DESIGN AND PERFORMANCE EVALUATION OF INDEXING METHODS FOR DYNAMIC ATTRIBUTES IN MOBILE DATABASE MANAGEMENT SYSTEMS

A THESIS
SUBMITTED TO THE DEPARTMENT OF COMPUTER
ENGINEERING AND INFORMATION SCIENCE
AND THE INSTITUTE OF ENGINEERING AND SCIENCE
OF BILKENT UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE

By
Jamel Tayeb
May, 1997

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Asst. Prof. Özgür Ulusoy (Supervisor)

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Prof. Varol Akman

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Asst. Prof. Attila Gürsoy

Approved for the Institute of Engineering and Science:

Prof. Mehmet Baray
Director of Institute of Engineering and Science

ABSTRACT

DESIGN AND PERFORMANCE EVALUATION
OF INDEXING METHODS FOR
DYNAMIC ATTRIBUTES IN MOBILE
DATABASE MANAGEMENT SYSTEMS

Jamel Tayeb
M.S. in Computer Engineering and Information Science
Supervisor: Asst. Prof. Özgür Ulusoy
May 1997

In traditional databases, we deal with static attributes which change very infrequently over time and their change is handled with an explicit update operation. In temporal databases, the time of change of attributes is also important and every update creates a new version. Attributes, typically change more frequently over time. A more agitated category of attributes are the so-called dynamic attributes whose value changes continuously over time, thus making it impractical to explicitly update them as they change. In this thesis, we conduct a performance evaluation study of two indexing methods for dynamic attributes. These are based on the key idea of using a linear function that describes the way the attribute changes over time and allows us to predict its value in the future based on any value of it in the past. The problem is rooted in the context of mobile data management and draws upon the fields of spatial and temporal indexing. We contribute various experimental results, mathematical analyses, and improvement and optimization algorithms. Finally, inspired by a few of the observed shortcomings of both of the studied techniques, we propose a novel indexing method which we call the FP-Index which is shown analytically to have promising prospects and to beat both methods over most performance parameters.

Keywords: Mobile Data Management, Spatial Indexing, Dynamic attributes.

ÖZET

HAREKETLİ VERİ TABANI SİSTEMLERİNDE DİNAMİK ÖZNİTELİKLER İÇİN İNDEKSLEME YÖNTEMLERİ TASARIMI VE PERFORMANS ÖLÇÜMÜ

Jamel Tayeb
Bilgisayar ve Enformatik Mühendisliği, Yüksek Lisans
Tez Yöneticisi: Yard. Doç. Dr. Özgür Ulusoy
Mayıs 1997

Genel amaçlı veri tabanlarını oluşturan öznitelikler, nadiren ve bazı işlemlerin uygulanması sonucu değer değişikliklerine uğradıkları için, durağan öznitelikler olarak adlandırılmaktadırlar. Zamansal veri tabanlarında, özniteliklerin değişikliğe uğrama zamanları da önemlidir ve her değişiklikte özniteliğin yeni bir versiyonu yaratılır. Bu tip veri tabanlarında öznitelikler durağan özniteliklere oranla genelde daha sık değişikliğe uğrarlar. Özniteliklerin, dinamik öznitelikler olarak adlandırılan bir diğer kategorisi ise, zamana bağlı olarak devamlı değişme özelliğine sahiptir ve bu nedenle, her değer değişiminin bir işlem aracılığıyla veri tabanına yansıtılması sisteme çok büyük bir yük getirecektir. Bu tezde, dinamik öznitelikler üzerinde iki değişik indeksleme metodunun performans analizi gerçekleştirilmektedir. Bu yöntemlerdeki temel fikir, doğrusal bir işlev aracılığıyla, özniteliklerin geçmişteki değerlerini de kullanarak, gelecek için alabilecekleri değerlerin belirlenmesidir. Bu yöntemlerin araştırılması, hareketli veri tabanı yönetimi alanı temel alınarak gerçekleştirilirken, zamansal ve uzaysal indeksleme konularından da gerekli yerlerde faydalanılmıştır. Tezin başlıca katkıları, çeşitli deneysel araştırma sonuçlarının sunulması yanında bazı matematiksel analizlerin gerçekleştirilmesi ve elde edilen sonuçlar ışığında FP-Index olarak adlandırdığımız yeni bir indeksleme yöntemin geliştirilmesi olmuştur. Yeni yöntemin diğer indeksleme yöntemlerine olan üstünlüğü matematiksel analiz yoluyla kanıtlanmıştır.

Anahtar sözcükler: Hareketli Veri Tabanı Yönetimi, Uzaysal İndeksleme, Dinamik Öznitelikler.

ACKNOWLEDGMENTS

First and foremost, I would like to express my deepest thanks and gratitude to my advisor Asst. Prof. Özgür Ulusoy for his patient supervision of this thesis. He granted me the right amount of independence which was particularly beneficial to the progress of this work. His kind acceptance to finish my thesis at this stage is also deeply appreciated. Special thanks go also to Prof. Ouri Wolfson for providing the substance of this research work and for his valuable comments. I am grateful to Prof. Varol Akman and Asst. Prof. Attila Gürsoy for reading the thesis and for their instructive comments.

I would also like to thank my friend Elkafi Elhassini for his faith in me and his valuable advice. Finally, My thanks go to my brother Soufien and sister Lamia for their categorical encouragement and insistence on my continuing in the path of research.

Dedicated to my brother Soufien and sister Lamia

Contents

Chapter 1

INTRODUCTION

The recent technological progress in portable laptop computers and the smaller palmtop computers led to the speculation that they will be ubiquitous in the near future. They are becoming increasingly more powerful, cheaper, and dotted with the ability to communicate with fixed networks via the wireless medium. If they become cheap enough we expect to have hundreds of thousands of people (some even talk about millions) moving around with such devices, each person having his own information needs. Users' information needs will be particularly stimulated with the mobile devices' ability to communicate. Both this ability to communicate with hosts in a fixed network and the expected information demands of these mobile users challenged the research community with new problems and started up the new research field of mobile computing. We are here at one of its very young branches; namely the field of mobile data management.

The standard architecture of a mobile computing system consists of a small set of fixed hosts linked via a wide area network and a larger set of mobile computers which would occasionally request services from the fixed hosts. Typically, the fixed hosts have ample computational power and a large storage capacity while the mobile computers are limited by both CPU speed and storage capacity as well as battery resources. A small set of special fixed hosts are called *mobile support stations* and are dedicated to serving the wireless communication needs of the mobile users and also linking them to the resourceful

network which will supply them with various (commercial) services. There then immediately arises the necessity to handle mobility of the users (among other things).

Location is the most important piece of information in a mobile system. It gives rise for example to a new paradigm of queries called *location-dependent queries* [IB92] which are queries whose answer depends on the location from which they were issued. Consider a person driving a car who occasionally wants to be informed about motels that are within five miles of his location in order to select a reasonably priced hotel. Such a service would be valuable to any traveler (especially at night) and would indeed make his search for a suitable motel a very easy and comfortable task. It is clear that the set of motels computed as an answer to his query would be different each time his car moves by a reasonable distance.

Alternatively, the driver could request the information to be continuously updated on his on-board computer screen as the car moves. For this to be possible, the mobile support stations will need to track his position in an efficient way that does not involve very frequent communication nor the unpleasant and unrealistic prospect of having to update the position attribute in a quasi-continuous manner. Remember also that there might be thousands of moving vehicles each with similar demands. A solution that was first proposed in [SWCD97] is for the moving object/vehicle to supply its motion equation and current position which would then be used by the mobile support stations to predict the position in the 'near' future. This way, we will be able to answer location-dependent queries without the need to communicate very frequently with the moving object to check its location. Since the motion equation is just an approximation, we expect that after some time the discrepancy between the object's real position and its position as perceived by the mobile support station will reach an unacceptable magnitude. This may start to compromise the correctness of the supplied answers. However, it suffices to update the position occasionally (or the motion function or both) to remedy this problem. The issue of imprecision modeling in the context of this application is treated in more detail in [WCD+97].

Consider then the case where we have a large number of such mobile users.

At any given time, we need to track their positions. The focus of our work is on queries on the set of objects themselves such as which subset will cross a given segment of space during a given time slice. This is in contrast to queries issued by the mobile objects seeking information about their surrounding space. If we view position as one attribute of the moving object, then it is indeed a peculiar kind of attribute in that it is continuously changing with time without explicit 'continuous' updates on the database storing it. Attributes with such a property were called *dynamic attributes* in [SWCD97]. At this point we are ready to describe the goal of the research work presented in this thesis and state our major contributions.

The purpose of this work is to conduct a thorough experimental evaluation of two indexing techniques for dynamic attributes which support a variety of range searches. Besides the experimental study which is based on simulation, we occasionally resort to modest mathematical analysis of a few performance parameters. We present an index construction algorithm that is optimal in the disk access and CPU costs. It is given as an improvement over a naive algorithm. The thesis is further enriched by our contribution of a novel indexing method that amends many of the shortcomings of the first two. In the context of this new method, we present an optimal query processing algorithm that handles one category of queries very efficiently and can potentially reduce the cost of a single query to less than one disk access. We leave any further details to the forthcoming chapters.

The outline of this thesis is as follows. In Chapter 2, we describe the research problem in full detail and also introduce the two indexing techniques to be studied. Chapter 3 provides some background information to introduce the field into which our work is situated. We also describe related work and briefly survey from the literature what we deemed worth mentioning or what was helpful to us in the early stages of our work. The next three chapters represent the core of the thesis. Chapter 4 provides the results of studying the first technique which is the quadtree method. Chapter 5 is dedicated to the second indexing technique called the cross point method. In Chapter 6, we introduce our own novel access method which we dubbed the FP-Index. Its conception draws largely upon the insight gained from studying the previous

two techniques. Finally, we present our conclusions in Chapter 7.

Chapter 2

PROBLEM DESCRIPTION

In our work, we conduct a performance study of two indexing techniques for dynamic attributes. The purpose of this chapter is to elaborate on this problem statement and introduce the techniques to be studied. We start in Section 2.1 by introducing prerequisite concepts that are central to our study. After that, we describe the first technique which is called the quadtree method in Section 2.2. The second technique named the cross points is presented in Section 2.3.

2.1 Introductory Notions

Our work is largely based on the ideas introduced by Sistla et al. in [SWCD97]. The authors present a new data model suitable for representing moving objects in database systems. The model is called the *Moving Objects Spatio-Temporal* (MOST) data model and relies on the key idea of representing the position as a function of time. In other words, each object is associated with a motion function that allows us to predict its position in the future given its position at some point in the past. If the position of the moving object is one of its attributes, then this technique allows us to avoid the naive and unrealistic scheme of having to update the position attribute 'whenever' it changes. A peculiar property of such an attribute is that it changes continuously with

time. We are familiar with the static attribute whose value changes in a discrete (as opposed to continuous) way over time and have to be explicitly updated upon each such change. Examples include the salary attribute in an employee relation. However, we are here faced with 'attributes that change continuously with time without being explicitly updated' [SWCD97]. Such attributes were given the name *dynamic attributes* by the authors of [SWCD97]. Below, we present their description of the characteristics of such attributes in the form of a definition.

Definition 1 *A* **dynamic attribute A** *is represented by three sub-attributes, A.value, A.updatetime, and A.function where A.function is a function of a single variable t that has value 0 at t = 0. The semantics is that at time A.updatetime the value of A is A.value, and until the next update of A, the value of A at time A.updatetime+t_0 is given by A.value+A.function(t_0). An update of a dynamic attribute may change the value or function sub-attributes (or both).*

We thus want to explore how we could efficiently index and retrieve information about such attributes. We next need to describe the types of queries expected or supported.

In [SWCD97], three types of queries are discerned for the MOST data model. These are the *instantaneous*, the *continuous*, and the *persistent* queries. An instantaneous query submitted at time t_i is processed against the database state at t_i. The following example is given in [SWCD97]: 'Display the motels within 5 miles of my position.' A continuous query submitted at time t_i is processed against all database states starting from t_i (i.e., $[t_i \ldots \infty)$). It is described as 'an instantaneous query being continuously reissued at each clock tick.' In the motels example, the user will just require to be 'continuously' informed about which motels are coming within 5 miles of his position. If we let S_j denote the state of the database at time t_j, then at each $t_j > t_i$, the continuous query is reevaluated against S_j. The persistent query is a bit more demanding. Like the continuous query, it has to be evaluated at each clock tick after its time of submission t_i. However, unlike the continuous query, at t_j it has to be evaluated against the set of states $S_i, S_{i+1}, \ldots, S_{j-1}, S_j$ rather than against

S_j alone. The example query provided in [SWCD97] is the following: 'let me know when the speed of object o in the direction of the x-axis doubles within 10 minutes'. Persistent queries arise in the expression of temporal triggers in active database applications [SW95].

In our research, the focus is on supporting instantaneous and continuous queries. However, the semantics we have in mind is slightly different from that exhibited in the examples provided by Sistla et al. in [SWCD97]. In the paper, the authors consider queries issued by the moving objects themselves (i.e., the mobile client) that mainly seek information about spatially static objects such as motels, restaurants, hospitals, etc. Our focus in this study is on range queries which ask about the moving objects themselves. The generic and generalized form of our queries is the following.

> Give me all the objects whose value for attribute A falls in the attribute range $[a_i \ldots a_j]$ at some time between time instances t_b and t_e.

Let t_{now} denote the time at which the query was submitted. Depending upon the values for t_b and t_e, we may have any of the following four types of queries:

1. If $t_b = t_e = t_{now}$, we have an instantaneous range query asking about the present.
2. If $t_b = t_e = t_i$ and $t_i > t_{now}$, we have an instantaneous range query asking about the future.
3. If $t_b = t_{now}$ and $t_e = \infty$, we have a continuous range query.
4. If $t_b = t_i$ and $t_e = t_j$ $(t_{now} \leq t_i < t_j)$, we have a general two-dimensional range query over the attribute and time dimensions.

We remark that in practice, we map the first three types of queries to the fourth one which is more general. In instantaneous range queries (cases 1 and 2), we approximate the time point t_i by the time range $[t_i - \delta t \ldots t_i + \delta t]$ where

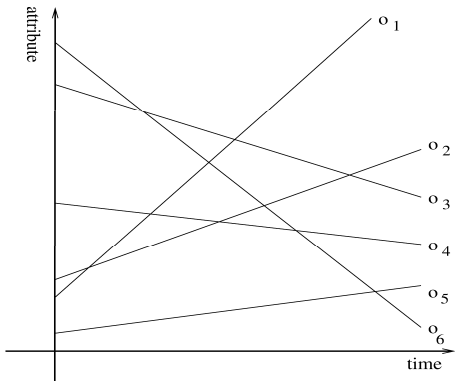

Figure 2.1: Trajectories of moving objects in the time-attribute space.

