evolution this change is considered as a deletion at t = 21 but for bucket 5 it is an addition of oid 15 at the same t = 21.

If b_j(t) is available, searching through its contents for oid k is performed by a linear search. This process is lower bounded by $O(|b_j(t)|/B)$ I/O's since these many pages are at least needed for storing b_j(t). (This is similar with traditional hashing where a query about some oid is translated into searching the pages of a bucket; this search is also linear and continues until the oid is found or all the bucket's pages are searched.) What is therefore needed is a method which for any given t

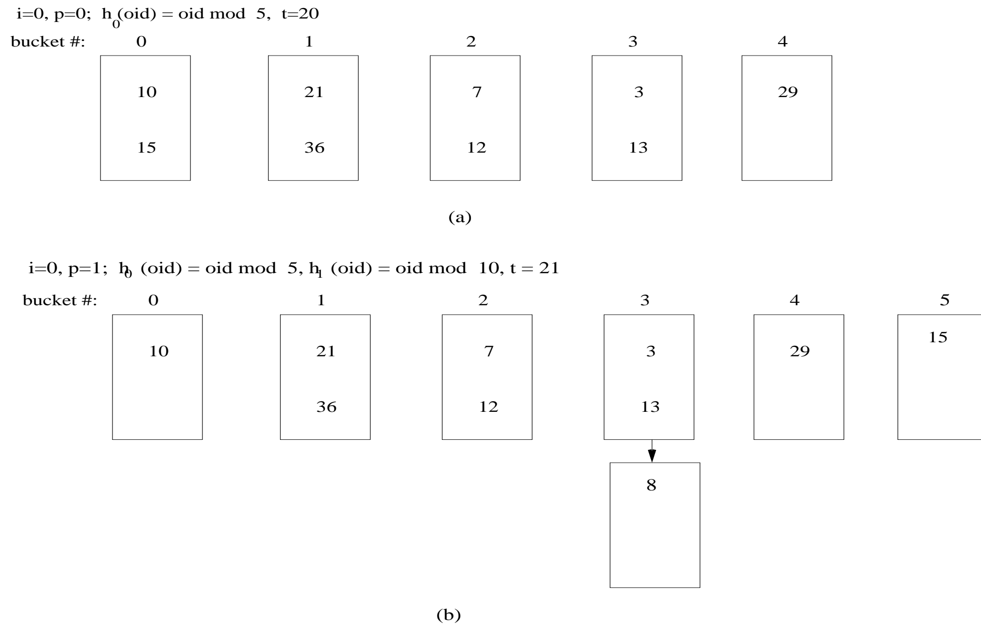


Figure 4.1: Two instants in the evolution of an ephemeral hashing scheme. (a) Until time t = 20, no split has occurred and p = 0. (b) At t = 21, oid 8 is mapped to bucket 3 and causes a controlled split. Bucket 0 is rehashed using h_1 and p = 1.

can reconstruct b_j(t) with effort proportional to $|b_j(t)|/B$ I/O's. Since every bucket b_j behaves like a set evolving over time, the Snapshot Index [TK95] can be used to store the evolution of each b_j and reconstruct any b_j(t) with the required efficiency.

We can thus conclude that given an evolving set S, partially persistent linear hashing answers a temporal membership query about oid k at time t, with almost the same query time efficiency (plus a small overhead) as if a separate ephemeral hashing scheme existed on each S(t). A good ephemeral hashing scheme for S(t) would require an expected $O(1)$ I/O's for answering a membership query. This means that on average each bucket $b_j(t)$ used for S(t) would be of limited size, or equivalently, $|b_j(t)|/B$ corresponds to just a few pages (in practice one or two pages). In perspective, partially persistent linear hashing will reconstruct $b_j(t)$ in $O(|b_j(t)|/B)$ I/O's, which from the above is expected $O(1)$.

The small overhead incurred by partially persistent linear hashing is because it stores the whole history of S's evolution and not just a single state S(t). Array H stores an entry every time

a split occurs. Even if all changes are new oid additions, the number of splits is upper bounded by $O(n/B)$. Hence array H indexes at most $O(n/B^2)$ pages and searching H takes $O(log_B(n/B^2))$ I/O's.

Having identified the hashing function the appropriate bucket b_j is pinpointed. Then time t must be searched in the time-tree associated with this bucket. The overhead implied by this search is bounded by $O(log_B(n_j/B))$ where n_j corresponds to the number of changes recorded in bucket b_j's history. In practice, we expect that the n changes in S's evolution will be concentrated on the first few in the b_{total} bucket sequence, simply because a prefix of this sequence is always used. If we assume that most of S's history is recorded in the first 2^iM buckets (for some i), n_j behaves as $O(n/(2^iM))$ and therefore searching b_j's time-tree is rather fast.

A logarithmic overhead that is proportional to the number of changes n, is a common characteristic in query time of all temporal indices that use partial persistence. The MultiVersion B-Tree (MVBT) proposed by Becker et al. (or the MultiVersion Access Structure proposed by Verman and Varma) will answer a temporal membership query about oid k on time t, in $O(log_B(n/B))$ I/O's. We note that MVBT's logarithmic bound contains two searches. First, the appropriate B-tree that indexes S(t) is found. This is a fast search and is similar to identifying the hashing function and which bucket to search in partially persistent linear hashing. The second logarithmic search in MVBT is for finding k in the tree that indexes S(t) and is logarithmic on the size of S(t). Instead partially persistent linear hashing finds oid k in expected $O(1)$ I/O's.

Update and Space Analysis

Here we show that the partially persistent linear hashing scheme uses $O(n/B)$ space. An $O(1)$ amortized expected update processing per change can then be derived. For the space analysis we assume that the corresponding ephemeral linear hashing scheme LH(t) starts with M initial buckets, uses controlled splits and has upper threshold g. Assume for simplicity that set S evolves by only adding oids (oid additions create new records and hence more copying; deletions do not create splits).

Clearly array H satisfies the space bound. Next we show that the space used by bucket

histories is also bounded by $O(n/B)$. Since for partially persistent linear hashing, each split creates artificial changes (the oids moved to a new bucket are deleted from the previous bucket and subsequently added to a new bucket as new records), it must be shown that the number of oid moves (copies) due to splits is still bounded by the number of real changes n. We will first use two lemmas that upper and lower bound such oid moves during the first round of splits.

Lemma 6 *For any N in* $1 \leq N \leq M$*, N splits occur after at least* $(M + N - 1)Bg$ *real object additions happen.*

Proof: Just before the N-th split we have exactly M+N-1 buckets and hence the load is $l = |S|/(B(M + N - 1))$. By definition, for the next controlled split to occur we must have $l > g$, which implies that $|S| > (M + N - 1)Bg$. ■

Lemma 7 *For any N in* $1 \leq N \leq M$*, N splits can create at most* $(M + N - 1)B + 1$ *oid copies.*

Proof: Assume that all buckets are initially empty. Observe that the number of oid copies made during a single round is at most equal to the number of real object additions. This is because during one round an oid is rehashed at most once. The number of real additions before a split increases with g (lower g triggers a split earlier, i.e., with less oids). Hence the most real additions occur when g = 1. The N-th controlled split is then triggered when the load exceeds 1. Since there are already M+N-1 buckets, this happens at the $((M + N - 1)B + 1)$-th object addition. ■

The basic theorem about space and updating follows.

Theorem 4 *Partially Persistent Linear Hashing uses space proportional to the total number of real changes and updating that is amortized expected* $O(1)$ *per change.*

Proof: As splits occur, linear hashing proceeds in rounds. In the first round variable p starts from bucket 0 and in the end of the round it reaches bucket M-1. At that point 2M buckets are used and all copies (remappings) from oids of the first round have been created. Since M splits have occurred, lemma 7 implies that there must have been at most (2M-1)B+1 oid moves (copies). By construction, these copies are placed in the last M buckets.

For the next round, variable p will again start from bucket 0 and will extend to bucket 2M-1. When p reaches bucket 2M-1, there have been 2M new splits. These new splits imply that there must have been at most (4M-1)B+1 copies created from these additions. The copied oids from the first round are seen in the second round as regular oid additions. At most each such oid can be copied once more during the second round (the original oids from which these copies were created in the first round, cannot be copied again in the second round as they represent deleted records in their corresponding buckets). Hence the maximum number of copies after the second round is $((2M-1)B+1+(4M-1)B+1)$. The total number of copies C_{total} created in i rounds satisfies:

$$C_{total} \leq \{(2M-1)B+1\} + \{(2M-1)B+1+(4M-1)B+1\} + ...$$

where each $\{\}$ represents copies per round. Using that $M \geq 1$ and $B > 1$ we get:

$$C_{total} \leq 2MB + \{4MB + 2MB\} + ... \leq MB(\sum_{k=0}^{i}\sum_{j=0}^{k} 2^j) =$$

$$MB(2(2^{i+1}-1)-(i+1)) < 2^{i+2}MB$$

Lemma 6 implies that after the first round we have at least $(2M-1)Bg$ real oid additions. At the end of the second round the total number of oid additions is at least $(4M-1)Bg$. However $(2M-1)Bg$ of these were inserted and counted for the first round. Thus the second round introduces at least $2MBg$ new additions. Similarly, the third round introduces at least $4MBg$ new oid additions and so on. The total number of real oid additions A_{total} after all rounds, is lower bounded by:

$$A_{total} \geq (\{(2M-1)Bg\} + \{2MBg\} + \{4MBg\} + ...) \geq (MBg + 2MBg + 4MBg + ...)$$