δt is a small time lapse. In continuous queries, the theoretically infinite range $[t_b \ldots \infty)$ is usually decomposed into the set of contiguous intervals

$$[t_i \ldots t_i + \Delta T], [t_i + \Delta T \ldots t_i + 2\Delta T], \ldots, [t_i + n\Delta T \ldots t_i + (n+1)\Delta T], \ldots \quad (1)$$

This again reduces it to the fourth case. We return back to the key idea behind the management of dynamic attributes.

As mentioned above, we tame the rapid change of object positions by using a motion function; the idea is due to [SWCD97][1]. Each object upon entering the system supplies its current position and its motion function and may later on update both of them. Since in practice the objects will not follow perfectly their motion functions, answers to queries are only approximate and are bound to contain some amount of error. This is expressed in [SWCD97] with the following remark: 'the answer to future queries is tentative and should be regarded as correct according to what is currently known about the real world'. The work of Wolfson et al. [WCD+97] is devoted to this specific topic of imprecision and the provision of bounds on the amount of error. However, it is more specialized to the context of moving vehicles across a given route.

In our work, we consider objects with linear motion functions $f(t) = at + b$; no other types of functions are considered. This work could however in principle be extended to quadratic or more complicated functions. We leave that for future work. With the motion function in hand, we could plot the trajectory of each object in the time-attribute space as shown in Figure 2.1. In fact

[1] " ... we propose to solve this problem by representing the position as a function of time; it changes as time passes without an explicit update."

the two indexing methods we study in this thesis take the plotted trajectories as a starting point. The two methods are the cross points and the quadtree method and are due to Wolfson [Wol96]. Below, we describe the details of each approach.

2.2 The Quadtree Method

As we mentioned above, since we can plot the objects' trajectories in two-dimensional space, the problem of dynamic attribute indexing is transformed into a spatial indexing problem. For this, we could draw upon the literature for spatial access methods [Sam89] and adapt one access method to our specific problem. We have selected the quadtree indexing structure.

The quadtree is treated thoroughly in [Sam84] with several of its variants. The idea common to all quadtree variations is the recursive decomposition of indexed space. However, we are interested in the so called *region quadtree* which is based on the successive subdivision of space into four equal-sized quadrants. This is shown in Figure 2.2. Among the region quadtree variants, we are interested in a special class called *PR quadtrees*. In the PR quadtree, we recursively partition quadrants until we have no more than one data element stored in every leaf. The main disadvantage of this policy is that if we have two points that are very near to each other, we will need to have a very fine partitioning of indexed space just for the sake of having those two points fall into different subquadrants. This results in an unnecessarily deep quadtree directory. We note at this point that the quadtree is mainly used as an in-memory index. Samet's discussion of quadtree variants in [Sam84] draws largely from its use in image processing applications. Adaptation of the PR quadtree to handle data stored in disk pages yields the so called *bucket PR quadtree*. In the bucket PR quadtree, given a set of N points we partition the indexed space recursively until no more than B points fall in every single subquadrant. We call B the *bucket size*. Typically, B will be equal to the number of data records that fit in a single disk page. Let us then look into the structure of the quadtree directory.

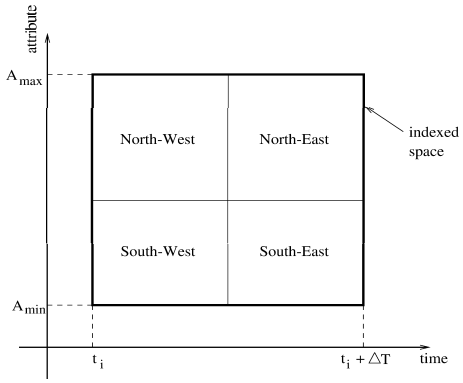

Figure 2.2: Partitioning of the indexed space in the region quadtree.

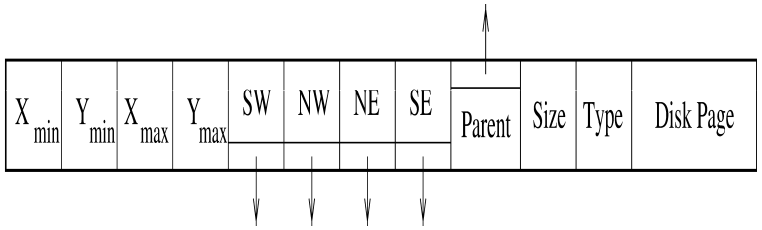

Figure 2.3: Record structure of the quadtree node.

Like a binary tree, a node in the quadtree directory has to embody enough information to direct the search for data to the appropriate lower subtrees. In other words, it has to ensure proper branching. It also has to have pointers to its four subtrees. This is the minimum information that we have to include in the record definition for quadtree nodes. For the information that guides searches, we choose to include the boundaries of the subquadrants of space that is being indexed by the tree rooted at the current node. The coordinates (X_{min}, Y_{min}) and (X_{max}, Y_{max}) of the lower left and upper right corners (respectively) of the indexed quadrant are enough. We then need four pointers to the south-west, north-west, north-east, and south-east children of the node. These will take the value NULL if the node is a leaf node. The complete record structure of a quadtree node is shown in Figure 2.3. The *Parent* field points to the parent node and takes the value NULL for the case of the root. The *Size* field counts the number of index elements in the data page pointed to by a leaf node. It is needed to detect bucket overflow and underflow and is not used for internal nodes. The *Type* field differentiates between leaf and internal nodes for the search, insertion, and deletion operations. Finally, the disk page field is a pointer to the page on the disk which contains the data points falling

in the subquadrant indexed by the current node. This field is also active only in leaf nodes and is not used by internal nodes. We then look at how we adapt the bucket PR quadtree to our application.

The idea is the following. We start with an empty data page that is supposed to contain index elements from the whole data space. Every trajectory is known to cross the initial quadrant since it encompasses the whole data space. Because of this, we insert the first B trajectories' associated data with no problem. The index elements consist of the object ID together with the intercept b and slope a of its trajectory's equation $f(t) = at + b$. A bucket split will occur at the $B + 1^{st}$ insertion; it is executed as follows. We take the (a, b) pair of each object in the bucket and use it in conjunction with the bucket boundaries $X_{min}, Y_{min}, X_{max}, Y_{max}$ to find out which of the four subquadrants the trajectory crosses. We then insert the corresponding $< ID, a, b >$ record in every crossed subquadrant (there can be at most three out of four). An example of this is shown in Figures 2.4 and 2.5. A bucket split requires that we allocate four new disk pages, one for each subquadrant. Upon completion of bucket split computations, the father data page is disposed of. The leaf quadtree node that was pointing to it must itself allocate memory for four leaf nodes which will point to the four newly allocated disk pages. It thus becomes an internal node. Furthermore, the boundaries of the four subquadrants have to be computed from the boundaries of the father quadrant. In our application, the quadtree directory consisting of the search nodes will reside in memory whereas data pages are on the disk.

Once the application is running and the tree has been constructed, we would execute a search for an object or trajectory in the following manner. The linear motion equation $f(t) = at + b$ is our search key and the ID of the search object is used once we reach the relevant data pages. Starting at the root of the quadtree, we use the (a, b) pair and the boundary fields $X_{min}, Y_{min}, X_{max}$ and Y_{max} found in the root to decide 'which ways to go'; that is, which subquadrants are crossed by the sought object. We then descend the next level and repeat the same thing until we reach the relevant leaves at which point we use the disk page pointer field to bring the relevant data pages into the buffer (if they are not there). The difference between the quadtree and the binary tree searches

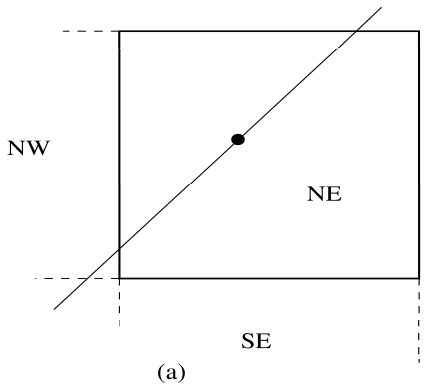

Figure 2.4: An object's trajectory crossing the north-east subquadrant of a larger quadrant.

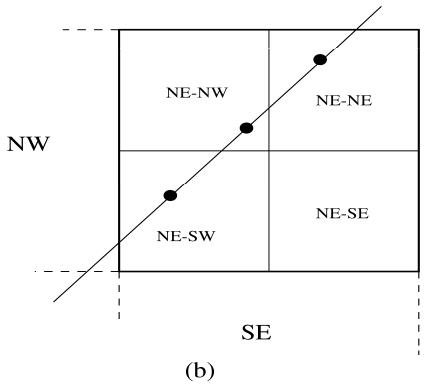

Figure 2.5: Upon overflow, the father quadrant itself splits into four subquadrants and the object generates three copies in the south-west, north-west, and north-east subquadrants.

is that in the binary tree we branch only one way while in our application of the quadtree we can have up to three ways to branch from a single index node. The search procedure as described is the basis of insertion, deletion, and update operations. Traversal of the quadtree to answer range queries is done in an analogous manner except that the search key consists of the boundaries of the range and we check for overlap between the range and the quadrant at each node reached. The study of performance of the quadtree method is presented in Chapter 4.

2.3 The Cross Points Method

The idea of the cross points method is illustrated in Figure 2.6. for a simple system with only three objects. A *cross point* is the name given to the intersection between two trajectories. Given the plotted trajectories, we record all the intersections between all the pairs of objects (or only those which will take place in the 'near' future). An intersection or a cross point is defined by the time t_{CP} at which it takes place and the IDs of the two objects participating in it. Now, since at any point in time t_i each object has a fixed value on the attribute dimension computable from its associated linear function of time,

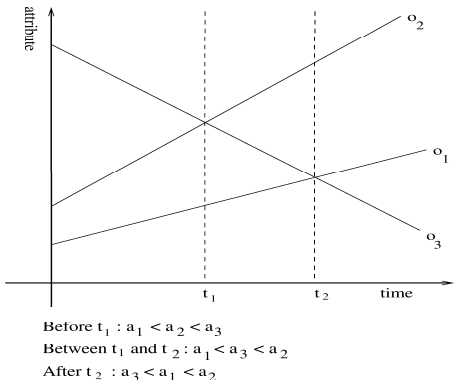

Figure 2.6: The basic idea of the cross points method: between intersection events, the order is fixed.

there is a unique ordering $O(t_i) =< a_{i1}, \ldots, a_{iN} >$ of the attribute values of the objects in the system. The key observation upon which the method is based is that between any two cross point events, the ordering of the objects' attribute values does not change. More formally, if t_{CP1} and t_{CP2} are the times at which the first and second intersection events took place, then $\forall t \in [t_{CP} \ldots t_{CP2})$ we have $O(t) = O(t_{CP1})$. It then suffices to store this order in any unidimensional index or data structure and maintain it whenever an intersection event takes place by simply flipping the order of the two objects which intersected. Between cross point events, we are able to answer any instantaneous range queries about the present by executing a range search over the indexing structure and retrieving the IDs of all the objects found in the range. A few observations are in order.

It is immediately clear that for N objects we have to store N data elements or records. In other words, no duplication of any kind is necessary. Moreover, we also have to store information about all the cross point events that will take place in the future. To do this, we compute the forthcoming intersection events periodically. Denote the period by ΔT. More precisely, at time point t_i we compute all the cross points which occur in the future time slice $[t_i \ldots t_i + \Delta T]$. For this reason, we call ΔT the *lookahead interval.* Another important observation is that we have to store the cross points in sorted order according to their occurrence in time. We finally state that the cross points method does not support continuous queries. The problem of adapting it to do so is left as a future work. We have chosen the binary tree as the data structure for storing

the sorted attribute order of the N objects in the system. Besides, we store the set of forthcoming intersection events in a linked list with the earliest cross point at the head of the list. The study of performance of the cross points method is presented in Chapter 5.

Chapter 3

BACKGROUND AND RELATED WORK

In this chapter, we provide an overview of the relevant research literature and introduce the necessary background for the coming chapters. In Section 3.1, we talk about the indexing problem in general. Section 3.2 introduces the particular field of spatial indexing. Section 3.3 talks about access methods that involve the time dimension. In Section 3.4, we talk about performance evaluation studies and introduce major performance criteria for assessing indexing techniques. In Section 3.5, we introduce the field of mobile data management. Finally, we include a brief mention of the field of computational geometry in Section 3.6.

3.1 Indexing in General

Indexing is at the heart of the field of database management. Whenever there is a repository of data, there immediately arises the problem of accessing it efficiently which is the indexing problem stated in its most general form. Over the past decades, the research community has continuously been challenged by new forms of the indexing problem. Factors accounting for this renewal include the advent of new hardware, new disk technologies, the prospects of

exploiting concurrency or parallelism, and above all new kinds of data and more challenging database sizes. These factors stimulated researchers to design new access methods and improvements or variants of existing ones. The purpose of this section is to report on prior work on indexing that is most relevant to our research or that helped us in indirect ways to improve our understanding of the indexing problem.

The ubiquitous B-tree [Com79] (and its even more ubiquitous variant the B^+-tree) is the father of many of today's popular access methods. It is the successful secondary storage indexing solution for single attribute data over an ordered attribute space. We assume the reader is familiar with its basic principles. We just remind here that the B^+-tree stores all data records at the leaf nodes and uses the non-leaf nodes (index nodes) only as a search directory. The B-tree on the other hand, uses the nodes as both a data repository and an indexing directory. The typically large fanout of B-trees has the consequence that the height of the B-tree index rarely exceeds two [Sal88]. Given that the root is always stored in main memory, most insertions, deletions, and exact match queries require no more than four disk accesses [SL91]. B-trees continue to be studied until recently though from different perspectives. An example is the field of concurrency control where special techniques were needed to 'prevent indices from becoming concurrency bottlenecks' [SC91]. B-tree concurrency control algorithms were experimentally evaluated in [SC91] where more than a dozen such algorithms are cited. In [SL91], an attempt is made to parallelize B-trees by storing a single B-tree on multiple disks. Among the techniques proposed are *record distribution* that relies on partitioning the key space, *page distribution*, and the use of large pages where each page is fragmented and its fragments are stored on multiple disks.

Underlying B^+-trees is a general philosophy of indexing captured in Lomet's notion of *Grow and Post Index Trees* (GP-trees) which describe a class of methods. The main characteristics are given in the following excerpt taken from [Lom91].

> GP-trees have nodes that map to disk blocks and possess a growth paradigm in which adding data in a key range leads eventually to

> growth in the space used locally to hold it, together with the posting of a new index term or terms for the growing space to parent nodes.