Hence for i rounds:

$$A_{total} \geq MBg\sum_{k=0}^{i} 2^k = (2^{i+1}-1)MBg \geq 2^i MBg$$

From the above equations it can be derived that there exists a positive constant $const$ such that $C_{total}/A_{total} < const$. Since A_{total} is bounded by the total number of changes n, we have that

$C_{total} = O(n)$. For proving that partially persistent linear hashing has O(1) expected amortized updating per change, we note that when a real change occurs it is directed to the appropriate bucket where the structures of the Snapshot Index are updated in O(1) expected time. Rehashings must be carefully examined. This is because a rehashing of a bucket is caused by a single real oid addition (the one that created the split) but it results in a ”bunch” of copies made to a new bucket (at worse the whole current contents of the rehashed bucket are sent to the new bucket). However, using the space bound any sequence of n real changes can at most create $O(n)$ copies (extra work) or equivalently O(1) amortized effort per real change. ■

Optimization Issues

Optimizing the performance of partially persistent linear hashing involves the load factor l of the ephemeral Linear Hashing and the usefulness parameter u of the Snapshot Index. The load l lies between thresholds f and g. Note that l is an average over time of $l(t) = |S(t)|/B|b(t)|$, where $|S(t)|$ and $|b(t)|$ denote the size of the evolving set S and the number of buckets used at t. A good ephemeral linear hashing scheme will try to equally distribute the oids among buckets for each t. Hence on average the size (in oids) of each bucket b_j(t) will satisfy: $|b_j(t)| \sim |S(t)|/|b(t)|$.

One of the advantages of the Snapshot Index is the ability to tune its performance through usefulness parameter u. The index will distribute the oids of each b_j(t) among a number of useful pages. Since each useful page (except the acceptor page) contains at least uB alive oids, the oids in b_j(t) will be occupying at most $|b_j(t)|/uB$ pages, which is actually $l(t)/u$. Ideally, we would like the answer to a snapshot query to be contained in a single page. Then a good optimization choice is to maintain: $l/u < 1$. Conceptually, the load l gives a measure of the size of a bucket (”alive” oids) at each time. These alive oids are stored into the data pages of the Snapshot Index. Recall that an artificial copy happens if the number of alive oids in a data page falls below uB. At that point the remaining $uB - 1$ alive oids of this page are copied to a new page. By keeping l below u we expect that the alive oids of the split page will be copied in a single page which minimizes the number of I/O's needed for finding them.

On the other hand, the usefulness parameter u affects the space used by the Snapshot

Index and in return the overall space of the persistent hashing scheme. As mentioned in previous section, higher values of u imply frequent time splits, i.e., more page copies and thus more space. Hence it would be advantageous to keep u low but this implies an even lower l. In return, lower l would mean that the buckets of the ephemeral hashing are not fully utilized. This is because low l causes set S(t) to be distributed into more buckets not all of which may be fully occupied.

At first this requirement seems contradictory. However, for the purposes of partially persistent linear hashing, having low l is still acceptable. Recall that the low l applies to the ephemeral hashing scheme whose history the partially persistent hashing observes and accumulates. Even though at single time instants the $b_j(t)$'s may not be fully utilized, over the whole time evolution many object oids are mapped to the same bucket. What counts for the partially persistent scheme is the total number of changes accumulated per bucket. Because of bucket reuse, a bucket will gather many changes creating a large history for the bucket and thus justifying its use in the partially persistent scheme. Our findings regarding optimization will be verified through the experimentation results that appear in the next section.

4.3.2 The Evolving-List Approach

The elements of bucket $b_j(t)$ can also be viewed as an evolving list $lb_j(t)$ of alive oids. Such an observation is consistent with the way buckets are searched in ephemeral hashing, i.e., linearly, as if a bucket's contents belong to a list. Accessing the bucket state $b_j(t)$ is then reduced to reconstructing $lb_j(t)$. Equivalently, the evolving list of oids should be made partially persistent.

We note that [VV97] presents a notion of persistent lists called the C-lists. A C-list is a list structure made up of a collection of pages that contain temporal versions of data records clustered by oid. However, a C-list and a partially-persistent list solve different problems. (1) A C-list addresses the query: "given an oid and a time interval, find all versions of this oid during this interval". Instead, a partially persistent list finds all the oids that its ephemeral list had at a given time instant. (2) In a C-list the various temporal versions of an oid must be clustered together. This implies that oids are also ordered by key. In contrast, a persistent list does not require ordering its keys, simply because its ephemeral list is not ordered. (3) In a C-list updates and queries are

performed by first using the MVAS structure. Updating/querying the partially persistent list always starts from the top of the list and proceeds through a linear search.

For the evolving-list approach, a list page has now two areas. The first area is used for storing oid records and its size is B_r (B_r is $O(B)$). The second area, of size $B - B_r$ ($B - B_r$ is also O(B)), accommodates an extra structure (array NT) which will be explained shortly. When the first oid k is added on bucket b_j at time t, a record $< k, [t, now) >$ is appended in the first list page. Additional oid insertions will create record insertions in this page. When the B_r part of the first page gets full of records, a new page is added after this page in the list and so on.

Good clustering is achieved if each page in $lb_j(t)$ is a useful page. A page is useful if : (i) it is full of records and contains at least uB_r alive records ($0 < u \leq 1$), or, (ii) it is the last list page. A non-useful page is taken off list $lb_j(t)$ and its alive records are copied to the current last list page. If the last page in $lb_j(t)$ does not have enough free space, a new last page is appended.

Reconstructing $b_j(t)$ is equivalent to finding the pages in $lb_j(t)$. We will use two array structures. Array $FT_j(t)$ provides access to the first page in the list for any time t. Entries in FT_j have the form $< time, pid >$ where pid is the address of the first page. If the first page of the list changes at t, a new entry is appended in FT_j. This array can be implemented as a multilevel index since entries are added in increasing time order. Each remaining page of $lb_j(t)$ is found by the second structure, which is an array of size $B - B_r$, implemented inside every list page. Let $NT(A)$ be the array inside page A. This array is maintained for as long as the page is useful. Entries in $NT(A)$ are also of the form $< time, pid >$, where pid corresponds to the address of the next list page after page A.

To answer a temporal membership query for oid k at time t the appropriate bucket b_j, where oid k would have been mapped by the hashing scheme at t, is found. This part is the same in evolving-set approach. To reconstruct bucket b_j(t), the first page in $lb_j(t)$ is found by searching t in array FT_j. This search is $O(log_B(n_j/B))$. The remaining pages of $lb_j(t)$ are found by locating t in the NT array of each subsequent list page. Since all pages in the list $lb_j(t)$ are useful, the oids in $b_j(t)$ are found in $O(|b_j(t)|/B)$ I/O's. Hence, the space used by the evolving-list approach is $O(n_j/B)$ while updating is $O(|b_j(t)/B)$ per update.

There are two differences between the evolving-list and the evolving-set approaches. First, updating using the Snapshot Index remains constant, while in the evolving list the whole current list may have to be searched for deleting an oid. Second, reconstruction in the evolving-list starts from the top of the list pages while in the evolving-set reconstruction starts from the last page of the bucket. This may affect the search for a given oid depending whether it has been placed near the top or near the end of the bucket.

4.4 Performance Evaluation

For the Partially Persistent Linear Hashing we implemented the set-evolution (PPLH-s) and the list-evolution (PPLH-l) approaches, using controlled splits. Both are compared against Linear Hashing (in particular Atemporal linear hashing, which is discussed below), the MVBT and two R*-tree implementations (R_i which stores intervals in a 2-dimensional space, and R_p which stores points in a 3-dimensional space). The partial persistence methodology is applicable to other hashing schemes as well. We implemented Partially Persistent Extendible Hashing using the set-evolution (PPEH-s) and uncontrolled splits and compared it with PPLH-s.

4.4.1 Experimental Setup

We set the size of a page to hold 25 oid records (B=25). An oid record has the following form, $< oid, start - time, end - time, ptr >$, where the first field is the oid, the second is the starting time and the third the ending time of this oid's lifespan. The last field is a pointer to the actual object (which may have additional attributes).

We first discuss the Atemporal linear hashing (ALH). It should be clarified that ALH is not the ephemeral linear hashing whose evolution the partially persistent linear hashing observes and stores. Rather, it is a conventional linear hashing scheme that treats time as just another attribute. This scheme simply maps objects to buckets using the object oids. Consequently, it "sees" the different lifespans of the same oid as copies of the same oid. We implemented ALH using the scheme originally proposed by Litwin in [Lit80]. For split functions we used the hashing by division

functions $h_i(oid) = oid \quad mod \quad 2^i M$ with $M = 100$. For good space utilization controlled splits were employed. The lower and upper thresholds (namely f and g) had values 0.7 and 0.9 respectively.

Another approach for Atemporal hashing would be a scheme which uses a combination of oid and the start-time or end-time attributes. However this approach would still have the same problems as ALH for temporal membership queries. For example, hashing on start-time does not help for queries about time instants other than the start-times.