According to Lomet, 'B-trees represent both the genesis and the archetype of the class of GP methods' [Lom91]. The paper provides an interesting insight and many useful remarks on access methods in general and guidelines on adapting GP-tree like indices to spatial data. Remember that growth of B^+-trees occurs by splitting a full node into two reducing the local storage utilization from 100% to 50%. This results in an average space utilization of about 69% [Sal94]. In [Lom87] and [Lom88], a new technique called *partial expansion* is used that improves utilization to 83%. In this technique, two node sizes are used, a small and a large one. When a small node overflows, instead of doubling space locally through splitting its data into two small nodes, the small-sized node is replaced by the large-sized one (which obviously is less than double the size of small nodes). When the large node in turn overflows, it is replaced by two small nodes. This way, growth is more smooth since doubling of local space occurs over two steps. The papers include a thorough analysis of the technique considering also its more generalized version where doubling occurs over an arbitrary (finite) number of steps. The same problem of improving bucket utilization is explored in [Hen96] in the context of spatial data and for k-d tree based access methods. The heuristic used is essentially the same: 'In order to improve the bucket utilization, bucket splits have to be avoided whenever possible' [Hen96]. The idea is to perform local redistributions of objects over adjacent buckets, provided this does not increase the imbalance of the indexing structure. From this more or less general exposition on indexing we move to the more specialized field of spatial data indexing.

3.2 Spatial Indexing

Spatial data is simply n-dimensional data defined over the n-dimensional Euclidean space. It ranges from a set of points in 2-dimensional space (which is the focus of our specific research problem) to a set of points in n-dimensional space ($n > 2$) where we have the problem of multiattribute indexing. In both

cases, we have points which are objects of *zero size*. Alternatively, we may have objects of *non-zero size* which range from a collection of (2-dimensional) rectangles to hyperrectangles and hypervolumes in n-dimensional space. The B^+-tree, being essentially for single attribute indexing or unidimensional data, is not suitable for solving the spatial indexing problem. A classical source on spatial data structures is [Sam89]. Samet's treatment of the subject is divided into point data, curvilinear data, volume data, and data that consist of a collection of small rectangles (a typical VLSI application). The emphasis is on hierarchical methods that rely on a recursive decomposition of the indexed space into disjoint subspaces. [GB90] identifies four major application domains of spatial data management. These are mechanical CAD (described as 'the application with the most demanding geometrical requirements') and constructive solid geometry, VLSI CAD, vision systems (in robotics and manufacturing), and geographic and cartographic applications (which are most relevant to our problem). These have given rise to novel access methods which themselves stimulated many improvements yielding several variants. Let us then mention some notable spatial indexing techniques.

One of the most popular spatial indices is the R-tree [Gut84]. It associates with each object its *minimal bounding rectangle* (MBR)

$$([a_0, b_0], [a_1, b_1], \ldots [a_{n-1}, b_{n-1}])$$

where $[a_i, b_i]$ denotes the extent of the object along the i^{th} dimension. The indexed space is also represented by similar n-dimensional hyperrectangles and the index structure is similar to that of the B-tree. Each index node is an array of entries wherein each entry contains the n boundaries of a hyperrectangle and a pointer to the node encompassing all objects whose MBRs are contained in that hyperrectangle. A peculiar feature of the R-tree is that the indexing hyperrectangles may overlap. This means that the indexed space is not divided into disjoint subspaces. The consequence of this on the search operation is that 'more than one subtree under a node visited may need to be searched' [Gut84]. This is because an object's bounding rectangle may be situated inside two or more covering (and overlapping) hyperrectangles. An example of this for the two dimensional case is depicted in Figures 3.1 and 3.2. Furthermore, insertions and splits require some reorganizations and make use

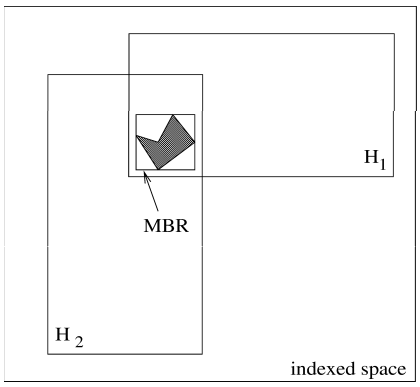

Figure 3.1: Overlapping hyperrectangles in R-tree representation of the indexed space.

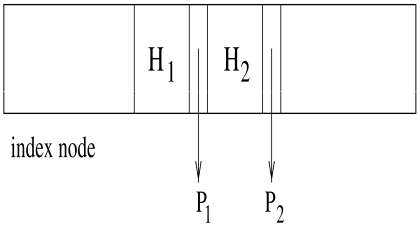

Figure 3.2: Both pointers P_1 and P_2 will have to be followed when searching for object o bounded by MBR.

of a heuristic optimization which consists of trying to minimize the area of the hyperrectangles that encompass objects MBRs.

Several variants of the R-tree have been proposed in the literature. Among them, we count the R^+-tree [SRF87] and the R^*-tree [BKSS90]. In the latter, Beckmann et al. question the choice of the heuristic that minimizes the area of hyperrectangles and propose to minimize the overlap between them, the overall storage utilization, and the margin[1] of hyperrectangles. A more recent proposal is the X-tree [BKK96] which starts from the observation that R-tree like index structures 'are not adequate for indexing high-dimensional data sets.' An emphasis is again put on minimizing the overlap between the index's bounding boxes, and the so called *supernodes* are used to mitigate the effects of high overlap which quickly becomes a dominant phenomenon at higher dimensionality. Another recent R-tree based index is the SS-tree [WJ96] which uses ellipsoid bounding regions.

Other spatial access methods include the *filter tree* of Sevcik and Koudas [SK96]. It is based on the principle of separating the objects by size, placing larger objects at higher levels of the tree and smaller ones at the bottom levels. It yields good performance for range queries and spatial joins. A spatial index that departs slightly from the R-tree is the *cell tree* of Günther and Bilmes [GB91]. It solves the same problem of indexing multidimensional objects of

[1]The margin is the sum of the lengths of the edges of a rectangle [BKSS90].

non-zero size but does not use bounding rectangles. Instead, it uses clipping and represents objects as unions of convex cells called *convex chains*. Cell trees are height-balanced and have the same structure as R-trees except that hyperrectangles in index nodes are replaced by convex polyhedra.

All the above methods are for objects of non-zero size in n-dimensional space. Another version of the spatial indexing problem alluded to earlier is multiattribute indexing where objects are simply points in n-dimensional space. An example of such access methods is Lomet and Salzberg's hB-tree or *holey brick B-tree* [LS90a]. It is also similar in spirit to the B-tree but its indexed regions may have holes or are 'bricks with perhaps smaller bricks removed from them'. The list of spatial access methods is definitely long. We should mainly retain that we are indexing a two-dimensional space where time is one of the dimensions. For this reason, both spatial indexing structures and indexes where the time dimension is involved might prove useful in our research.

3.3 The Time Dimension

Since we are indexing data that changes dynamically over time, the notion of time is crucial when approaching our problem. A brief digression on access methods which involve time is in place.

Relevant areas include mainly temporal and multiversion access methods. A prominent temporal index is the *time index* [EWK90]. Data records have interval attributes of the form $[t_s, t_e]$ recording the start and ending of the record's valid time. Since there is a total order on the values of t_s and t_e in the system, they could be used for indexing. Quoting Elmasri et al. [EWK90] 'the idea behind our time index is to maintain a set of linearly ordered indexing points on the time dimension'. In fact, they use the points t_s and $t_e + 1$ in conjunction with a simple $B+$-tree. An append only policy is adopted where no deletions are allowed so that the database size can potentially grow without bounds. To remedy this, Elmasri et al. assume the existence of a purge mechanism which cuts off data pages that are old enough or transfers them to archival storage. The time index is important to us mainly because we suspect

that it might somehow be adapted to solve our problem. In fact, an interesting research problem is whether we could map our problem of indexing dynamic attributes to the temporal indexing domain. Another question is whether we could have in our solution a way to purge 'old' data. This would relieve us of the need to destroy and rebuild our index periodically.

The latter property is more pronounced in the so called *time-split B-tree* or TSB-tree of Lomet and Salzberg [LS89]. It is an access method for multiversion data that also follows a non-delete policy and 'migrates data incrementally from a magnetic disk to an optical disk'. In the TSB-tree nodes, both timestamps and keys are used for indexing. When data nodes overflow, they could be split either on the key or timestamp (hence time) attributes. Splits based on timestamps are called *time splits* and are the mechanism for purging historical data 'one node at a time' [LS89]. Our interest in the TSB-tree comes from its smooth progress along the time dimension as newer timestamps dominate the tree and older data are moved to optical (write-once) disks where they become historical data. Although the methods we study in this thesis for dynamic attribute indexing do not have such a feature, we would still like to find one that supports it. In [LS90b], an analytic and experimental study of the TSB-tree's performance is provided.

We remark that many multiversion access structures are based on the B^+-tree. This is mainly because a partial order could easily be imposed on timestamps or valid times. Lanka and Mays [LM91] propose what they call a *fully persistent B^+-tree* for multiversion data where they use a graph for storing the information on the partial order formed by versions. In [BGO+93], a technique is presented for transforming a normal (single version) B-tree into a multiversion B-tree. The authors claim that the technique is general and could apply to a number of spatial and non-spatial hierarchical external access structures. Furthermore, the resulting multiversion B-tree is proved to be asymptotically optimal in the worst case in time and space for several operations and queries.

3.4 Performance Studies

A performance study, whether analytic or experimental, is an integral part of any work on access methods. Moreover, comparative studies are more important once a multitude of indexing techniques have been proposed as solutions to the same problem. Zobel et al. [ZMR96] list 'four principal ways of comparing algorithms such as indexing techniques' which are direct argument, mathematical modeling, simulation, and experiment. Our work is largely based on simulation experiments. However, where experiments are missing we resort to argumentation and also make modest attempts at providing a mathematical analysis whenever it illuminates better and improves understanding.

Our interest in performance studies is three-fold. First, we would like to have an idea about the real values used in the literature for some hardware parameters such as typical disk page sizes, buffer sizes, or disk access times. Let us take the example of page size for which we adopt the value of 4 Kilobytes. This is a standard value which we found in most experimental studies [Sal94, Lom91, SL91, BKK96, BKS96]. In [Sal94], Salzberg states the following: 'in 1994, minimum size pages are usually 4K'. In [GB91], Günther and Bilmes base their experiments on page sizes of 512 bytes, 1024 bytes, and 2048 bytes; this is the only exception we found. Another example parameter is the time it takes to access or fetch a disk page. We use minimum and maximum values of 10 msec and 30 msec respectively. Again, Salzberg [Sal94] states that 'each fetch of a page takes about 10 msec on average on the fastest disk drives'. We quote a more accurate characterization of disk access cost from [BKS96].

> ... In the following, we assume an average seek time of 9 msec, an average latency time of 6 msec, and a transfer time for one page (i.e. for 4 KB) of 1 msec. These parameters are typical values for current disks and result in 16 msec for reading a page.

The second reason why we are interested in surveying performance studies is to have an idea about the comparative behavior of the important and popular access methods in the literature. We have to say however that studies comparing more than two methods are rare. A notable example of such a study is

the comparative performance evaluation of spatial access methods conducted by Greene [Gre89]. Four indexing schemes were implemented which are the R-tree, R^+-tree, K-D-B-tree, and $2D$-ISAM. The paper concludes that the R-tree 'provides the best tradeoff between performance and implementation complexity'. The study is based on real implementation within the POSTGRES system which makes it more reliable. Another work dedicated to the comparison of several access methods is [KSSS89]. The paper contains two separate parts, one for point access methods (objects of zero size) and one for spatial access methods. In sum, eight indexing structures were implemented and evaluated. The paper also departs from the uniformly distributed data assumption (so prevalent in the literature) and uses skewed data trying several probability distributions beside real data.

Finally, the third reason behind our interest in performance studies is to survey the major performance parameters. In simpler words, we want to see what it takes to be a good access method. Among the parameters one has to keep an eye on when designing, analyzing or assessing an access method we list the following:

1. Disk storage requirements of the resulting indexed data.
2. Primary and secondary (if index nodes may reside on the disk) memory requirements of the indexing nodes (or index's directory).
3. The ratio of directory pages to data pages.
4. The height of the index.
5. The time needed to construct the index (especially if it has to be reconstructed periodically).
6. The auxiliary temporary memory needed to construct the index.
7. The disk access cost of insertions, deletions, and updates.
8. Point and range search costs (in terms of number of disk accesses).
9. Average memory/storage utilization (a figure of 70% is considered acceptable) [Sal94].

10. The cost of any auxiliary maintenance operations necessary for the proper functioning of the index.

11. Performance under non-uniform (even ugly) data distributions.

12. Scalability to larger database sizes.

13. Existence of opportunities to exploit parallelism and concurrency on both the index nodes (or directory) and data nodes.

14. Ease of implementation and simplicity of associated algorithms.

However, we do not expect any single access method to score positively on all these criteria. As Zobel et al. note 'no indexing scheme is all powerful' [ZMR96].

3.5 Mobile Data Management

The field of mobile data management represents the larger context into which our research problem fits. Recent technological developments that made portable computers more powerful and cheap started up the new field of mobile computing. The advent of palmtop computers capable of running a number of popular applications led to the speculation that such devices will be ubiquitous in the near future. Indeed, Dunham and Helal [DH95] use the year 2005 in a fictitious example comparing a transaction in the present with its counterpart 'in the mobile world of 2005.' On the power of recent laptop computers, Alonso and Korth [AK93] remark that "laptops have become capacious enough to hold databases that would have been called 'very large' not too long ago, and fast enough to support complex database operations". Mobility has thus stirred up the research community to deal with new challenging problems.

We might safely say that both mobile computing and mobile data management have just been born as research fields. Satyanarayanan, in a 1996 paper entitled 'Fundamental Challenges in Mobile Computing' [Sat96] attempts to answer the question 'what is unique and conceptually different about mobile computing?' and provides 'fertile topics for exploration'. This shows that the

field of mobile computing is still trying to define its problems. As for the field of mobile data management, it recently witnessed a series of papers all of which attempting to answer basically the same question, namely, what new research problems does mobility bring about into the field of data management? [IB93b, IB93a, IB94, DH95, AK93]. A cursory survey of activity in the field leads us to suggest that the year 1993 is the 'birth date' of mobile data management. There has been very faint activity (to the best of our knowledge) in 1992 (but see [IB92]). In this latter paper, Imielinski and Badrinath forward the following remark in their section on future work: '... the research work described in this paper is to our knowledge, one of the few efforts that is addressing issues of information access in a mobile distributed environment'. In a 1993 paper [Wol93], Wolfson describes a new project in which the problem of efficiently allocating and deallocating copies of data items residing in databases to mobile computers is investigated.

In general, major areas of impact of mobility include query processing, transaction processing, location management, replication management, handling disconnection, coping with scale, and security. There is ongoing research and preliminary solutions have been proposed in most of these areas. However, we will not attempt to present them here. [OU97] explores the impact of mobility on real-time mobile data management.

Finally, we mention a narrower field which fits more as a context for our work; this is the area of *vehicle navigation systems* [Ege93]. It seems to combine all of the fields mentioned above so that we need spatial, mobile, and real-time data management. It is directly related to our research problem as we are basically indexing data that is rapidly changing over time. A relevant work in this area is that of Shekhar and Yang [SY91] where indexing is also considered in the context of intelligent vehicle navigation systems. Their index is called *MoBiLe file* and maps the two-dimensional space of motion to the disk tracks and sectors while attempting to preserve proximity relationships. This map requires a mapping function and knowledge about the population distribution. an object's geographical location is then used as the primary key to locate the disk block where it resides. However, since they do not make use of a motion equation, the nature of their work is different from ours.