The two Partially Persistent Linear Hashing approaches (PPLH-s and PPLH-l) observe an ephemeral linear hashing LH(t) with controlled splits and load between f=0.1 and g=0.2. Array H which identifies the hashing scheme used at each time is kept in main-memory (i.e., no I/O cost is counted for accessing H). In our experiments the size of H varied from 5 to 10 KB (which can easily fit in today's main memories). Unless otherwise noted, PPLH-s was implemented with $u = 0.3$ (other values for usefulness parameter u were also examined). Since the entries in the time-tree associated with a bucket have half the oid record size, each time-tree page can hold up to 50 entries.

In the PPLH-l implementation, the space for the oid records B_r can hold 20 such records. A page in the list is considered useful as long as the number of alive oids in the page is greater than or equal to 5 (i.e., u = 0.25). The remaining space in a list page (of size 5 oid records) is used for the page's NT array. Similarly with the time-arrays, NT arrays have entries of half size, i.e., each page can hold 10 NT entries. For the same reason, the pages of each FT_j array can hold up to 50 entries.

The Multiversion B-tree (MVBT) implementation uses buffering during updates; this buffer stores the pages in the path to the last update (LRU buffer replacement policy is used). Such buffering can be very advantageous since updates are directed towards the most current B-tree, which is a small part of the whole MVBT structure. In our experiments we set the buffer size to 10 pages. The original MVBT uses buffering for queries, too. For a fair comparison with the partially persistent methods, during queries we allow the MVBT to use a buffer that is as large in size as array H for that experiment. Note that during querying the MVBT uses a root* structure. For every

time t, root* identifies the root of the B-tree at that time (i.e., where the search for the query should start from). Even though root* can increase with time, it is small enough to fit in the above main-memory buffer. Thus we do not count I/O accesses for searching root*.

As with the Snapshot Index, a page in the MVBT is "alive" as long as it has at least q alive records. If the number of alive records falls below q this page is merged with a sibling (this is called a weak version underflow in [BGO+96]). On the other extreme, if a page has already B records (alive or not) and a new record is added, the page splits. Both conditions need special handling. First, a time-split happens (which is like the copying procedure of the Snapshot Index). All alive records in the split page are copied to a new page. Then the resulting new page should be incorporated in the structure. The MVBT requires that the number of alive records in the new page should be between q+e and B-e where e is a predetermined constant. Constant e works as a threshold that guarantees that the new page can be split or merged only after at least e new changes. Not all values for q, e and B are possible as they must satisfy some constraints; for details we refer to [BGO+96]. In our implementation we set q = 5 and e = 4. The directory pages of the MVBT have the same format as the data pages.

The R_i implementation assigns to each oid its lifespan interval. One dimension is used for the oid and one for the interval. When a new oid k is added in set S at time t, a record $< k, [t, now), ptr >$ is added in an R*-tree data page. Note that now is stored in the R*-tree as some large number (larger than the maximum evolution time). Directory pages in the R*-tree include one more attribute per record for representing an oid range. The R_p implementation has similar format for data pages, but it assigns separate dimensions for the start-time and the end-time of the object's lifespan interval. Hence a directory page record has seven attributes (two for each of the oid, start-time, end-time and one for the pointer). During updating, both R*-tree implementations used a buffer (10 pages) to keep the pages in the path leading to the last update. A buffer as large as the size of array H was again used during the query phase.To further optimize the R*-tree approaches towards temporal membership queries we forced the trees to cluster data first based on the oids and then on the time attributes.

Another popular external dynamic hashing scheme is Extendible Hashing [FNP+79],

which differs from linear hashing in two ways. First, a directory structure is used to access the buckets. Second, an overflow is addressed by splitting the bucket that overflowed. Note that traditional extendible hashing uses uncontrolled splitting. To implement Partially Persistent Extendible Hashing one needs to keep the directory history as well as the history of each bucket. The history of each bucket was kept using the evolving-set approach (PPEH-s). In our experiments the directory history occupied between 7 and 18KB so we kept it in main memory. Since uncontrolled splitting is used, most buckets will be full of records before being split. This implies that the alive records in a bucket are close to one page. Consequently, to get better query performance (with the expense of higher space overhead), the history of such buckets was implemented using Snapshot indices with higher utilization parameter (u=0.5).

Various workloads were used for the comparisons. Each workload contains an evolution of a dataset S and temporal membership queries on this evolution. More specifically, a workload is defined by triplet $W = (U, E, Q)$, where U is the universe of the oids (the set of unique oids that appeared in the evolution of set S), E is the evolution of set S and $Q = \{Q_1, ..., Q_r\}$ is a collection of queries, where $r = |U|$ and Q_k is the set of queries corresponds to oid k.

Each evolution starts at time 1 and finishes at time $MAXTIME$. Changes in a given evolution were first generated per object oid and then merged. First, for each object with oid k, the number n_k of the different lifespans for this object in this evolution was chosen. The choice of n_k was made using a specific random distribution function (namely $Uniform$, $Exponential$, $Step$ or $Normal$) whose details are described in the next section. The start-times of the lifespans of oid k were generated by randomly picking n_k different starting points in the set $\{1, ..., MAXTIME\}$. The end-time of each lifespan was chosen uniformly between the start-time of this lifespan and the start-time of the next lifespan of oid k (since the lifespans of each oid k are disjoint). Finally the whole evolution E for set S was created by merging the evolutions for every object.

For another "mix" of lifespans, we also created an evolution that picks the start-times and the length of the lifespans using Poisson distributions; we called it the $Poisson$ evolution.

A temporal membership query in query set Q is specified by tuple (oid, t). The number of queries Q_k for every object with oid k was chosen randomly between 10 and 20; thus on average,

$Q_k \sim 15$. To form the (k, t) query tuples the corresponding time instants t were selected using a uniform distribution from the set $\{1, ..., MAXTIME\}$. The $MAXTIME$ is set to 50000 for all workloads. The value used in the R-trees for now was 100000.

Each workload is described by the distribution used to generate the object lifespans, the number of different oids, the total number of changes in the evolution n (object additions and deletions), the total number of object additions NB, and the total number of queries.

4.4.2 Experiments

First, the behavior of all implementations was tested using a basic $Uniform$ workload. The number of lifespans per object follows a uniform distribution between 20 and 40. The total number of distinct oids was $|U| = 8000$, the number of real changes n = 466854 and NB = 237606 object additions. Hence the average number of lifespans per oid was $\bar{NB} \sim 30$ (we refer to this workload as Uniform-30). The number of queries was 115878.

Figure 4.3.a presents the average number of pages accessed per query by all methods. The PPLH methods have the best performance, about two pages per query. The ALH approach uses more query I/O (about 1.5 times in this example) because of the larger buckets it creates. The MVBT also uses more I/O (about 1.75 times) than the PPLH approaches since a tree path is traversed per query. The R_i uses more I/O's per query than the MVBT, mainly due to tree node overlapping and larger tree height (the R_i height relates to the total number of oid lifespans while in the MVBT the height corresponds to the number of alive oids at the time specified by the query). The R_p tree has the worse query performance; it used an average of 28.3 I/O's per query in this experiment (its performance has been truncated in Figure 4.3.a to fit the graph). While using a separate dimension for the two endpoints of a lifespan interval allows for better clustering (see also the space usage in Figure 4.3.c) it makes it more difficult to check whether an interval contains a query time instant.

Figure 4.3.b shows the average number of I/O's per update. The best update performance was given by the PPLH-s method. In PPLH-l the NT array implementation inside each page limits the actual page area assigned for storing oids and thus increases the number of pages used per bucket. The MVBT update is longer than PPLH-s since the MVBT traverses a tree for each update

(instead of quickly finding the location of the updated element through hashing). The update of R_i follows; it is larger that the MVBT since the size of the tree traversed is related to all oid lifespans (while the size of the MVBT structure traversed is related to the number of alive oids at the time of the update). The R_p tree uses larger update processing than the R_i because of the overhead to store an interval as two points. The ALH had the worse update processing. This is because in ALH all lifespans with the same oid are thrown on the same bucket thus creating large buckets that must be searched serially during an update.

The space consumed by each method appears in Figure 4.3.c. The ALH approach uses the smallest space since it stores a single record per oid lifespan and uses "controlled" splits with high utilization. The PPLH-s method has also very good space utilization, very close to ALH. The R-tree methods follow; R_p uses slightly less space than the R_i because paginating intervals (putting them into bounding rectangles) is more demanding than with points. Note that similarly to ALH, both R* methods use a single record per oid lifespan; the additional space is mainly because the average R-tree page utilization is about 65NT array implementation reduces page utilization. The MVBT has the largest space requirements, about twice more space than the ALH and PPLH-s methods.

In summary, the PPLH-s has the best overall performance. Similarly with the comparison between ephemeral hashing and B-trees, the MVBT behaves worse than temporal hashing (PPLH-s) for temporal membership queries. The ALH is slightly better than PPLH-s only in space requirements, even though not significantly. The R-tree based methods are much worse than PPLH-s in query and update performance.