3.6 Computational Geometry

Another related area is the field of computational geometry [PS85]. The dynamic range search is the field that is most relevant to our work [Mul94]. However, the methods of computational geometers have mostly been theoretical putting an emphasis on asymptotic analysis rather than more fine-grained analyses required by a practitioner interested in real performance issues. Moreover, the data set is always assumed to be in memory and secondary storage issues are not considered. A more detailed exposition of these views appears in [Tam96]. The authors conclude the paper with recommendations for the computational geometry community to provide practical tools and techniques together with its contributions to the understanding and solving of basic geometric problems. Among the data structures developed in computational geometry we mention the range tree and the segment tree [PS85]. In general, these data structures and the algorithms developed as solutions for the dynamic range search problem might prove useful in our research. We feel however that some effort is needed to bridge the gap between the wealth of theoretical results of computational geometers and our concerns about the design of a secondary storage indexing structure. As such, a future research question is whether we could transform our dynamic attribute indexing problem into a more abstract version for which ingenious solutions already exist in computational geometry. The literature of the field should at least serve as a source of inspiration.

Chapter 4

THE QUADTREE METHOD

This chapter provides detailed experimental and analytic studies about the quadtree approach to dynamic attribute indexing. We start by describing the general simulation model common to both the quadtree and cross points approaches in Section 4.1. We then study the data structures and programs specific to the quadtree simulation experiments in Section 4.2. In Section 4.3, we review the storage requirements of the quadtree based index and present the relevent experimental results. In Section 4.4, we provide a mathematical analysis of the average percentage space utilization of the quadtree in the context of our specific application. In Section 4.5, we describe the cost involved in the regeneration of the index which takes place periodically; this is called build cost. We then contribute an important improvement over the naive approach to index reconstruction; this is an optimal algorithm described in detail in Section 4.6. Finally, we mention query processing performance of the method in Section 4.7.

4.1 The Simulation Model

The general model is roughly common to both the quadtree and cross points approaches and consists of a few parameters intrinsic to the performance study plus a set of workload parameters. The former set of parameters include N,

the number of objects in the system which serves to test the scalability of each method and the maximum number of objects it can handle within reasonable performance constraints. Given N, we generate randomly the corresponding N linear equations describing the way attributes change over time. This is done by generating the intercepts b randomly and uniformly distributed over the attribute space $[A_{min} \ldots A_{max}]$. We then generate the values of the slopes a in the range determined by how fast we would like our objects' attribute values to change (on the average). They also include ΔT, a period of time which we call *quadtree regeneration period* in the quadtree method and the *lookahead interval* in the cross points method. In the former method, the quadtree is destroyed and rebuilt every ΔT time units so that its rebuilding at time t_{now} means that it will be used to index objects only until time $t_{now} + \Delta T$. In the cross-points method, the set of cross points that may be stored is limited by space constraints and hence only intersections that fall in the time interval $[t_{now} \ldots t_{now} + \Delta T]$ are computed at time t_{now}.

The average speed of the objects moving in the system is also important; let this be denoted by $\overline{v}$. Of more value however is how big is $\overline{v}$ compared to the total indexed distance which we denote by ΔA. We capture this observation in a model parameter which we call *speed ratio* defined as follows.

Definition 2 *The* **speed ratio** α *of a system is the ratio of the average speed of objects divided by the total length of the indexed unidimensional space. It can also be expressed as the fraction of distance an object moves in a single time unit relative to the total distance. The formula is:*

$$\alpha = \frac{\overline{v}}{\Delta A}$$

This parameter is an intrinsic property of the system of objects and for this matter we will even assume it given. Conceptually, α is an indication of the dynamism of our system; the higher it is the more agitated the objects are while the lower it is the more sluggish the overall system becomes.

To evaluate query processing performance, we use the range size of queries as a parameter since we are mainly interested in range searches. Here too, it is the relative value of the range with respect to ΔA that is significant rather

than its absolute range length (which we shall denote by ΔR). The number of pages in the buffer BF (*buffer size*) is also an important factor when disk access comes into play. Note that data are disk-resident only in the quadtree approach while in the cross-points method, both the binary tree and its associated set of intersections are memory-resident. Finally, there is the so-called *merge threshold* used only in the quadtree approach. It is needed in its delete function where we have to check whether the number of elements in the four sibling leaf nodes has fallen below a certain value to justify merging them into one leaf node (which shrinks four disk pages into one). The merge threshold is the name given to this value. In fact, we used one single threshold value throughout the simulations that was determined to be optimal by experimentation.

The second set of parameters is related to workload and query processing. The former consists of the number of insertion, deletion, and update requests expected every ΔT time units. These result in changes in the contents of the indexing structure and are characteristic of the nature of both the application domain and the (resultant) indexing problem to be solved. The main query processing parameters are the number of instantaneous and continuous queries every ΔT time units. Note that a continuous query arriving at some time point in the interval $[t_i \ldots t_i + \Delta T]$ should also be considered at later intervals $[t_j \ldots t_j + \Delta T]$ ($t_j = t_i + n\Delta T$ for some $n > 0$) until explicitly cancelled. The parameters we have described are summarized in Table 4.1. Next, we describe the simulation programs.

4.2 Programs and Data Structures

Since we study two techniques, we have two main driver programs for our simulation, one for the quadtree approach and the second for the cross points approach implemented using a binary tree. Besides, we isolate all the code necessary for generating workload into a separate file and all the functions responsible for buffer management into yet another file making up for roughly four distinguishable program components. The programs were written in C (approximately 3000 lines of code). The experiments were run on Sun Sparc

N	Number of objects in the system
ΔT	Regeneration or looka-head period
α	Speed ratio
ΔR	Range size of queries
BF	Buffer size
NIQ	Number of instantaneous queries every ΔT
NCQ	Number of continuous queries every ΔT
INS	Number of insert requests every ΔT
DEL	Number of delete requests every ΔT
UPD	Number of update requests every ΔT

Table 4.1: Simulation model parameters.

workstations running the Solaris operating system. In this section, we describe quadtree-related data structures. Description of data structures for the cross points method is given in the next chapter dedicated to it. We use the notation $R =< Field_1, \ldots, Field_n >$ for record structures.

First we describe page-related data structures. In the quadtree approach, we model the disk as an array of pages and represent pages as a collection (again array) of objects plus a field indicating the number of elements in the page. Using symbols, we write $D =< p_1, \ldots, p_{N_D} >$ to denote that a disk D is a vector of N_D pages, where each page $p_i =< N_i, E_i >$ is a record with a number of elements field N_i and the elements themselves $E_i =< e_1, \ldots, e_{N_i} >$ stored in an array (or vector). Similarly, $BF =< p_1, \ldots, p_{BF} >$ means that our buffer is modeled as a vector of BF buffer pages each having the same record structure as a disk page. In our experiments, N_D was fixed to the maximum that we expect will be needed and is also limited by the machine's main memory capacity while BF is the model parameter described above.

We manage the buffer using the *Least Recently Used* (LRU) page replacement policy. For this we have a buffer manager data structure $BM = < ps_1, \ldots, ps_{BF} >$ where $ps_i =< dp_i, lru_i >$ records the page status of the i^{th} page of the buffer. This page status is composed of the number of the disk page currently residing in $BF[i]$ and the LRU stamp lru_i with which it was brought into the buffer. The LRU policy is naively implemented using a global counter incremented each time a new page is fetched from the disk and brought into the buffer,

then its value is assigned to lru_i if the page is assigned to $BF[i]$. The main functions implemented in our buffer manager program include functions that transfer a page from disk to buffer (page fetch) and from buffer to disk (page flush) besides a set of functions for freeing a given number of buffer pages (at most 4 in our application) given the constraint that one or more designated pages must remain there (even if their LRU order has come). The latter are needed in insertion and deletion functions of the quadtree. Next, we see in-memory data structures.

The simplest such structure is the *object*. The record is $O =< ID, a, b >$ where ID is the object ID and a and b are the slope and intercept (respectively) of the object's trajectory described by the linear function $f(t) = at+b$. Objects are the elements e_i mentioned above that occupy disk (or buffer) pages. The major data structure in the quadtree approach is the *quadnode* upon which the quadtree implementation is based. A quadnode record QN was described earlier and is defined as follows:

$$QN =< X_{min}, Y_{min}, X_{max}, Y_{max}, N_{QN}, Type, PN, SW, NW, NE, SE, Par >$$

The fields X_{min}, Y_{min}, X_{max}, and Y_{max} define the boundaries of the rectangular space being indexed by the quadtree rooted at QN. Field N_{QN} stores the number of index points indexed by the quadtree rooted at QN. The $Type$ field designates the type of the node which can be *leaf* ($Type = 0$), *next-to-leaf* ($Type = 1$), or *internal* ($Type = 2$). Differentiation between leaf and next-to-leaf is crucial to all search, insert, update, and delete operations while the need to distinguish between next-to-leaf and internal nodes arises in the context of delete and merge operations. Field PN stands for page number and is 'active' when QN is a leaf node in which case it gives the number of the disk page containing the actual indexed elements. When QN is not a leaf node, the field PN is simply not used. The SW, NW, NE, and SE fields are pointers to the four subquadrants of QN which are respectively the south-west, north-west, north-east, and south-east subquadrants. These fields take the $NULL$ value when QN is a leaf node and point to actual quadnode records otherwise. Finally, Par is also a pointer that points to the parent node of QN and is required by the deletion and merge functions.

4.3 Storage Requirements

The main parameters we are interested to measure before looking into query processing performance are the number of disk accesses required to build the quadtree every ΔT time units (henceforth denoted by TU) and the storage requirements of the tree. Figures 4.1 and 4.2 show disk consumption as a function of the number of objects N being indexed (also called *system size*). Note that both graphs have the same pattern in which the number of disk

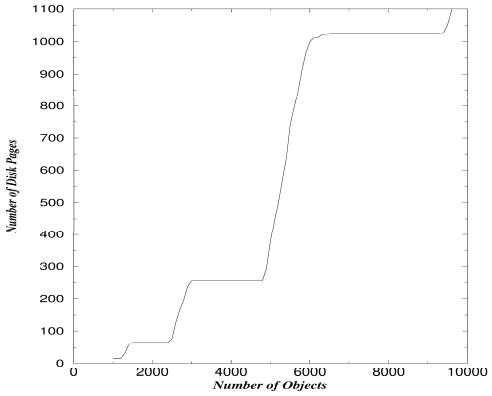

Figure 4.1: Disk consumption for small system sizes.

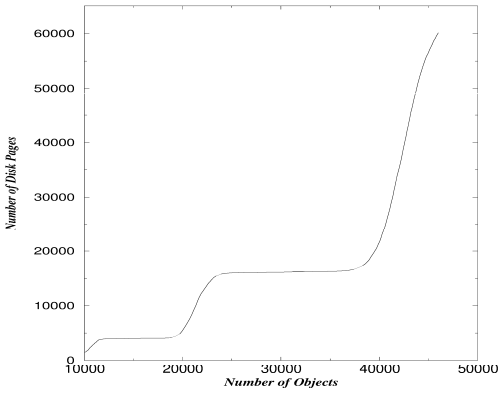

Figure 4.2: Disk consumption for large system sizes.

pages remains fixed for a while then increases rapidly over a small interval of objects count N until it reaches some new plateau at which it remains fixed for an even longer interval of N. The notion or pattern of plateau is so prevalent in other experiments and almost intrinsic to the nature of our application that it deserves a precise definition so that we can make frequent use of the term later.

Definition 3 *A* **plateau** *is an interval* $[N_i \ldots N_{i+L})$ *in the number of objects over which the resulting quadtree requires exactly the same number of disk pages. We say* L *is the* **plateau length**.

If L is very small, we will not really have a 'plateau' in the graph and the definition would thus not be very faithful to the term selected. However, for the moment let us assume that L is usually relatively big and we shall later go deeper into the anatomy of plateaus in our specific application. To continue

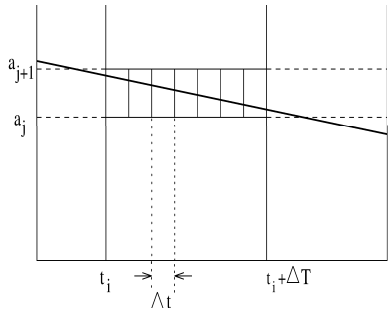

Figure 4.3: The trajectory spends the whole session in the same attribute interval $[a_j \ldots a_{j+1}]$ hence causing object copies to be redundantly stored in every time interval (of width Δt in the figure).

the description, we see that plateaus occur at values of 16, 64, 256, 1024, 4096, and 16384 disk pages. The last plateau is not reached by the graph and occurs at 65552 and is the disk consumption when N reaches 50000 (experiments stop at $N = 46000$). The number of disk pages in a plateau is thus four times that of the previous plateau.

The quadtree starts with a single page and every disk page in our application has a capacity $B = 340$ index elements (4096 bytes divided by 12 bytes, the size of an object record). When filled, the single page is *split* into four and the $B + 1$ objects are redistributed over the four subquadrants generating N' index elements satisfying $B < N' \leq 3(B + 1)$ as each trajectory may cross at most three subquadrants. The peculiarity of our application lies in the fact that almost all splits generate exactly two index elements and these fall either in the northern (NW and NE) or the southern (SW and SE) subquadrants.

To explain why this is so, we first define a *session* to be the time span between two consecutive reconstructions of the quadtree. Thus a session lasts ΔT TUs and we let the i^{th} session (call it S_i) begin at time $t_i = i\Delta T$. Note also that the time span of a subquadrant in our quadtree is halved upon splitting which also divides the attribute interval indexed into two. When the number of objects is relatively large our original two-dimensional space becomes very finely partitioned so that the time interval indexed by any single leaf subquadrant becomes too short for most objects to 'cross' the attribute boundary. This is especially so when the speed ratio α of the system is low.

Figure 4.3 reveals the redundancy (though in an exaggerated fashion). Assume that for a given session S_i and a given number of objects N the attribute space is divided into p intervals and the time space into q intervals. Figure 4.3 illustrates the situation where an arbitrary object o's position y_o satisfies $a_j \leq y_o \leq a_{j+1}$ during the session span $[t_i \ldots t_i + \Delta T)$. In other words, the trajectory passes through only one single attribute interval (here denoted by $[a_j \ldots a_{j+1}]$) over the whole session. Corresponding to this attribute interval, object o will have as many copies as there are subintervals of $[t_i \ldots t_i + \Delta T)$, that is q. The q copies are redundant in the sense that they convey the same information and could thus be replaced by one. Figure 4.3 is an extreme case and in practice an object may cross two or more attribute intervals in a single session. How many such attribute intervals an object is expected to cross in a single session depends on the speed ratio α and the granularity of partitioning of the indexed space (in other words N). Low values of α and a very fine partitioning lead to much redundancy of the type depicted in Figure 4.3. On the other hand, a high α and a coarse partition of space reduces redundancy as trajectories are more likely to cross many attribute intervals in a single session. Let us quantify with more precision the average number of copies of a single object.

To state the question in a different way, we would like to count the number of subquadrants a given trajectory crosses in a given partitioned (two-dimensional) space. For this, we denote by m the average number of attribute intervals (or slices) that a trajectory with a slope $\overline{v}$ spans in a time interval ΔT. Let us further assume that the attribute space is subdivided into equal-sized intervals each of length δa. Then using the motion equation $f(t) = \overline{v}t + b$ defining the trajectory, we know the object will move a distance $\overline{v}\Delta t$ along the attribute dimension. m can then take one of two values given in the form of the following inequality.

$$\left\lfloor \frac{\overline{v}\Delta T}{\delta a} \right\rfloor \leq m \leq \left\lceil \frac{\overline{v}\Delta T}{\delta a} \right\rceil \qquad (1)$$

Remember that m is a natural number and it takes one of the two consecutive values given in inequality 1 depending upon where the object starts and finishes inside the attribute slices at the beginning t_i and end $t_i + \Delta T$ of the session. Once we know m, we need also to fix the number of time slices into which our

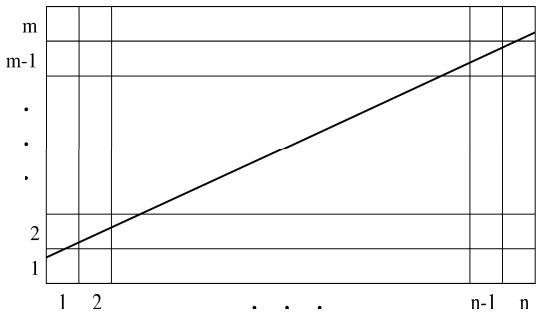

Figure 4.4: A line crossing an $n \times m$ grid.

session is partitioned; we will denote this by n. At this point we know that our object's trajectory spans n time slices and m attribute slices. A more abstract version of the problem is then the following. Given an $n \times m$ grid of squares and a line crossing it from the square $(1,1)$ to the square (n,m), how many squares $C_{n,m}$ does the line pass through? This is depicted in Figure 4.4. Note that in any single column (which corresponds to a time slice in our application) the trajectory can cross one or two squares depending on whether or not it crosses one of the horizontal boundaries of that column. Since there are $m-1$ such boundaries to be crossed in m squares, and since each such crossing adds an extra square, the total number of squares $C_{n,m}$ crossed is given by the simple formula:

$$C_{n,m} = n + m - 1 \tag{2}$$

Thus when an object moves across m attribute slices in a session partitioned into n time slices it will incur $n+m-1$ copies. However, this is a maximum and we may subtract one for every boundary which the trajectory crosses at a corner (the point of intersection of four squares). Generally, we expect this to be rare and it will anyway not make a big difference. We remark that n is the lower bound for $C_{n,m}$ and occurs when the trajectory is a perfect diagonal of the rectangular grid. We have gone through this analysis to give an idea and a feeling of where the high space requirements of the quadtree come from.

Coming back to our specific application, we remark as we said above that our objects tend to spend the whole session in a very small number of attribute intervals; in other words m is small. This leads to the fact that upon splitting very few objects will generate one or three copies while most of them generate exactly two copies that reside either in the upper or lower halves of the parent quadrant. Even in the case where a trajectory crosses three subquadrants of

a given parent quadrant, it is likely that it will cross only one subquadrant of a nearby quadrant hence averaging to two. The result is that in most cases we have a next-to-perfect duplication factor of two in a local split of a data bucket. This turns out to have a relation with plateaus which we reexamine at this point.

We have earlier stated that the number of disk pages required by the quadtree reaches plateaus at the values of 16, 64, 256, 1024, 4096, 16384, and 65552. It is not difficult to see that plateaus occur at values of 4^i ($i > 0$). The question is then why do we have plateaus or why does the number of disk pages stabilize for a while at values of 4^i? The reason is that all subquadrants tend to fill up at the same rate and reach capacity B at the same time. This leads to a wave of splits across all quadrants of the quadtree that is triggered by a relatively small number of newly inserted objects. Furthermore, insertion of a single object sometimes triggers splits in all (or a big portion of) quadrants that correspond to a single attribute interval (it splits a whole row). After these splits are over, it will take quite some time (i.e., many insertions) before the new disk pages are filled again. During these insertions, disk pages are getting nearer to full capacity but no new disk pages are being required by the quadtree. For this reason, we have a plateau shape followed by a sharp rise that only ends at the next plateau. The number 4^i comes from the fact that the quadtree starts with one (4^0) disk page and every split wave multiplies the number of disk pages by four. Notice also that $4^i = 2^i \times 2^i$ so that in a plateau of 4^i, we actually have our original index space partitioned into a $2^i \times 2^i$ grid of quadrants. The ubiquitousness of this quadtree configuration in our application and the alternating pattern of plateaus and splits compel us to introduce the following term which we will need later.

Definition 4 *An i^{th}-**regular quadtree** is one which partitions the underlying indexed space into a $2^i \times 2^i$ grid of quadrants. We also say that it is at the i^{th}* **plateau**.

This leads (among other things) to a prohibitively large number of index points illustrated in Figure 4.5 for a system size above 10000. Notice that we reach one million index points at $N = 15000$ and as much as 11 million index points

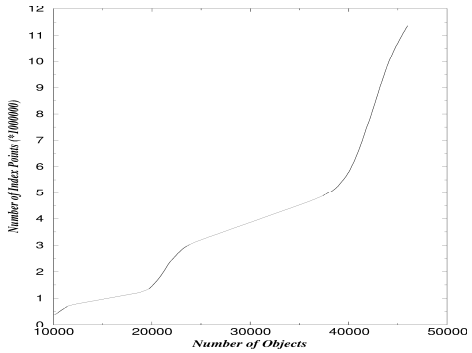

Figure 4.5: Number of index points.

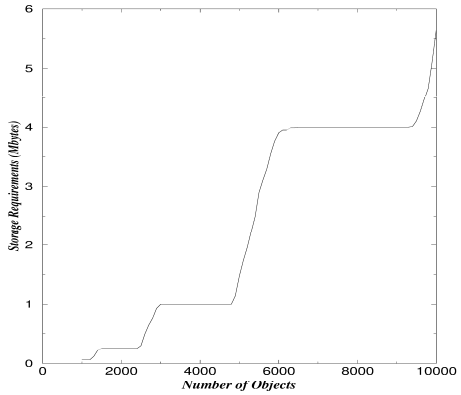

Figure 4.6: Storage requirements for small system sizes.

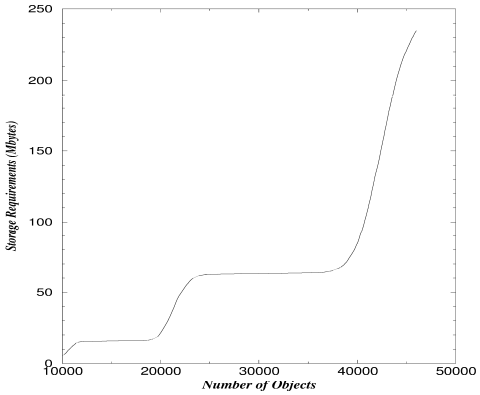

Figure 4.7: Storage requirements for large system sizes.

at $N = 45000$. This huge number of index points requires substantial memory. We have translated the number of pages parameter in earlier graphs into the corresponding memory requirements in megabytes (4 Kbytes/page) and the result is given in Figures 4.6 and 4.7. Here again, we have the plateau pattern with a plateau at 1 Mbyte for the (number of objects) interval $[3000 \ldots 5000]$, the next at 4 Mbytes for the interval $[6000 \ldots 9500]$, the next at 16 Mbytes for the interval $[12000 \ldots 20000]$, and the plateau at 64 Mbytes for the interval $[23000 \ldots 38000]$. We also have a plateau at 256 Mbytes that starts beyond the point $N = 50000$. We think this space overhead to be impractical especially that we may have several attributes. The main reason for this is the redundancy or number of copies which becomes higher and higher as the number of objects in the system increases. To understand this better, we introduce a measure of redundancy.

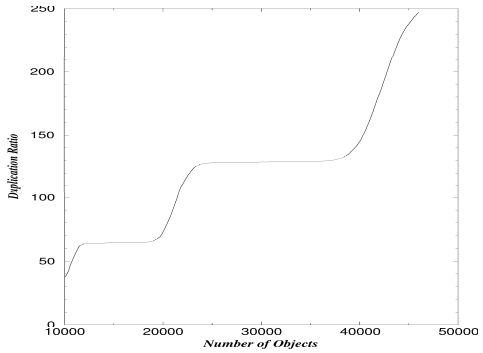

Figure 4.8: Duplication ratio.

Definition 5 *The* **Duplication Ratio** D *of a quadtree is the average number of copies that a single object has in the quadtree:*

$$D = \frac{\text{Number of Index Points}}{\text{Number of Objects}}$$

The duplication ratio is given in Figure 4.8 as a function of N. Again the plateau pattern prevails but this time plateaus occur at powers of two (2^i : $i > 0$) rather than four so that we also have plateaus at $D = 8$ and $D = 32$ (not included in the figure). The figure shows a plateau at $D = 64$, then at $D = 128$ (not a power of 4), then at $D = 256$ for $N \approx 50000$. We observe that D corresponds to the number of quadrants on each side of the indexed space which we denote by $[A_{min} \ldots A_{max}], [t_i \ldots t_i + \Delta T]$. At plateaus, D is a perfect power of two and is exactly equal to the number of quadrants on every side of the original quadrant. In between plateaus (or during split waves), the value of D is related to the number of quadrants on the border of the indexed quadrant that experienced a split; let us see how.

First call this number m and let $\Delta A = A_{max} - A_{min}$ be the length of the indexed attribute space. If we are in a split wave between the i^{th} and $i + 1^{st}$ plateau then $0 \leq m \leq 2^i$ so that when the wave ends we have 2^{i+1} quadrants on the space border. When m border quadrants split, they produce $2m$ border quadrants of length $\frac{1}{2^{i+1}}\Delta A$ and we still have $2^i - m$ quadrants of length $\frac{1}{2^i}\Delta A$. The total number of border quadrants at this point (after m^{th} border split but before $m + 1^{st}$ one) is thus $2m + (2^i - m) = 2^i + m$. We then conjecture that D could be approximated by this value so that if $D_{i,m}$ denotes the duplication ratio after the i^{th} plateau is over and m border quadrant splits took place then

$$D_{i,m} \approx 2^i + m$$

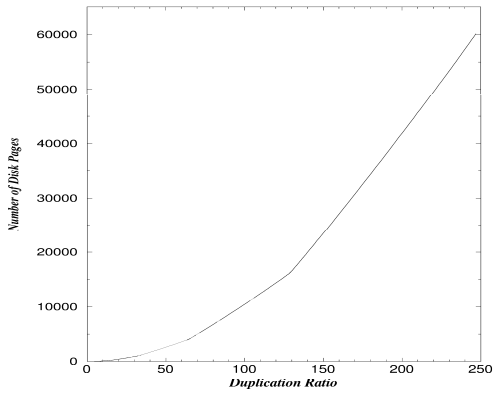

Figure 4.9: The relation between the number of disk pages and the duplication ratio.

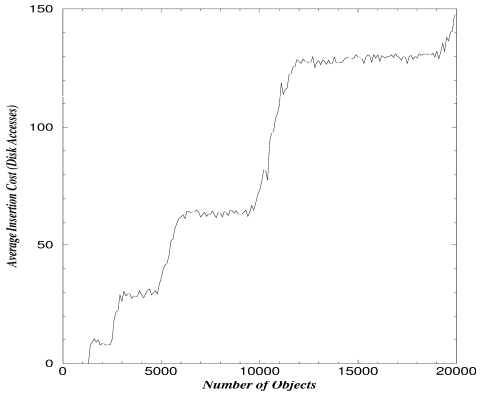

Figure 4.10: Insertion cost.

A simple experiment might prove its validity but we will stop considering this topic and retain that the redundancy as captured by the duplication ratio D is responsible for most of the disadvantages of the bucket PR quadtree indexing structure in the narrow context of our application.

Before we close the topic however, we have a final remark on the relation between D and the number of disk pages which is given in Figure 4.9. Recall that every disk page is pointed to by a leaf node of the quadtree residing in memory. Then, the number of disk pages is equal to the number of quadrants produced through insertions and splits. This, as the small digression above suggests, is roughly equal to the square of the number of border quadrants on a single side of the indexed space. Thus disk consumption is the square of D which is confirmed by Figure 4.9.

Among the performance parameters affected by the redundancy described above is the cost of insertions, deletions, and updates. If an object is duplicated q times then it resides in q disk pages and any of the above operations targeting it will have to access all those q pages. The same thing applies to the insertion of a new object. If in time span ΔT its trajectory crosses m attribute slices and the current quadtree is i^{th}-regular then its insertion cost will be $2^i + m - 1$ as analyzed earlier. The average number of disk accesses needed to insert an object is given in Figure 4.10 as a function of N. The value of the buffer size used was 32. Again we have plateaus at 2^i. This unfortunately means

that when our quadtree (by the effect of insertions and splits) passes from the i^{th} plateau to the $i+1^{st}$ plateau, the cost of insertion and deletion of an object which crosses m attribute slices (in ΔT TUs) passes from $2^i + m - 1$ to $2^{i+1} + 2m - 1$ hence effectively doubling. An examination of our graphs reveals that in the process N increases by only about 30%. The deterioration of performance is not only dependent on N but also disproportionate to its percentage increase. One more performance variable dramatically affected by redundancy is the cost of reconstructing the quadtree which we call *build cost.* Before we look into this problem we prefer to digress for a while on the efficiency of memory utilization by our quadtree.

4.4 Percentage Space Utilization

In this section, we are interested in computing and knowing about the space efficiency of our quadtree in the context of our particular application. The main characterizing features of our application have been mentioned previously and include the fact that upon quadrant split each object generates two copies and usually resides in the upper or lower half of the split quadrant. We will also assume that a trajectory falls in a single attribute interval $[a_j \ldots a_{j+1})$. We will attempt a mathematical analysis of utilization in which we seek a formula for the average space efficiency of our quadtree. We begin by establishing the necessary terminology.