To consider the effect of lifespan distribution all approaches were compared using five additional workloads (called the exponential, step, normal, poisson and uniform-consecutive). These workloads had the same number of distinct oids ($|U| = 8000$), number of queries (115878) and similar n ($\sim 0.5M$) and $\bar{NB}$ (~ 30) parameters. The Exponential workload generated the n_k lifespans per oid using an exponential distribution with probability density function $f(x) = \beta e^{-\beta x}$ and mean $1/\beta = 30$. The total number of changes was n = 487774, the total number of object additions was NB = 245562 and $\bar{NB} = 30.7$. In the Step workload the number of lifespans per oid follows a step function. The first 500 oids have 4 lifespans, the next 500 have 8 lifespans and so on, i.e., for

every 500 oids the number of lifespans advances by 4. In this workload we had n = 540425, NB = 272064 and $\bar{NB} = 34$. The Normal workload used a normal distribution with $\mu = 30$ and $\sigma^2 = 25$. Here the parameters were: n = 470485, NB = 237043 and $\bar{NB} = 29.6$.

For the *Poisson* workload the first lifespan for every oid was generated randomly between time instants 1 and 500. The length of a lifespan was generated using a Poisson distribution with mean 1100. Each next start time for a given oid was also generated by a Poisson distribution with mean value 500. For this workload we had n = 498914, NB = 251404 and $\bar{NB} = 31$. The main characteristic of the Poisson workload is that the number of alive oids over time can vary from a very small number to a large proportion of $|U|$, i.e., there are time instants where the number of alive oids is some hundreds and other time instants where almost all distinct oids are alive.

The special characteristic of the *Uniform_consecutive* workload is that it contains objects with multiple but consecutive lifespans. This scenario occurs when objects are updated frequently during their lifetime. Each update is seen as the deletion of the object followed by the insertion of the updated object at the same time. Since the object retains its oid through updates, this process creates consecutive lifespans for the same object (the end of one lifespan is the start of the next lifespan). This workload was based on the Uniform-30 workload and had n = 468715, NB = 236155 and $\bar{NB} = 30$. An object has a single lifetime which is cut into consecutive lifespans.The start-times of an object's lifespans are chosen uniformly.

Figure 4.4 presents the query, update and space performance under the new workloads. For simplicity only the R_i method is presented among the R-tree approaches (as with the uniform load, the R_p used consistently more query and update than R_i and similar space). The results resemble the Uniform-30 workload. As before, the PPLH-s approach has the best overall performance using slightly more space than the "minimal" space of ALH. PPLH-l has the same query performance with PPLH-s but uses more updating and space. Note that in Figure 4.4.a, the query performance of R_i has been truncated to fit the graph (on average, R_i used about 9.2, 10.2, 9.9, 10.2 and 27.5 I/O's per query in the exponential, step, normal, poisson and consecutive workloads respectively). Likewise, in Figure 4.4.c the space of the MVBT is truncated (MVBT used about 26K, 28K, 25K, 35K and 28K pages for the respective workloads). We have tried the consecutive work-

loads for the exponential, step, normal and poisson distributions, too. The comparative behavior of all methods in the consecutive workloads was similar and is thus not presented.

The effect of the number of lifespans per oid was tested using eight uniform workloads with varying average number of lifespans. All used $|U| = 8000$ different oids and the same number of queries ($\sim 115K$). The other parameters are shown in Table 4.1 below.

Table 4.1: Uniform Datasets.

workload	n	NB	$\bar{NB}$
uniform-10	149801	75601	9.4
uniform-20	308091	155354	19.4
uniform-30	466854	237606	29.7
uniform-40	628173	316275	39.5
uniform-50	787461	396266	49.5
uniform-80	1264797	635604	79.5
uniform-100	1585949	796451	99.5

The results appear in Figure 4.5. The query performance of atemporal hashing deteriorates as NB increases since buckets become larger (Figure 4.5.a). The PPLH-s, PPLH-l and MVBT methods have a query performance that is independent of NB (this is because in all three methods the NB lifespans of a given oid appear at different time instants and thus do not interfere with each other). The query performance of R_i was much higher and it is truncated from the figure. Interestingly, the R_i query performance decreases gradually as $\bar{NB}$ increases (from 11.7 I/O's to 8.9 I/O's). This is because R_i clustering improves as $\bar{NB}$ increases (there are more records with the same key).

PPLH-s outperforms all methods in update performance (Figure 4.5). As with querying, the updating of PPLH-s, PPLH-l and MVBT is basically independent of $\bar{NB}$. Because of better clustering with increased $\bar{NB}$, the updating of R_i gradually decreases. In contrast, because increased $\bar{NB}$ implies larger bucket sizes, the updating of ALH increases. The space of all methods increases with $\bar{NB}$ as there are more changes n per evolution (Table 4.1). The ALH has the lower space, followed by the PPLH-s; the MVBT has the steeper space increase (for $\bar{NB}$ values 80 and 100, MVBT used 68K and 84.5K pages).

The effect of the number of distinct oids per evolution was examined by considering four uniform workloads. The number of distinct oids $|U|$ was: 5000, 8000, 12000 and 16000, respectively. All workloads had similar average number of lifespans per distinct oid ($\bar{NB} \sim 30$). The other parameters appear in Table 4.2. The results are depicted in figure 4.6. The query performance of the hashing methods (PPLH-s, PPLH-l and ALH) is basically independent of $|U|$, with PPLH-s and PPLH-l having the lowest query I/O. In contrast, it increases for both MVBT and R_i (the R_i used 9.1, 10.7, 11.1 and 12.4 I/O's per query). The increase is because more oids are stored in these tree structures, thus increasing the structure's height. This is more evident in R_i as all oids appear in the same tree. Similar observations hold for the update performance (i.e., the hashing based methods have updating that is basically independent of $|U|$, while the updating of tree based methods tends to increase with $|U|$). Finally, the space of all methods increases because n increases (Table 4.2).

Table 4.2: Datasets with different number of oids.

workload	n	NB	#of queries
uniform-5K	291404	146835	72417
uniform-8K	466854	237606	115878
uniform-12K	700766	353067	174167
uniform-16K	937443	472294	226456

From the above experiments, the PPLH-s method has the most competitive performance among all solutions. As mentioned previously, the PPLH-s performance can be further optimized through the setting of usefulness parameter u. Figure 4.7 shows the results for the basic Uniform-30 workload ($|U| = 8000$, n = 466854, NB = 237606 and $\bar{NB} \sim 30$) but with different values of u. As expected, the best query performance occurs if u is greater than the maximum load of the observed ephemeral hashing. For these experiments the maximum load was 0.2. As asserted in Figure 4.7.a, the query time is minimized after $u = 0.3$. The update is similarly minimized (Figure 4.7.b) for u's above 0.2, since after that point, the alive oids are compactly kept into few pages that can be updated easier (for smaller u's the alive oids can be distributed into more pages which increases the update process). Figure 4.7.c shows the space of PPLH-s. For u's below the maximum load the alive oids are distributed among more data pages, hence when such a page becomes non-useful it contains

Data Sets	I/Os per Query		I/Os per Update)		Space (Pages)	
	PPLH-s	*PPEH*	*PPLH-s*	*PPEH*	*PPLH-s*	*PPEH*
Uniform-30	2	3.22	2.18	2.53	13256	17035
Exponential-30	2	2.8	2.18	2.43	13646	15985
Step-30	2	2.8	2.18	2.43	15003	16688
Normal-30	2	3.18	2.28	2.52	13462	16982
Poisson-30	2	2	2.18	2.23	14394	16760
Consecutive-30	2	2.41	2.23	2.60	14246	17636

Table 4.3: Performance comparison of PPLH-s (controlled splits) versus PPEH-s (uncontrolled splits).

less alive oids and thus less copies are made, resulting in smaller space consumption. Using this optimization, the space of PPLH-s can be made similar to that of the ALH at the expense of some increase in query/update performance.

Finally, we compared the PPLH-s with partially persistent extendible hashing, a method that also uses the evolving-set approach but uncontrolled splits (PPEH-s) (Table 4.3). With uncontrolled splits a bucket b_j is split only when it overflows. Thus the "observed" ephemeral hashing tends to use less number of buckets but longer in size. As it turns out, this policy has serious consequences. First, for the Snapshot Index that keeps the history of bucket b_j the state b_j(t) is a full page. This means that the Snapshot Index will distribute b_j(t) into more physical pages in the bucket's history. Consequently, to reconstruct b_j(t) more pages will be accessed. Second, the Snapshot Index will be copying larger alive states and thus occupy more space. We run the same set of experiments for the PPEH-s. In summary, PPEH-s used consistently larger query, update and space than PPLH-s.