We say that an i^{th} plateau is *full* if all the $2^i \times 2^i = 2^{2i}$ disk pages are full (contain B index elements). Furthermore, when the first quadrant in a full i^{th} plateau splits we say the i^{th} plateau *breaks* and that the i^{th} split wave *begins.* The i^{th} split wave *ends* when the last quadrant of the i^{th}-regular quadtree that has not yet split undergoes a split yielding the $i+1^{st}$-regular quadtree and initiating the $i+1^{st}$ plateau. Let P_i denote the number of index elements at the i^{th} full plateau; then $P_i = 2^{2i}B$ where B is as usual, the bucket size. Let P_i^{sw} denote the number of index elements immediately after the i^{th} split wave ends. We will perform our analysis based on a scenario of a shortest split wave, that is one which ends with the minimum number of insertions. This wave is as follows. We start with a full i^{th} plateau containing 2^{2i} quadrants

and 2^i attribute intervals. As such, and based on our assumption of a single trajectory falling in a single attribute interval $[a_j \dots a_{j+1}]$, we need exactly 2^i insertions to begin and end the i^{th} split wave. Each such insertion causes 2^i splits along the time axis which transforms a 1×2^i row into a $2 \times 2^{i+1}$ grid. All these assumptions are inspired by the real behavior of splits and trajectories in our application and a different split speed may not affect the average space efficiency which we are seeking to compute. Let $P(N)$ be equal to the number of index points in the quadtree which indexes N objects and $D(N)$ be the number of disk pages required to store a quadtree which indexes N objects. We use a classical definition of utilization.

Definition 6 *The* **Utilization Ratio** U *is the fraction of memory used by the index elements in a given quadtree relative to the total memory capacity of the disk pages consumed by that quadtree:*

$$U(N) = \frac{P(N)}{D(N)B} \tag{3}$$

The notation $U(N)$ is for utilization ratio in a quadtree of N objects. We are interested in the *mean utilization ratio* $\overline{U}$ which we define using the formula

$$\overline{U} = \lim_{N \to \infty} \frac{\sum_{p=1}^{N} U(p)}{N} \tag{4}$$

However this being theoretical, we need to compute the mean over a finite range of values of N. We choose the range of values between two consecutive plateaus as our finite range. Let N_i denote the number of objects at an i^{th} full plateau of a quadtree. We define the i^{th} *mean utilization ratio* $\overline{U_i}$ as follows.

$$\overline{U_i} = \frac{\sum_{p=N_i}^{N_{i+1}} U(p)}{N_{i+1} - N_i} \tag{5}$$

Let us express formula 5 in words. Starting from the point when the i^{th} plateau breaks, we insert objects consecutively until we reach the full $i + 1^{st}$ plateau and compute the utilization after every insertion (this is the term $U(p)$ in the summation). We then calculate $\overline{U_i}$ as the arithmetic mean of those intermediate values. Let us then compute P_i^{sw}.

Recall that P_i^{sw} was defined as the number of index points immediately after the i^{th} split wave ends. At this point, we have the previous P_i index points

each generated two copies during the split waves and the newly inserted 2^i objects (which trigger 2^i row splits) each yielding 2^{i+1} copies (2^{i+1} is the new side length of the indexed space). We then have $P_i^{sw} = 2P_i + 2^i 2^{i+1}$ which after replacing P_i by its values of $2^{2i}B$ simplifies into the formula

$$P_i^{sw} = 2^{2i+1}(B+1) \tag{6}$$

We can then compute the disk utilization immediately after the i^{th} split ends which we denote by U_i^{sw}. Using Equation 3, in which $P(N) = P_i^{sw}$ and $D(N) = 2^{i+1} \times 2^{i+1} = 2^{2i+2}$ and replacing we find

$$U_i^{sw} = \frac{B+1}{2B} \approx \frac{1}{2} \tag{7}$$

The utility of this formula lies in the fact that it corresponds to the minimal space utilization of the quadtree in our application. This is so because any insertion beyond this point will increase the occupancy of the $2^{i+1} \times 2^{i+1}$ quadrants (whose number remains fixed) and this continues until we reach the full $i+1^{st}$ plateau. Thus, it tells us that the worst case percentage utilization of our quadtree is 50%. This is not very comforting though and we now pass to the more serious task of computing the i^{th} mean utilization ratio.

We start by computing the number of insertions (new objects) needed to pass from the i^{th} full plateau to the $i+1^{st}$ full plateau which we denote by ΔN_i. We know $\Delta N_i = N_{i+1} - N_i$ (by definition). Note that ΔN_i is the sum of two values. The first is 2^i, the number of insertions needed to end the split wave and the second is the number of insertions needed to bring the newly formed $i+1^{st}$-regular quadtree to the status of a full $i+1^{st}$ plateau. These two phases will also govern our subsequent analysis and we need to give them names. Let us call the first one the $SPLIT$ phase and the second one the $FILL$ phase. At a full $i+1^{st}$ plateau, the duplication ratio is 2^{i+1} (side length of the grid). Hence the number of objects needed in the $FILL$ phase may be computed from the number of index points generated during this phase divided by the duplication ratio. This results in the following formula for ΔN_i.

$$\Delta N_i = 2^i + \frac{P_{i+1} - P_i^{sw}}{2^{i+1}} \tag{8}$$

Stated in words, the difference between the number of index points at the full $i+1^{st}$ plateau and that just after the split wave ended divided by the

duplication ratio yields the number of objects that were inserted in the $FILL$ phase. Simplifying Equation 8 gives the cleaner formula

$$\Delta N_i = 2^i B \tag{9}$$

As there are 2^i insertions in the $SPLIT$ phase, the number of insertions in the $FILL$ phase is $\Delta N_i - 2^i = 2^i(B-1)$. Now, let $U^i_{SPLIT}(m)$ denote the utilization ratio when m objects have been inserted in the $SPLIT$ phase; m satisfies the constraint $0 < m \leq 2^i$. By analogy, we define $U^i_{FILL}(m)$ as the utilization ratio when m objects have been inserted and we are in the $FILL$ phase; that is m satisfies the constraint $2^i < m \leq \Delta N_i$. These are exactly the component utilizations we talked about earlier that we want to sum and divide by $N_{i+1} - N_i$ to obtain $\overline{U_i}$ (see Equation 5). A more precise expression for $\overline{U_i}$ is then as follows.

$$\overline{U_i} = \frac{\sum_{0<m\leq 2^i} U^i_{SPLIT}(m) + \sum_{2^i<m\leq 2^i B} U^i_{FILL}(m)}{\Delta N_i} \tag{10}$$

It now suffices to find formulas for $U^i_{SPLIT}(m)$ and $U^i_{FILL}(m)$. To save space, we will not give the details of the derivations involved. Below is the formula for $U^i_{SPLIT}(m)$ in terms of i, m, and B.

$$U^i_{SPLIT}(m) = 1 - \left(\frac{m}{2^{i+1}}\right)\left(\frac{B-1}{B}\right) \tag{11}$$

The reader may verify for the special case $m = 2^i$, utilization equals $\frac{B+1}{2B}$ which agrees with Equation 7 for U^{sw}_i derived in a different way. The formula for $U^i_{FILL}(m)$ is

$$U^i_{FILL}(m) = \frac{1}{2} + \frac{m}{2^{i+1}B} \tag{12}$$

Finally, using Equations 11, 12, and 9 then substituting them into Equation 10 and simplifying gives us

$$\overline{U_i} = \left(\frac{B-1}{B}\right)\left(\frac{3}{4} + \frac{5 - 4\left(\frac{2^i+1}{2^{i+2}}\right)}{4B}\right) \tag{13}$$

We now shall use a few approximations to simplify further the above formula and see better what it stands for. We approximate $\frac{B-1}{B}$ by 1 which we expect is acceptable in most applications. In our specific application $B = 340$ so that $\frac{B-1}{B} = 0.997$. We also denote by α_i the quantity $\frac{2^i+1}{2^{i+2}}$. We then have the following formula for $\overline{U_i}$.

$$\overline{U_i} \approx \frac{3}{4} + \frac{5 - 4\alpha_i}{4B} \tag{14}$$

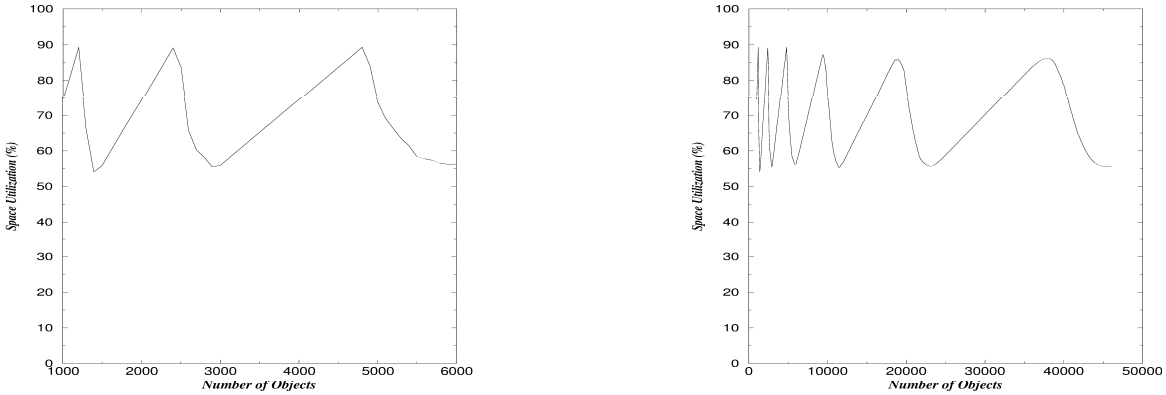

Figure 4.11: Percentage utilization.

To simplify the formula even further, notice that $\alpha_i = \frac{1}{4} + \frac{1}{2^{i+2}}$ so that $\lim_{i \to \infty} \alpha_i = \frac{1}{4}$. We may then approximate α_i by $\frac{1}{4}$ as i need not in fact be arbitrarily large (e.g., for $i > 3$, $\frac{1}{4} \leq \alpha_i < \frac{1}{4} + \frac{1}{64}$). In a sense, we are at this particular approximation removing the effect of selecting a certain i^{th} plateau when we started this analysis. In other words, any such i would produce roughly the same mean utilization and so we are compelled to drop the subscript i from $\overline{U_i}$. This leads to the more expressive and consize formula

$$\overline{U} \approx \frac{3}{4} + \frac{1}{B} \tag{15}$$

The final step is to neglect $\frac{1}{B}$ as well (0.002941 in our application) which leaves us with the elegant result

$$\overline{U} \approx \frac{3}{4} \tag{16}$$

The average percentage utilization in our particular application and use of the quadtree is therefore 75%. This is not surprising though as it is the midpoint between the maximum utilization of 100% (which occurs in a full plateau) and the minimum utilization of 50% proved earlier (see Equation 7) and which occurs at U_i^{sw}. The experimental evaluation of the percentage utilization as a function of the number of objects is given in Figure 4.11. It confirms the fact that minimum utilization does not drop below 50%. Maximum utilization however is always bounded by the 90% barrier. This is because in practice, the $i + 1^{st}$ split wave begins before the i^{th} plateau becomes full. In other terms, when the first quadrant split in a $2^i \times 2^i$ partitioned space takes place, there are still quadrants which are not yet full (i.e., contain less than B index elements).

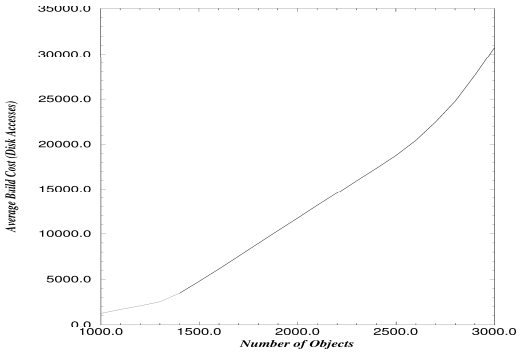

Figure 4.12: $BF = 8$ pages.

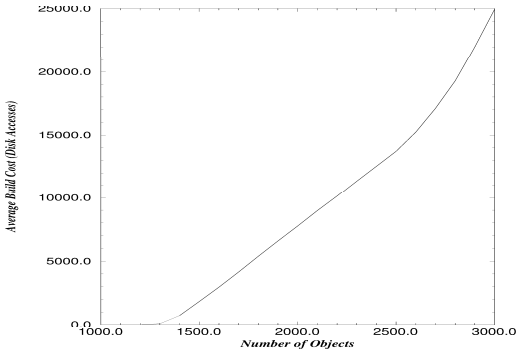

Figure 4.13: $BF = 16$ pages.

4.5 Build Cost

The cost of rebuilding the quadtree is more severely affected by an excessive amount of redundancy or object copies as it is based solely on insertion and has to undergo all the split waves that are within its range. We illustrate the average build cost as a function of the number of objects for different values of buffer size and a fixed $\Delta T = 100TUs$ in Figures 4.12, 4.13, 4.14, 4.15, 4.16, and 4.17. First, note that we conducted experiments to study the effect of ΔT on build cost. Interestingly enough, it transpires that it does not affect build cost. The regeneration period ΔT does not affect the number of index points created either; then it also will not affect the disk utilization of the quadtree (being directly related to index points set size). In one experiment, we fixed buffer size at 32 and N at 3500 varying ΔT from $100TUs$ to $10000TUs$ at steps of $100TUs$ obtaining a quasi-fixed average build cost of 32970. In a similar setting, we got a quasi-fixed number of index points around 56000. This may not be very surprising since the driving factor behind bucket overflows (and thus splits) is the number of objects while split and insertion overheads are what constitute build cost.

Does this mean that we could enjoy an infinite ΔT so that our quadtree is generated only once? Can we at least hope in practice to enjoy a 'long' period without reconstructing the quadtree? The answer is no. The reason lies in insertions and deletions. If all the objects which have existed in the index at some time t_{past} have left the system by time t_{now} and were all explicitly deleted (otherwise the index will grow without bounds) then we have practically regenerated the quadtree in the sense that we incurred the same cost. We thus

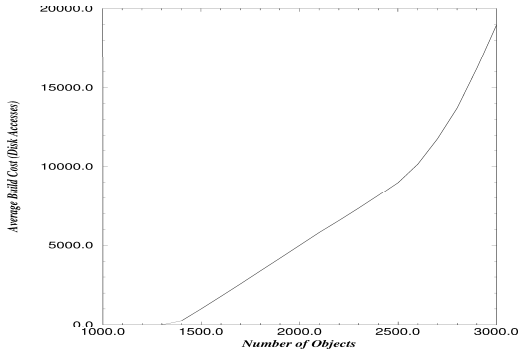

Figure 4.14: $BF = 32$ pages.

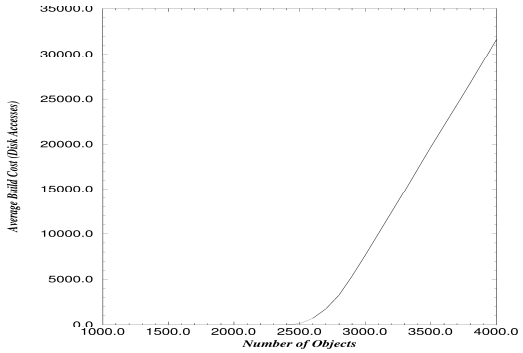

Figure 4.15: $BF = 64$ pages.