4.4.3 Summary

This chapter addressed the problem of Temporal Hashing, or equivalently, how to support temporal membership queries over a time-evolving set S. An efficient solution termed partially persistent hashing was presented. For queries and updates, this scheme behaves as if a separate, ephemeral dynamic hashing scheme is available on every state assumed by set S over time. However the method uses linear space. By hashing oids to various buckets over time, partially persistent hashing reduces the temporal hashing problem into reconstructing previous bucket states. Two fla-

vors of partially persistent linear hashing were presented, one based on an evolving-set abstraction (PPLH-s) and one on an evolving-list (PPLH-l). They have similar query and comparable space performance but PPLH-s uses much less updating. Both methods were compared against straightforward approaches, namely: traditional (atemporal) linear hashing scheme, the Multiversion B-Tree and two R*-tree implementations. The experiments showed that PPLH-s has the most robust performance. Partially persistent hashing should be seen as an extension of traditional external dynamic hashing in a temporal environment. The methodology is general and can be applied to other ephemeral dynamic hashing schemes, like extendible hashing, etc.

evolution of Set S up to time t= 25:

t	oid	oper
1	10	+
2	7	+
4	3	+
8	21	+
9	15	+
15	36	+
16	29	+
17	13	+
20	12	+
21	8	+
25	10	-

evolution of bucket 1:

t	oid	oper
1	10	+
9	15	+
21	15	-
25	10	-

records in bucket 1's history:

at t = 20	at t = 21	at t = 25
<oid, lifespan>	<oid, lifespan>	<oid, lifespan>
<10, [1, now)>	<10, [1, now)>	<10, [1, 25)>
<15, [9, now)>	<15, [9, 21)>	<15, [9, 21)>

evolution of bucket 3:

t	oid	oper
4	3	+
17	13	+
21	8	+

records in bucket 3's history

at t = 20	at t = 21	at t = 25
<oid, lifespan>	<oid, lifespan>	<oid, lifespan>
<3, [4, now)>	<3, [4, now)>	<3, [4, now)>
<13, [17, now)>	<13, [17, now)>	<13, [17, now)>
	<8, [21, now)>	<8, [21, now)>

evolution of bucket 5:

t	oid	oper
21	15	+

records in bucket 5's history

at t = 20	at t = 21	at t = 25
<oid, lifespan>	<oid, lifespan>	<oid, lifespan>
	<15, [21, now)>	<15, [21, now)>

Figure 4.2: The detailed evolution for set S until time t = 25 (a "+/-" denotes addition/deletion respectively). Changes assigned to the histories of three buckets are shown. The hashing scheme of Figure 4.1 is assumed. Addition of oid 8 in S at t = 21, causes the first split. Moving oid 15 from bucket 0 to bucket 5 is seen as a deletion and an addition respectively. The records stored in each bucket's history are also shown. For example, at t=25, oid 10 is deleted from set S. This updates the lifespan of this oid's corresponding record in bucket 0's history from $< 10, [1, now) >$ to $< 10, [1, 25) >$.

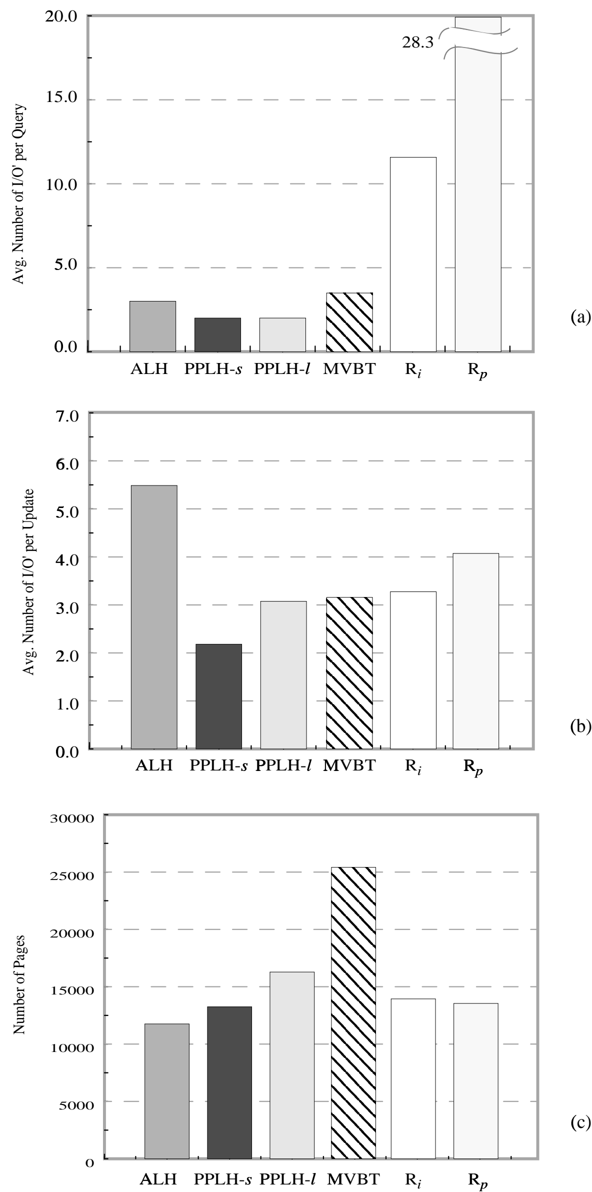


Figure 4.3: (a) Query, (b) Update, and, (c) Space performance for all implementations on a uniform workload with 8K oids, $n \sim 0.5M$ and $\bar{NB} \sim 30$.

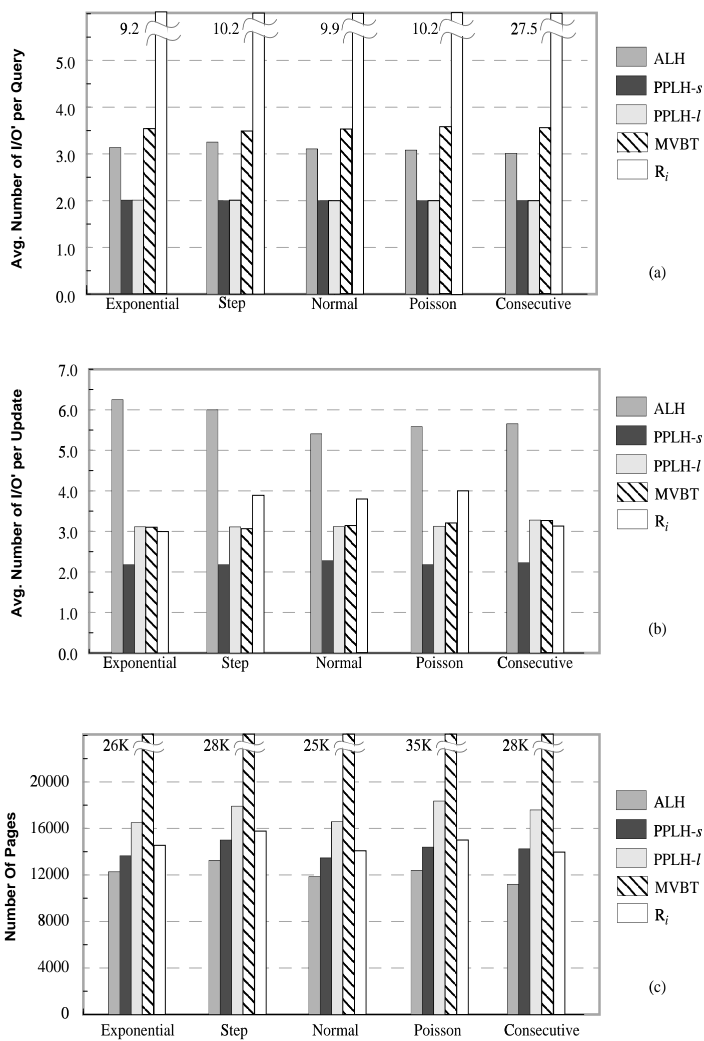


Figure 4.4: (a) Query, (b) Update, and, (c) Space performance for ALH, PPLH-s, PPLH-l, MVBT and R_i methods using the exponential, step, normal and poisson workloads with 8K oids, $n \sim 0.5M$ and $\bar{N}B \sim 30$.

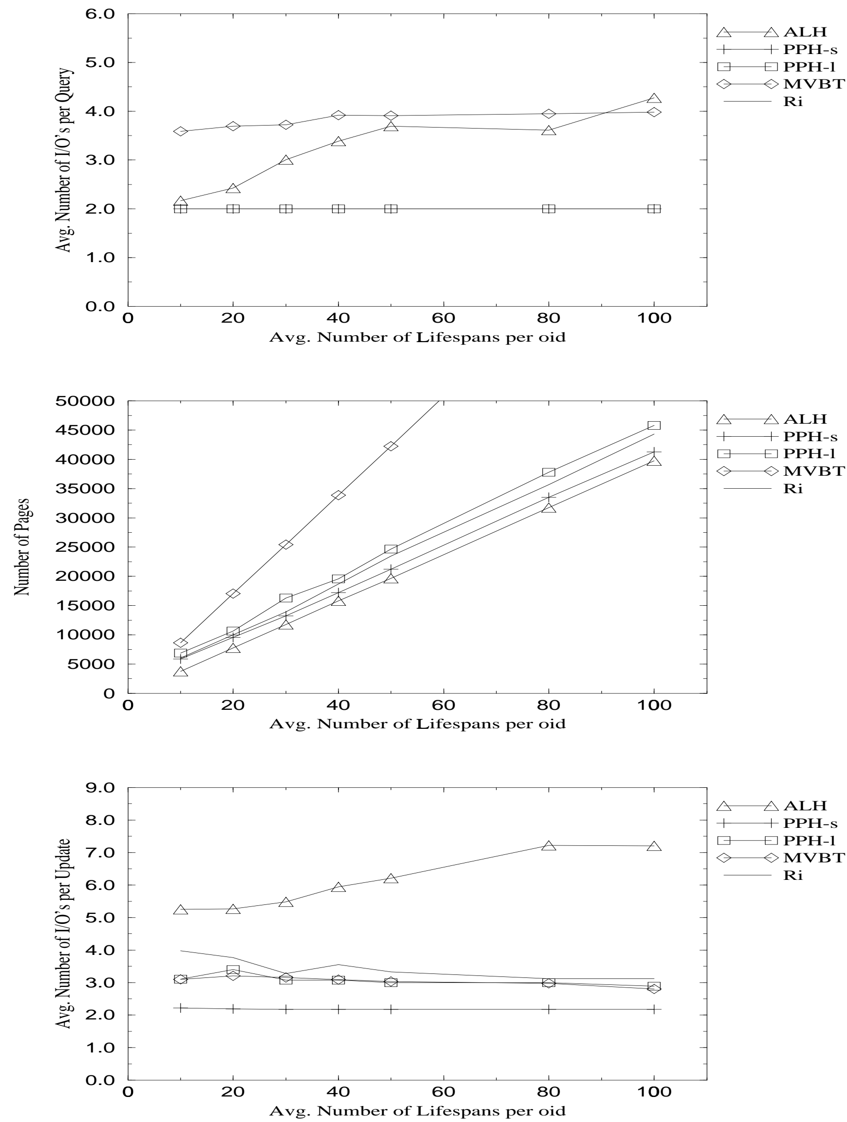


Figure 4.5: (a) Query, (b) Update, and, (c) Space performance for ALH, PPLH-s, PPLH-l, MVBT and Ri methods using various uniform workloads with varying $\bar{NB}$.

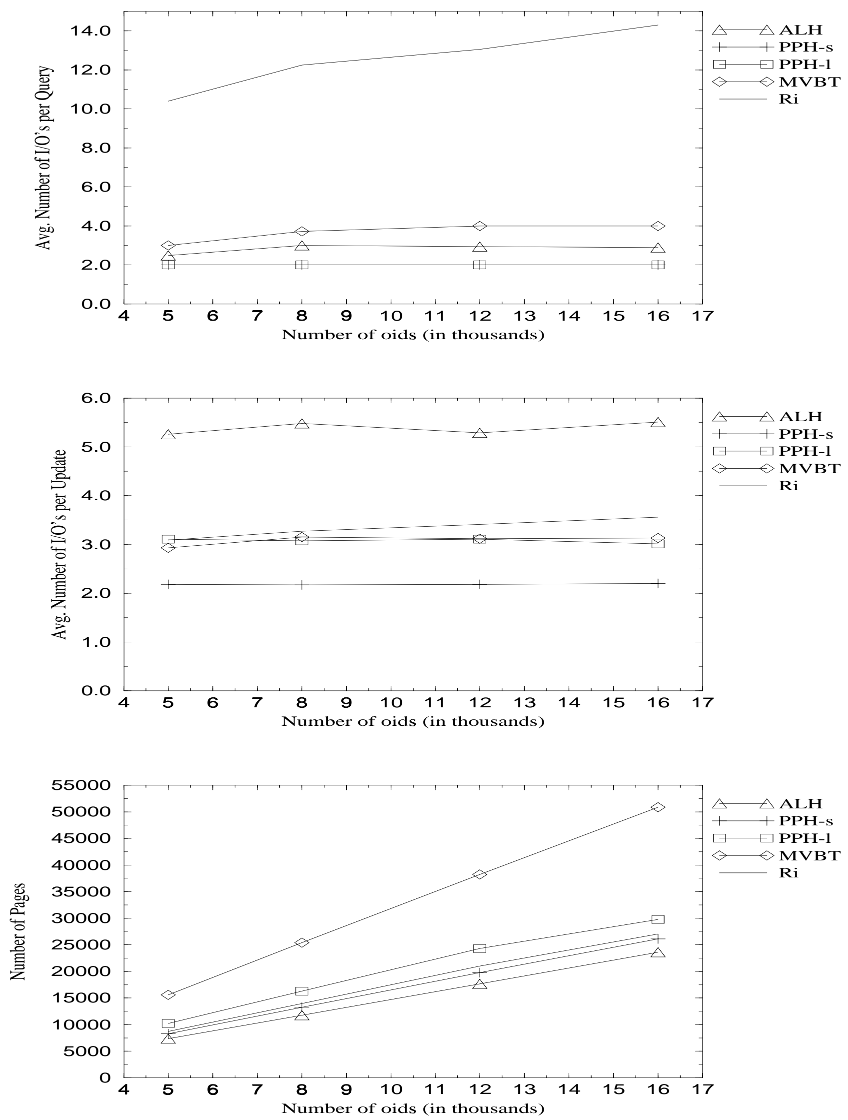


Figure 4.6: (a) Query, (b) Update, and, (c) Space performance for ALH, PPLH-s, PPLH-l, MVBT and Ri methods using various uniform workloads with varying $|U|$.

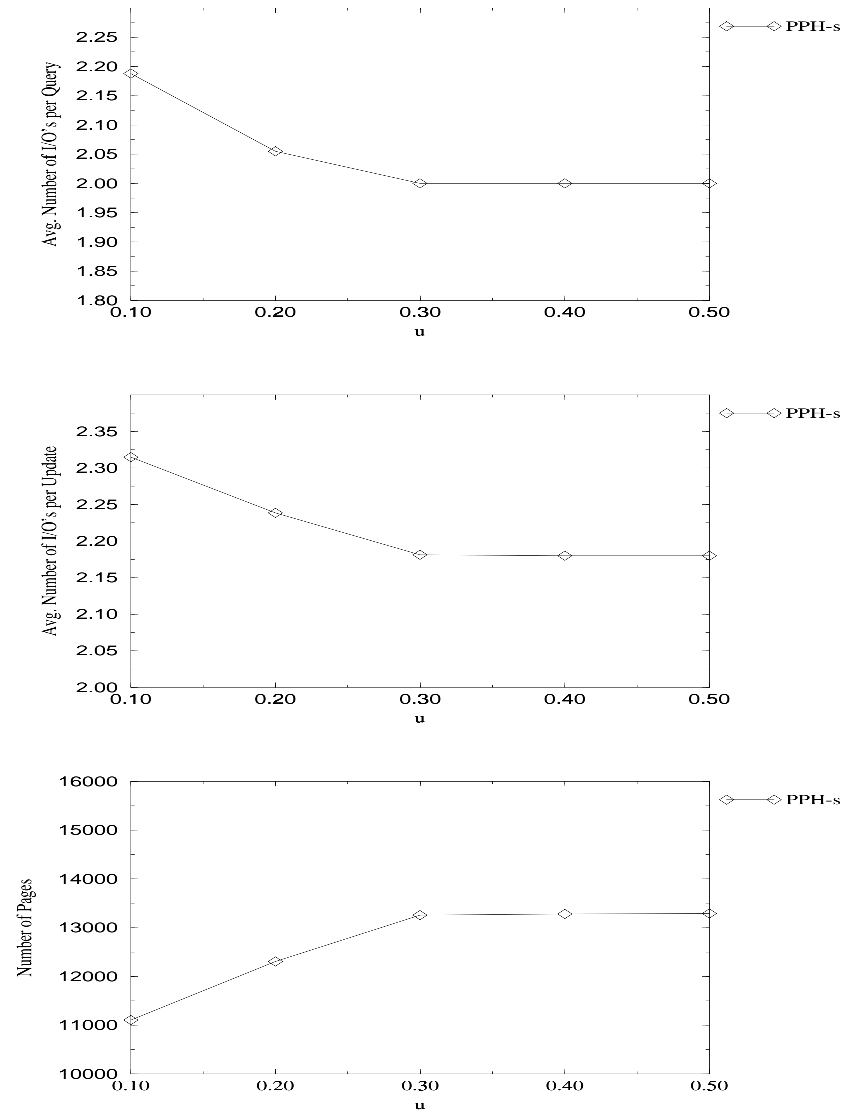


Figure 4.7: (a) Query, (b) Update, and, (c) Space performance for PPLH-s on a uniform workload with varying values of the usefulness parameter u.

Chapter 5

Summary and Future Research

The primary focus of this thesis has been to design efficient external memory access methods for spatiotemporal databases. In particular, this work presents solutions to the following problems:

1. Indexing Historical Spatiotemporal Databases: We study the problem of indexing spatiotemporal objects in order to answer efficiently proximity and topological queries about the past and current positions and extents of the indexed objects. For example, a user can define a region in space (usually a rectangle) and a specific time interval in the past, and the database must return the objects that intersected the query region during the specified time interval. The most important issue in indexing such data, is how to represent an object that moved from an initial position to a final position. In traditional spatial indexing, objects are approximated by their Minimum Bounding Rectangle (MBR) and an index is constructed over these MBRs. However, using the MBR of a moving object will introduce a lot of empty space, which greatly decreases the performance of any index that uses these MBRs. To reduce empty space, we proposed to introduce a limited number of artificial updates. An artificial update breaks the lifetime of an object into two contiguous but not overlapping intervals, reducing empty space and providing better clustering. In order to maintain the index storage space linear to the number of real updates N, the number of artificial updates was limited to be a

fraction of N. The problem now is to decide which objects and where should be artificially updated. We formulated this problem as an optimization problem and provided an optimal and efficient solution for the case that objects move or change extent in a linear fashion over time. To index the objects generated after the artificial updates, we used a partially persistent version of a very successful spatial index (R-tree). An extensive experimental study demonstrated the advantages of the proposed methods over other straightforward solutions.