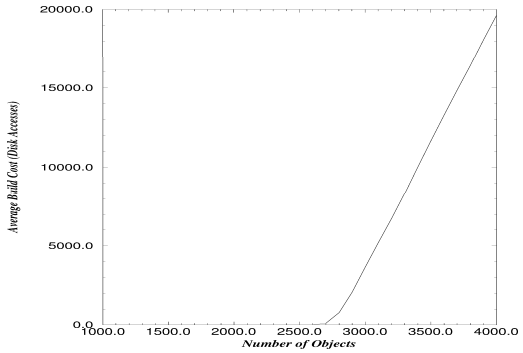

Figure 4.16: $BF = 128$ pages.

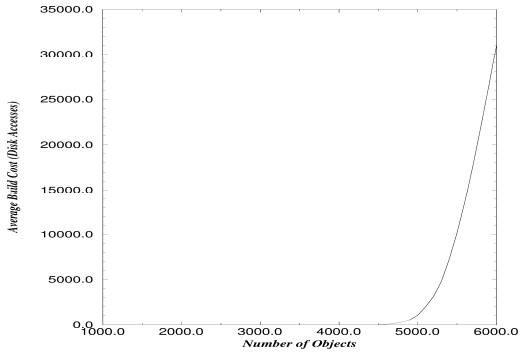

Figure 4.17: $BF = 256$ pages.

conjecture that a suitable ΔT depends on the average lifetime of the objects in the system which we will denote by $\overline{L}$. If ΔT is too small compared to $\overline{L}$ then we are (uselessly) incurring the high cost of regeneration while constructing essentially the same tree. On the other hand, if ΔT is much greater than $\overline{L}$ then we find ourselves effectively reconstructing the tree anyway as generations of objects come and go. Such observations compel us to conclude that ΔT's best value should be $\overline{L}$ (especially if the standard deviation of object lifetimes is small). We present this as a conclusion in the form of the following primitive equation where ΔT_{opt} stands for the optimal value of ΔT.

$$\Delta T_{opt} = \overline{L} \tag{17}$$

However, we do not in any way mean it to be a solution to its relevant optimization problem as it was derived through common sense rather than by analytic means. What is more important is that it is of no use if the build cost is prohibitive. A cursory look at the graphs illustrating build cost reveals values in the order of 10000 disk accesses for only 3000 objects. The buffer size provides little help in absorbing such a high cost. In fact, its only effect

as revealed by the graphs is to delay the 'explosion' of build cost by a few 100 objects. The graphs have an interesting pattern that is again related to the alternation of $SPLIT$ and $FILL$ phases in the construction of the quadtree. It is linear during $FILL$ phases. During the shorter $SPLIT$ phase it makes a small twist upward so that the next $FILL$ phase's graph is a steeper linear function. The reason for this high cost is that as we construct the quadtree from scratch, we have to bear all the costs of the consecutive split waves one after another hence effectively inserting every object many times. Besides, the CPU cost of all those insertions and reinsertions (at splits) is also unacceptable (it was the main reason why we did not try values of N beyond 6000, it simply takes hours). This unfortunately leads us to rule out the bucket PR quadtree as a choice to solve our problem as long as we have not devised a regeneration algorithm of tolerable cost. We can best characterize a tolerable (time) cost as one which is 'significantly' low compared to ΔT. The next section treats this problem in more depth and presents a solution which we think is optimal.

4.6 An Optimal Quadtree Regeneration Algorithm

Given the number of objects N in our system, we might know beforehand that we are to go through a certain number of costly split waves if we follow the naive approach of starting with a single bucket and making consecutive insertions. For example, in our application $B = 340$ and so we know that if $N > 340$ we will encounter a bucket split at object insertion operation number 341. This suggests that if we could know or predict right from the beginning (given only N) the final shape of our quadtree then we could just start with an empty quadtree of that shape and fill it properly with index points. In a sense, our earlier (naive) approach built the quadtree top-down; we are here proposing to build it bottom-up.

By the word 'shape' of the quadtree we actually meant its corresponding partition of the original indexed quadrant, and by 'final shape' we meant the final plateau at which the quadtree settles after N insertions. Let us call i in an

i^{th}-regular quadtree the *order* of that quadtree. The problem is then reduced to finding the order i given N which we explore in the next subsection.

4.6.1 Finding the Order of Quadtrees

We seek a formula for the order of a quadtree which we derive by simple analysis. Let us start with the base case. The condition $N > B$ tells us that we will have to split the bucket anyway. We then obtain $2N$ index points and 4 buckets of capacity $4B$. Notice that if $2N \leq 4B$ then the 2×2 quadtree is enough to store the N objects. The condition $2N > 4B$ similarly tells us that (if we went top-down) we will have to go through the second split wave anyway after which we have $4N$ index points (2 copies per object as assumed before) and 16 buckets of capacity $16B$. The next condition to evaluate is then $4N > 16B$. Generalizing this we find that the top-down insertion of N objects will reach the i^{th}-split wave if the condition $2^i N > 2^{2i} B$ is satisfied. By the same token, the $i + 1^{st}$ split will not be reached if $2^{i+1} N \leq 2^{2(i+1)} B$. Combining these two constraints, we obtain the following characterization of when N objects produce an i^{th}-order quadtree.

$$2^i B < N \leq 2^{i+1} B$$

The following formula for i is then readily computed.

$$i = \left\lfloor \log_2 \left(\frac{N}{B} \right) \right\rfloor \tag{18}$$

Remember that we assume that the i^{th} split wave breaks the i^{th} plateau only when it is full. We said earlier that this is not the case in practice where one or more of the $2^i \times 2^i$ buckets of the i^{th}-regular quadtree might become full and split before all 2^{2i} buckets become full. To remedy this we could simply add 1 to i hence constructing a larger grid than the theoretical prediction to cater for premature splitting (which is always the case in real applications). This would be a big squander of memory capacity if N were only slightly greater than $2^i B$ where we expect it is too early for splits of the next wave to begin. On the other hand it is reasonable to do so if N were too near to the value $2^{i+1} B$. In the former case we propose to use a $2^i \times 2^i$ grid and add overflow buckets for the (hopefully) few quadrants which are found to have overflowed. We may go

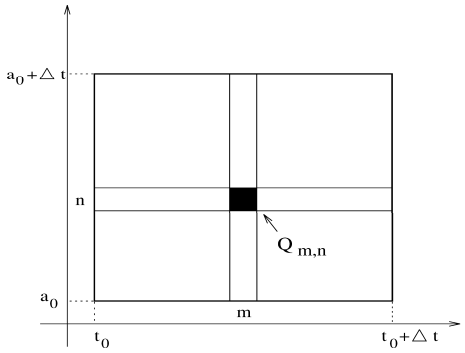

Figure 4.18: Definition of $Q_{m,n}$.

deeper into analyzing this tradeoff but we choose to stop here and come back to the main regeneration algorithm.

4.6.2 The Path Computation Algorithm

Before presenting the algorithm we introduce some necessary notation. In what follows we talk about an i^{th}-regular quadtree. We have earlier defined ΔA to be the length of the attribute dimension. Let $\delta t_i = \frac{\Delta T}{2^i}$ denote the length of each quadrant along the time axis and $\delta a_i = \frac{\Delta A}{2^i}$ denote the length of each quadrant along the attribute axis. Let our indexed space be $\mathcal{S} = [t_0 \ldots t_0 + \Delta T], [a_0 \ldots a_0 + \Delta A]$ and let

$$Q_{m,n} = [t_0 + m\delta t_i \ldots t_0 + (m+1)\delta t_i), [a_0 + n\delta a_i \ldots a_0 + (n+1)\delta a_i)(0 \leq m, n < 2^i)$$

designate the subquadrant of our space which lies at the intersection of the m^{th} time interval and n^{th} attribute interval as shown in Figure 4.18. Finally let $s = 2^i$ be the side length of the i^{th}-regular quadtree measured in number of intervals.

Since there will be 2^{2i} buckets in the final quadtree, we will need to fill and write 2^{2i} disk pages during reconstruction and this is then the minimum disk access cost which we can hope for. It would then be nice if we could shift all other auxiliary overhead into the CPU which is what we propose to do. The idea is to construct an in-memory $s \times s$ array (call it Q) which corresponds to the quadrants $Q_{m,n}$ of our indexed space defined above. We then compute for each of the N trajectories the coordinates (m and n: $0 \leq m, n < 2^i$) of

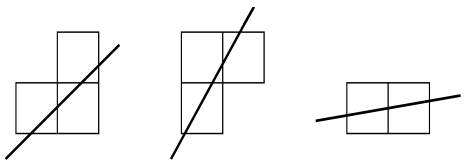

Figure 4.19: In two consecutive time intervals, a trajectory crosses either two or three quadrants.

quadrants it crosses and add the object information to every such quadrant. The entry $Q[m,n]$ of our array is thus a set defined as follows.

$$Q[m,n] = \{o_p : \text{trajectory of } o_p \text{ crosses } Q_{m,n}\}$$

We call the latter operation *object path computation.* Let us then provide a short description of the object path computation algorithm and see its complexity.

Given an object o_p, we examine the s time slices $[t_0+m\delta t_i \ldots t_0+(m+1)\delta t_i)$ $(0 \leq m < s)$ one by one in increasing order of m. Using the motion equation $f_p(t) = a_p t + b_p$, we compute at each time slice in which attribute slices object o_p's trajectory falls; there can be at most two of them. Figure 4.19 shows this and illustrates the important observation that for two consecutive time intervals there can be at most three quadrants through which the trajectory passes. There are $s = 2^i$ time slices and 2^{i-1} consecutive 'double slices'; then an object can have at most $3 \times 2^{i-1}$ index points (and at least 2^i) which is our upper bound. The path computation algorithm is thus $\mathcal{O}(3 \times 2^{i-1})$. Since 2^i is the minimum it is more accurate to say that the number of insertions into the array Q which we denote by $C_{path-comp}$ satisfies

$$2^i \leq C_{path-comp} \leq 3 \times 2^{i-1} \tag{19}$$

The main part of the quadtree regeneration algorithm is the loop for computing the paths of the N objects. This is given below.

for $p \leftarrow 1$ to N **do**
    for $m \leftarrow 1$ to s **do**
        n = find-attribute-interval(m, o_p)
        $Q[m,n] \leftarrow Q[m,n] \bigcup \{o_p\}$
        if $(a_p > 0)$ and (o_p crosses $Q_{m,n+1}$) then

$Q[m, n+1] \leftarrow Q[m, n+1] \bigcup \{o_p\}$
if ($a_p < 0$) and (o_p crosses $Q_{m,n-1}$) then
$Q[m, n-1] \leftarrow Q[m, n-1] \bigcup \{o_p\}$
endfor
endfor

If we let $C_{generation}$ denote the CPU cost of regenerating the quadtree, then $C_{generation} = NC_{path-comp}$. Hence the cost of regeneration satisfies the inequality

$$N2^i \leq C_{generation} \leq 3N2^{i-1}$$

The insertion of an object o_p in $Q[m, n]$ expressed in the above algorithm as a set union could be implemented to be $\mathcal{O}(1)$ if we use an array of size B for $Q[m, n]$. The CPU cost of regenerating our quadtree is also optimal in the sense that no multiple insertions or recomputations are done for a single object. In summary, no extra overhead is incurred to create and place index points in their correct quadrants other than the strict minimum. Once the quadrants array Q is filled, we just transfer its contents from memory to the disk by allocating one disk page for every $Q[m, n]$ ($0 < m, n \leq 2^i$) and copying the index points in $Q[m, n]$ to it. This amounts to exactly 2^{2i} disk page accesses which are also the minimal number of accesses possible hence the optimality of the algorithm.

We would now like to examine what the above bounds translate to in practice. We want to estimate the CPU cost or execution time $C_{generation}$ and the cost of 2^{2i} disk accesses (I/O cost) for several values of N. For the latter cost we will use maximal and minimal disk access costs of 10msec and 30msec respectively which are typical values of modern hardware used in recent studies as well. For the CPU cost, we have to estimate the execution cost of the body of the inner loop in the path computation algorithm which iterates between $2^i N$ and $3 \times 2^{i-1} N$ times. An accurate estimation is needed because the number of iterations is in the order of millions for a big N. Before that, we present a direct comparison between the naive approach to index reconstruction and our new optimal algorithm in terms of the number of disk accesses needed (not the resulting time). This is shown in Table 4.3 for representative values of

N	Min. disk time	Max. disk time	Min. CPU time	Max. CPU time	Min. total time	Max. total time
1000	0.04sec	0.12sec	0.1sec	0.15sec	0.14sec	0.27sec
5000	0.64sec	1.92sec	0.2sec	0.3sec	0.84sec	2.22sec
10000	2.56sec	7.68sec	0.8sec	1.2sec	3.36sec	8.88sec
25000	40.96sec	2.04min	8sec	12sec	48.9sec	2.24min
50000	2.73min	8.2min	32sec	48sec	3.26min	9min
75000	2.73min	8.2min	48sec	1.2min	3.53min	9.4min
100000	10.9min	32.77min	2.13min	3.2min	13.05min	35.97min

Table 4.2: Runtime estimations of CPU and disk access overheads; *sec* stands for seconds and *min* stands for minutes.

N	Naive QRA	Optimal QRA
1000	0	16
1200	0	16
1400	256	64
1500	1007	64
2500	8980	64
2800	13724	256
3000	18970	256

Table 4.3: Comparison of the naive and optimal quadtree reconstruction algorithms (QRAs) in required number of disk accesses.

N between 1000 and 3000. The values are hardly comparable. For the naive algorithm, we used the experiment where buffer size was 32. It is clear that the naive approach is extremely inefficient compared to the one we proposed here. We return to CPU cost estimation for the optimal algorithm.

The main statements of the loop body include a function for computing one attribute interval for the m^{th} time slice and an object o_i. To optimize code we could get rid of function call overhead and include directly the function body. Furthermore, find-attribute-interval(m, o_p)'s statements do not involve iteration and we could arrange it to use a closed formula to find n. It then reduces to a small number (two to four) of floating point operations to which we (pessimistically) give a few microseconds to execute. The remaining statements constitute a collection of 10 comparisons and one or two array assignment

statements whose overhead is about a few hundred nanoseconds (in fact, in a RISC pipelined architecture they may take much less). Then, we choose to assign the latter collection of statements 1μsec and the code for computing n 4μsec for a total cost of 5μsec. In Table 4.2, minimum and maximum values for disk cost, CPU cost, and total reconstruction cost are provided. For several values of N, the product $2^i N$ is used to compute minimum disk cost and $3 \times 2^{i-1} N$ to compute maximum disk cost. Let us compare what we obtained with the previous experiments on build cost. For our standard buffer size of 32 and a value of N as low as 3000 we needed 20000 disk accesses which last 10 minutes (using 30msec access cost). With this method, we only need 2.2 seconds to rebuild a quadtree with 5000 objects. The naive quadtree regeneration algorithm rendered the whole indexing structure impractical to use for applications where the number of objects exceeds 2500. The new algorithm yields reasonable performance for values of N around 25000 where the average cost is between 1 and 2 minutes. This is especially so if we could start rebuilding the tree before the current period ΔT ends. With this algorithm we thus have improved the manageable number of objects by an order of magnitude. However, our (ideal) target is $N = 100000$ for which we still have impractical regeneration time (half an hour!). We think that 25000 objects is already a big system size enough for a (relatively) wide spectrum of applications and that the execution time for $N = 10000$ for example is quite good (average is below 6 seconds). As for our target of 100000 objects, given the optimality of the algorithm, there is little hope to achieve it with improvements of the above method. A (radically!) different approach or indexing structure seems necessary to achieve a practical solution (if one exists) for such a large N. Let us now take a look at query performance.