2. Indexing the Future Positions of Continuously Moving Objects: Here, we consider a database that tracks objects moving continuously in one and two dimensions (cars moving in a highway system or mobile phone users moving in a 2-dimensional space). Storing the position of each object explicitly implies that the database has to be frequently updated, since moving objects change positions continuously. Instead, a better approach is to store in the database the function that describes the movement of each object. Now changes are considered only at the instants where the function parameters that describe the movements of a particular object change (speed, direction, etc). While this approach minimizes the update overhead, it introduces the novel problem of indexing the object moving functions. We designed methods to index these functions and answer proximity queries about the *future* locations of the indexed objects, assuming that they are moving in a linear fashion over time. For the one dimensional case, we proposed (i) a dynamic, external memory data structure with guaranteed worst case performance and linear space and (ii) a practical approximation algorithm with expected logarithmic query time. Also, we presented extensions of our techniques in two dimensions.

3. Temporal Hashing in Spatiotemporal Databases: We examined the problem of temporal hashing, that is, answering a membership query on a set that evolves over time. In traditional (non-temporal) databases this query can be answered efficiently by using traditional hashing schemes. We proposed to make a hashing method partially persistent, and we provided two approaches to achieve that. Experiments showed that partially persistent hashing outperforms other possible methods for indexing an evolving set for the membership query.

There are a number of interesting open problems that are related to the work presented in this thesis. In Historical Spatiotemporal Databases, a possible future research direction is to

consider how to index the data On-Line. In that case, objects need to be indexed at the time they arrive in the system and no knowledge about their future state exists. Another interesting extension of the work presented here is to use the proposed index methods in order to efficiently implement join between two spatiotemporal relations. Consider two spatiotemporal relations $R1$ and $R2$. Then a join query is the following: " find the pairs of objects whose extents intersected during the time interval T". To answer this join query we can use the tree index structures over the relations $R1$ and $R2$ and perform a synchronized tree traversal to compute the result.

For indexing continuously moving objects, it is interesting to consider parallel and main memory techniques to speed up the execution of range and nearest neighbor queries. Also, another interesting extension to work presented here is to consider objects that move with more complex functions (for example quadratic). A generalization of the 1.5 dimensional problem is when the terrain is subdivided into areas with various speed limits or terrain abnormalities that limit movement according to direction.

Bibliography

[AVIS96] S. Adali, K. Seljuk 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.

[AAE+98] P.K. Agarwal, L. Arge, J. Erickson, P. Franciosa and J.S. Vitter. Efficient Searching with Linear Constraints In *Proc. 17th ACM PODS Symposium on Principles of Database Systems*,pp. 169-178 1998.

[AAE00] P.K. Agarwal, L. Arge and J. Erickson. Indexing Moving Points In *Proc. 19th ACM PODS Symposium on Principles of Database Systems*, May 2000.

[AE98] P.K. Agarwal, and J. Erickson. Geometric range searching and its relatives. In *Discrete and Computational Geometry: Ten Years Latter.*, (B. Chazelle, E. Goodman, and R. Pollack eds.), American Math. Society, Providence, 1998.

[AV88] A. Aggarwal and J. S. Vitter. The Input/Output complexity of sorting and related problems. In *Communications of the ACM*, 31(9):1116-1127, 1988.

[AS] I. Ahn, and R.T. Snodgrass. Performance evaluation of a temporal database management system. In *Proc. ACM SIGMOD Conf.*, pp.96-107, 1986.

[ARC98] ArcView GIS. ArcView Tracking Analyst. 1998.

[Arg95] 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.

[Arg97] L. Arge. External-Memory Algorithms with Applications in Geographic Information Systems. In *Algorithmic Foundations of Geographic Information Systems,* LNCS 1340, 1997.

[AV96] L. Arge and J. S. Vitter. Optimal Dynamic Interval Management in External Memory. In *Proc. 37th Annual Symp. on Foundations of Comp. Science*, pp. 560-569, 1996.

[BGH97] J. Basch, L. Guibas and J. Hershberger. Data Structures for Mobile Data. In Proceedings of the Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, New Orleans, Louisiana, 1997.

[BGO+96] B. Becker, S. Gschwind, T. Ohler, B. Seeger and P. Widmayer. An Asymptotically Optimal Multiversion B-Tree. *VLDB Journal*, 5(4):264-275,1996.

[BKS+90] 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.

[Ben77] J.L. Bentley. Algorithms for Klee's Rectangle Problems. *Technical Report*, Computer Science Department, Carnegie-Mellon University, Pittsburgh, 1977.

[BS96] J. Van den Bercken and B. Seeger. Query Processing Techniques for Multiversion Access Methods. *Proc. of VLDB Conf.*, pp 168-179, 1996.

[BCS97] E. Bertino, B. Catania and B. Shidlovsky. Towards optimal two-dimensional indexing for constraint databases. *Information Processing Letters*, 64(1997):1-8.

[CCM+97] 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.

[CCM+98] 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.

[CR92] B. Chazelle and B. Rosenberg. Lower Bounds on the Complexity of Simplex Range Reporting on a Pointer Machine. *Proc. 19th Intern. Colloquium on Automata, Languages and Programming. LNCS,* Vol. 693, Springer-Verlang, Berlin, 1992.

[CF98] K. L. Cheung and A. Wai-Chee Fu. Enhanced Nearest Neighbour Search on the R-tree. In *SIGMOD Record*, 27(3): 16-21, 1998.

[CR99] J. Chomicki and P. Revesz. A Geometric Framework for Specifying Spatiotemporal Objects. *Proc. 6th International Workshop on Time Representation and Reasoning*, May 1999.

[Col86] R. Cole. Searching and Storing Similar Lists. *Journal of Algorithms*, 7(2):202-220, 1986.

[Com79] D. Comer. The Ubiquitous B-Tree. *Computing Surveys*, 11(2):121–137, June 1979.

[CLR90] T. Cormen, C. Leiserson and R. Rivest. Introduction to Algorithms. *The MIT Press,* Cambridge, Mass., 1990.

[DKG+99] 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.

[DPM98] V. Delis, D. Papadias, N. Mamoulis. Assessing Multimedia Similarity: A Framework for Structure and Motion. In *Proc. ACM Multimedia*, pp. 333-338, 1998.

[DKM+88] M. Dietzfelbinger, A. Karlin, K. Mehlhorn, F. Meyer, H. Rohnhert and R. Tarjan. Dynamic Perfect Hashing: Upper and Lower Bounds. In *Proc. 29th IEEE FOCS*, pp. 524-531, 1988.

[DSS+86] 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.

[ED88] R.J. Enbody, H.C. Du. Dynamic Hashing Schemes. In *ACM Computing Surveys*, 20(2), pp. 85-113, 1988.

[EN94] R. Elmasri, S. Navathe. Fundamentals of Database Systems., Second Edition, Benjamin/Cummings, 1994.

[EGS+98] M. Erwig, R.H. Guting, M. Schneider and M. Vazirgianis. Spatio-temporal Data Types: An Approach to Modeling and Querying Moving Objects in Databases. In *Proc. of ACM GIS Symposium '98*.

[ELS95] G. Evangelidis, D. Lomet, and B. Salzberg. The hB$^{\Pi}$-tree: A Modified hB-tree Supporting Concurrency, Recovery and Node Consolidation. In *Proc. 21st Intern. Conf. on Very Large Data Bases*, Zurich, September 1995.

[FNP+79] R. Fagin, J. Nievergelt, N. Pippenger, H. Strong. Extensible Hashing - a fast access method for dynamic files. *ACM Transactions on Database Systems*, 4(3), pp. 315-344, 1979.

[FBF+94] 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.

[FNS+92] A. Fiat, M. Naor, J.P. Schmidt, A. Siegel. Nonoblivious Hashing. *JACM*, 39(4), pp.764-782, 1992.

[FZR98] M.J. Folk, B. Zoellick, G. Riccardi. *File Structures*, Addisson Wesley, Reading, MA, 1998.

[GG98] V. Gaede and O. Gunther. Multidimensional Access Methods. *ACM Computing Surveys*, 30(2):170-231, 1998.

[GRS+97] J. Goldstein, R. Ramakrishnan, U. Shaft and J.B. Yu. Processing Queries By Linear Constraints. In *Proc. 16th ACM PODS Symposium on Principles of Database Systems*, pp. 257-267, Tuscon, Arizona, 1997.

[Gun89] O. Gunther. The Design of the Cell Tree: An Object-Oriented Index Structure for Geometric Databases. In *Proc. Fifth IEEE International Conference on Data Engineering*, Los Angeles, CA, USA, February 1989.

[Gut84] 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.

[HGH+97] 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.

[HKP97] J. Hellerstein, E. Koutsoupias and C. Papadimitriou. On the Analysis of Indexing Schemes. In *Proc. 16th ACM Symposium on Principles of Database Systems*, pp. 249-256, Tucson, May 1997.

[HHK99] J. Hellerstein, L. Hellerstein and George Kollios. On the Generation of 2-Dimensional Index Workloads In *Proc. ICDT 1999*, pp. 113-130, January 1999.

[HSW89] A. Henrich, H.-W. Six, P. Widmayer. The LSD–tree: Spatial Access to Multidimensional Point and Nonpoint Objects. In *Proc. 15th Intern. Conf. on Very Large Data Bases*, pages 45–53, Amsterdam, August 1989.

[HS92] E.G. Hoel and H. Samet. A Qualitative Comparison Study of Data Structures for Large Linear Segment Databases. In *Proc. ACM-SIGMOD International Conference on Management of Data*, pages 205-214, San Diego, June 1992.

[Jag90] H. V. Jagadish. On Indexing Line Segments. In *Proc. 16th International Conference on Very Large Data Bases*, pages 614–625, Brisbane, Queensland, Australia, August 1990.

[J+94] C.S. Jensen, editor et. al. A Consensus Glossary of Temporal Database Concepts. *ACM SIGMOD Record*, 23(1), pp. 52-64, 1994.

[JS99] C.S. Jensen and R.T. Snodgrass. Temporal Data Management. In *IEEE TKDE*, 11(1): 36-44, 1999.

[KF94] 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.

[KF93] I. Kamel and C. Faloutsos. On Packing R–trees. In *Proc. Second Int. Conference on Information and Knowledge Management (CIKM)*, Washington, DC, Nov. 1-5, 1993.

[KRV+93] P. Kanellakis, S. Ramaswamy, D. Vengroff and J. Vitter. Indexing for Data Models with Constraint and Classes. In *Proc. 12th ACM SIFACT-SIGMOD-SIGART Symposium on Principles of Database Systems*,pages 233-243 , Washington, D.C, 1993.

[KGT99] G. Kollios, D. Gunopulos and V. Tsotras. On Indexing Mobile Objects In *Proc. 18th ACM Symposium on Principles of Database Systems*,pages 261-272 , June 1999.

[KS91] C. Kolovson and M. Stonebraker. Segment Indexes: Dynamic Indexing Techniques for Multi-dimensional Interval Data. In *Proc. ACM SIGMOD Conf.*, pp 138-147, 1991.

[KTF98] 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.

[LST95] G.M. Landau, J.P. Schmidt, V.J. Tsotras. On Historical Queries Along Multiple Lines of Time Evolution. *Very Large Data Bases Journal*, Vol. 4, pp. 703-726, 1995.

[Lar78] P. Larson. Dynamic Hashing. *BIT*, Vol.18, pp.184-201, 1978.

[Lit80] W. Litwin. Linear Hashing: A New Tool for File and Table Addressing. In *Proc. of VLDB Conf.*, pp 212-223, 1980.

[LNS96] W. Litwin, M.A. Neimat, D. A. Schneider. LH*-A Scalable, Distributed Data Structure. *ACM Trans. on Database Systems*, 21(4), pp 480-525, 1996.

[LM92] T.Y.C Leung, R.R. Muntz. Temporal Query Processing and Optimization in Multiprocessor Database Machines. In *Proc. of VLDB Conf.*, pp. 383-394, 1992.

[LR96] M-L. Lo, C.V. Ravishankar. Spatial Hash-Joins. In *Proc. ACM SIGMOD Conf.*, pp 247-258, 1996.

[LS89] D. Lomet and B. Salzberg. Access Methods for Multiversion Data. In *Proc. ACM SIGMOD Conf.*, 315-324, 1989.

[Mat92] J. Matousek. Efficient Partition Trees. *Discrete and Computational Geometry*, 8 (1992), 432-448.

[NS98] M. Nascimento and J. Silva. Towards Historical R-trees. In *Proc. ACM Symp. on Applied Computing,* pp. 235-240, 1998.

[NST99] 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.

[Ove83] M.H. Overmars. The Design of Dynamic Data Structures. *LNCS vol. 156, Springer-Verlag*, Heidelberg, West Germany, 1983.

[OS95] G. Ozsoyoglu, R.T. Snodgrass. Temporal and Real-Time Databases: A Survey. *IEEE TKDE*, 7(4), pp 513-532, Aug. 1995.

[PG97] W. Pannebaker and D. Le Gal. MPEG Digital Video Compression Standard., Wiley, John & Sons, April 1995.

[PJ99] D. Pfoser, C.S. Jensen. Incremental Join of Time-Oriented Data. In *Proc. 11th Intl. Conf. on SSDM*, Cleveland, Ohio, July 28-30, 1999.

[RT98] S.V. Raghavan and Satish K. Tripathi. *Networked Multimedia Systems: Concepts, Architectures, and Design.*, Prentice Hall, 1998.

[R97] R. Ramakrishnan. *Database Management Systems*, 1st ed., McGraw-Hill, 1997.

[RKV95] N. Roussopoulos, S. Kelley and F. Vincent. Nearest Neighbor Queries. In *Proc. ACM-SIGMOD Int. Conf. on Management of Data*, pages 71-79, June 1995.

[SJL+00] S. Saltenis, C. Jensen, S. Leutenegger and Mario A. Lopez. Indexing the Positions of Continuously Moving Objects In *Proc. ACM-SIGMOD Int. Conf. on Management of Data 2000*, May 2000.

[SJ99] S. Saltenis and C. Jensen. R-Tree Based Indexing of General Spatio-Temporal Data. Time-Center, Tech-Report, TR-45, 1999.

[S88] B. Salzberg. *File structures.* Prentice Hall, Englewood Cliffs, NJ, 1988.

[SL95] B. Salzberg, D. Lomet. Branched and Temporal Index Structures. *College of Computer Science Technical Report*, NU-CCS-95-17, Northeastern University.

[ST99] B. Salzberg and V.J. Tsotras. A Comparison of Access Methods for Time-Evolving Data. *ACM Computing Surveys*, June 1999.

[Sam90] H. Samet. *The Design and Analysis of Spatial Data Structures.*, Addison-Wesley, Reading, MA, 1990.

[SD90] D.A. Schneider, D.J. DeWitt. Tradeoffs in Processing Complex Join Queries via Hashing in Multiprocessor Database Machines In *Proc. of VLDB Conf.*, pp. 469-480, 1990.

[SRF87] 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.

[SY91] S. Shekhar and T.A. Yang. Motion in a Geographical Database System In *2nd International Symposium on Advances in Spatial Databases*, pages 339–357, Zürich, Switzerland, August 1991.

[SC96] J. R. Smith and S.F. Chang. VisualSEEk: A Fully Automated Content-Based Image Query System. In *Proc. ACM Multimedia*, pp. 87-98, 1996.

[SYV97] A. P. Sistla, C. T. Yu, R. Venkatasubrahmanian. Similarity Based Retrieval of Videos. In *Proc. IEEE ICDE*, pp. 181-190, 1997.

[SWC+97] A. P. Sistla, O. Wolfson, S. Chamberlain, S. Dao. Modeling and Querying Moving Objects. In *Proc. 13th IEEE International Conference on Data Engineering*, pages 422-432, Birmingham, U.K, April 1997.

[SR95] S. Subramanian and S. Ramaswamy. The P-range Tree: A New Data Structure for Range Searching in Secondary Memory. In *Proceedings of the 6th Annual Symposium on Discrete Algorithms*, New York, NY, USA, 1995.

[Sub98] V.S. Subrahmanian. *Principles of Multimedia Database Systems.*, The Morgan Kaufmann Series in Data Management Systems, Jim Gray, Series Editor, 1998.

[SC99] 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.

[TUW98] J. Tayeb, O. Ulusoy, O. Wolfson. A Quadtree-Based Dynamic Attribute Indexing Method. *The Computer Journal*, 41(3):185-200, 1998.

[TS96] 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.

[TSP+98] Y. Theodoridis, T. Sellis, A. Papadopoulos and Y. Manolopoulos, Specifications for Efficient Indexing in Spatiotemporal Databases. In *Proc. SSDBM*, pp. 123-132, 1998.

[TSN99] Y. Theodoridis, J. Silva and M. Nascimento. On the Generation of Spatiotemporal Datasets. In *Proc. SSD*, pp. 147-164, 1999.

[TGH95] V. J. Tsotras, B. Gopinath, G.W. Hart. Efficient Management of Time-Evolving Databases. *IEEE TKDE*, 7(4), pp 591-608, Aug.1995.

[TJS98] V.J. Tsotras, C.S. Jensen, R.T. Snodgrass. An Extensible Notation for Spatiotemporal Index Queries. *ACM Sigmod Record*, pp 47-53, March 1998.

[TK95] V.J.Tsotras, N. Kangelaris. The Snapshot Index, an I/O-Optimal Access Method for Timeslice Queries. *Information Systems*, 20(3), 1995.

[TW99] V.J.Tsotras, X. Sean Wang. Temporal Databases. *Wiley Encyclopedia of Electrical and Electronics Engineering*, Vol. 21, pp 628-641, 1999.

[TVM98] T. Tzouramanis, M. Vassilakopoulos, Y. Manolopoulos. Overlapping Linear Quadtrees: A Spatio-Temporal Access Method. In *Proc. ACM-GIS,* pp. 1-7, 1998.

[VTS98] M. Vazirgiannis, Y.Theodoridis, T.K. Sellis. Spatio-Temporal Composition and Indexing for Large Multimedia Applications. In *Multimedia Systems*, 6(4): 284-298, 1998.

[VV97] P.J. Varman, R.M. Verma. An Efficient Multiversion Access Structure. In *IEEE TKDE*, 9(3): 391-409, 1997.

[XHL90] 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.

[WCD+98] O. Wolfson, S. Chamberlain, S. Dao, L. Jiang, G. Mendez. Cost and Imprecision in Modeling the Position of Moving Objects. In *Proc. 14th IEEE Intern. Conf. on Data Engineering*, pages 588-596, Orlando, FL, February 1998.

[WXC+98] O. Wolfson, B. Xu, S. Chamberlain, L. Jiang. Moving Objects Databases: Issues and Solutions In *Proceedings of the 10th International Conference on Scientific and Statistical Database Management*. Capri, Italy, July 1998.