4.7 Query Processing

We study two types of range queries: instantaneous and continuous queries. Both are two dimensional range queries each with a time range and an attribute range. An instantaneous query submitted at time t_{now} with an attribute range $[R_{low} \ldots R_{high}]$ targets the time range $[t_{now} - \delta t \ldots t_{now} + \delta t]$ where δt is a small

time lapse to be chosen according to the application domain. Assume we are at the i^{th}-regular quadtree. Then we may constrain δt to be small compared to a single time interval of the quadtree (i.e $\delta t \ll \frac{\Delta T}{2^i}$). In an i^{th}-regular quadtree, the number of disk accesses required to answer a range query is (intuitively) equal to the number of quadrants covered by the range. Let us characterize this more accurately.

Let $A_Q = R_{high} - R_{low}$ denote the length of the attribute range of a query Q and $T_Q = T_{high} - T_{low}$ the length of the time range of query Q. Excluding the effect of buffering, the disk access cost C_Q^{disk} of such a query is

$$C_Q^{disk} = \left(1 + \left\lceil \frac{A_Q}{\delta a_i} \right\rceil\right)\left(1 + \left\lceil \frac{T_Q}{\delta t_i} \right\rceil\right) \tag{20}$$

We have simply multiplied the number of intervals covered by each of the two ranges. For the special case when one range starts exactly at the beginning of an interval, the 1 is removed from its corresponding multiplicand in Equation 20. If further, the range ends exactly before the beginning of an interval then we also remove the ceiling expression. As ranges are supplied independently of the current status of the quadtree and its partition, we expect the general case embodied in Equation 20 to hold most of the time. We can then determine the cost of instantaneous and continuous queries using this formula.

When we assumed $\delta t \ll \frac{\Delta T}{2^i}$ above, we actually wanted the instantaneous query to fit in a single time slice δt_i. Alternatively we may adopt the policy of evaluating an instantaneous query submitted at t_{now} using the time interval in which t_{now} falls since our purpose in using parameter δt was to have a finite approximation to the infinitesimal t_{now}. For continuous queries, the theoretical time range over which they are evaluated is $[t_{now} \ldots \infty)$. In practice, if a continuous query comes in a period $[p\Delta T \ldots (p+1)\Delta T)$, it is first evaluated over the time interval $[t_{now} \ldots (p+1)\Delta T)$ then over all subsequent periods of the application until it is explicitly deleted from the queries list. Letting C_{inst}^{disk} and C_{cont}^{disk} denote the disk cost of instantaneous and continuous queries (respectively) over an i^{th}-regular quadtree we obtain the following formulas.

$$C_{inst}^{disk} = 1 + \left\lceil \frac{A_Q}{\delta a_i} \right\rceil \tag{21}$$

$$C_{cont}^{disk} = 2^i \left(1 + \left\lceil \frac{A_Q}{\delta a_i} \right\rceil\right) \tag{22}$$

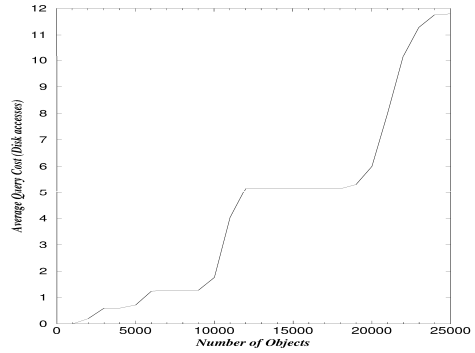

Figure 4.20: Range size = 10%

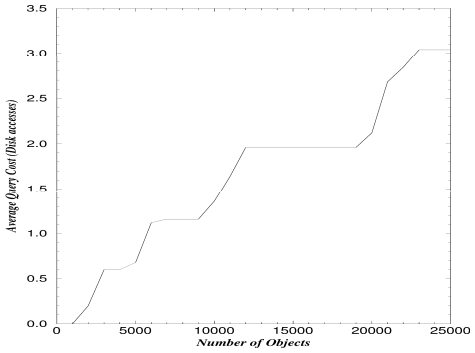

Figure 4.21: Range size = 1%

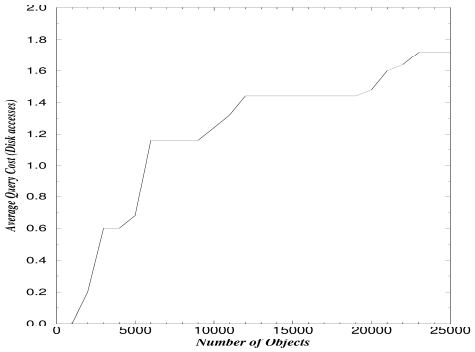

Figure 4.22: Range size = 0.1%

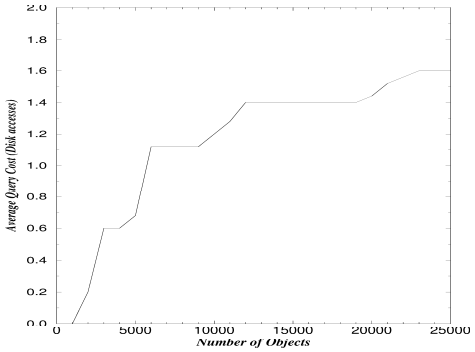

Figure 4.23: Range size = 0.01%

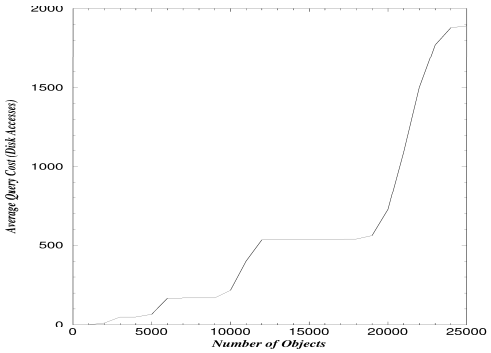

Figure 4.24: Range size = 10%

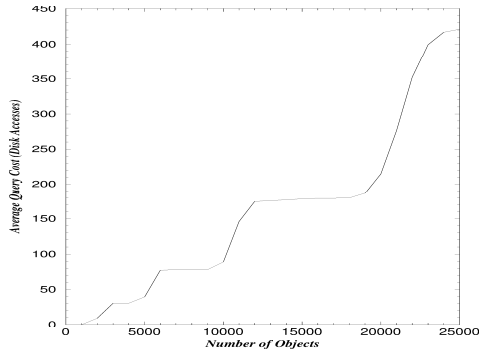

Figure 4.25: Range size = 1%

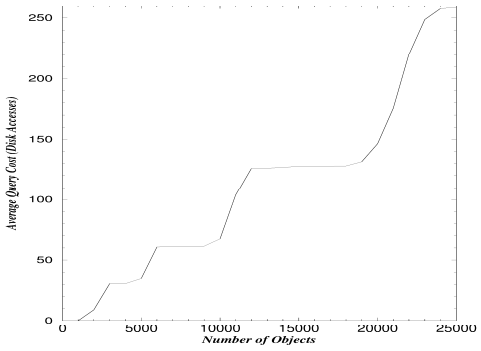

Figure 4.26: Range size = 0.1%

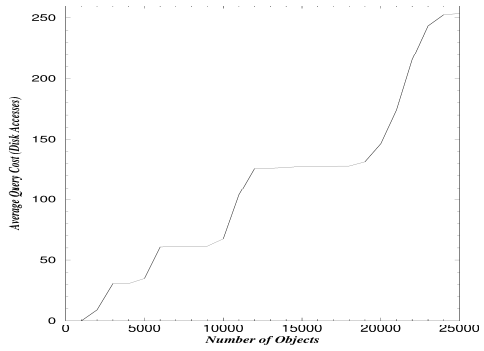

Figure 4.27: Range size = 0.01%

The cost of an instantaneous query is just the number of attribute intervals its attribute range spans while for continuous queries it is that number multiplied by 2^i, the number of time intervals in ΔT. Figures 4.20, 4.21, 4.22, and 4.23 show the average cost of instantaneous queries as a function of N across a few typical attribute range percentages (0.01%, 0.1%, 1%, and 10%). The range percentage is the length of the query range divided by ΔA and multiplied by 100. Notice that for small percentages (0.01% and 0.1%) we roughly need one disk access. The graphs exhibit the usual plateau pattern. The graphs for continuous queries' average disk access cost are given in Figures 4.24, 4.25, 4.26, and 4.27. As equation 22 predicts, for small percentages of the attribute space (0.01% and 0.1%) the average cost is exactly equal to 2^i at the i^{th} plateau (32, 64, 128, and 256 in the figures). For larger ranges they are a multiple of 2^i. The CPU overhead is that of navigating through the quadtree from the root to the relevant leaves. For a duplication ratio D, this cost is $\mathcal{O}(\log_4(DN))$. A major drawback of disk cost for both instantaneous and continuous queries that was alluded to earlier is that the cost of the same query doubles when the quadtree passes from the i^{th} plateau to the $i + 1^{st}$ plateau.

Chapter 5

THE CROSS POINTS METHOD

In this chapter, we provide the results of our study of the cross points approach to dynamic attribute indexing. We start in Section 5.1 with a description of the program used in the simulation study and associated data structures. Section 5.2 contains the performance results of the experiments that were run on the method and some pertinent observations. Finally, in Section 5.3 we provide a critique that sums up most of our thoughts and observations pertaining to the general feasibility of the approach.

5.1 Program and Data Structures

The binary tree program has two main data structures, the binary tree node *bintree* (BT) and the *crosspoint* (CP) defined as follows:

$$BT =< ID, a, b, Left, Right, Par >$$

$$CP =< Time, Pos, ID_{low}, ID_{high} >$$

In the BT record, the fields ID, a, and b are the object properties defined earlier while the $Left$, $Right$, and Par are the classical pointers of a binary tree node. A crosspoint is uniquely defined by four parameters. These are the

time and position at which the intersection happens and the two participating objects. The fields $Time$, Pos, ID_{low}, and ID_{high} embody this information. To be able to correctly process an intersection, we need to know which object had a lower position than the other before intersection; the subscripts in ID_{low} and ID_{high} indicate this.

Note that no information about the objects is stored in a cross point except their IDs. The binary tree thus defined, does not contain the keys which serve to build and maintain it. These keys are the positions of the objects and are computable for a given time t using the a and b fields from the linear motion equation so that $key(t) = at + b$. This computation is done at every node along the path in an insert, delete, or search operation and the result is compared with the given key to decide to go left or right or stop. This is a design decision. Before that we attempted using the time at which an object was last invoked in an intersection event (call it t_{CP}) to compute the position y_{CP} of an object o residing in the tree. We then would assign y_{CP} to the Key field of o in BT. Resorting to such a policy would have two virtues. First using a Key field allows us to get rid of including the slope and intercept fields in BT hence saving four bytes per object in the tree. Second, we would save floating point computations in each search operation ($\mathcal{O}(\log N)$) which might be a tangible performance gain for large N since search is an integral part of all query processing operations as well as insert, delete, and update requests. It would anyway be interesting to find a correct and efficient way to store keys even without saving up fields of the BT record.

Finally, we mention two more record structures from the workload generator program. These are the operation record OP which encompasses update, insert, and delete requests and the query record Q used to store information about instantaneous and continuous queries. These are defined as follows:

$$OP =< Time, ID, a, b >$$

$$Q =< Time, R_{low}, R_{high} >$$

The $Time$ field stores the time at which the request arrived to the system. Fields R_{low} and R_{high} store the lower and higher ends (respectively) of the queried attribute space ΔR for range query Q. Queries and operations are

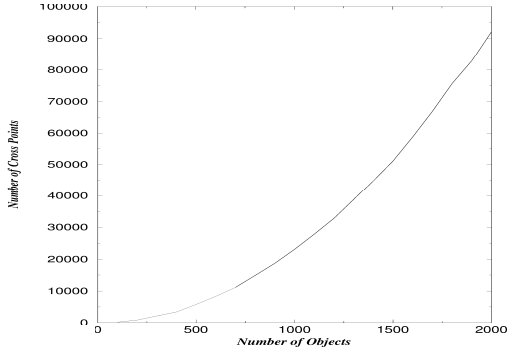

Figure 5.1: $\alpha = 0.1\%$.

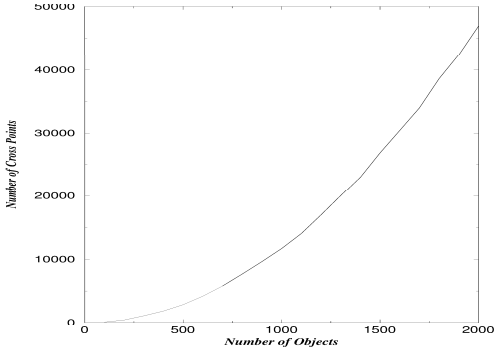

Figure 5.2: $\alpha = 0.05\%$.

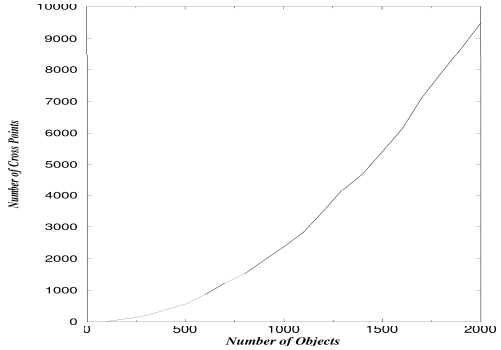

Figure 5.3: $\alpha = 0.01\%$.

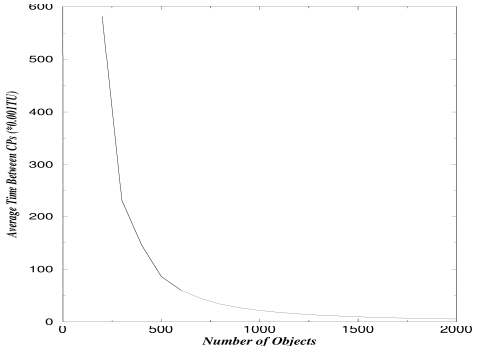

Figure 5.4: $\alpha = 0.1\%$.

stored in linked lists ordered by the *Time* field in non-decreasing order. At this point, we close the description of our programs which we hope is enough and pass to the results of the experiments.

5.2 Performance Results

The set of intersections or cross points turns out to play a decisive role upon which depends the very practicality and applicability of the method. First, we examine the size of this set in Figures 5.1, 5.2, and 5.3. Notice that the

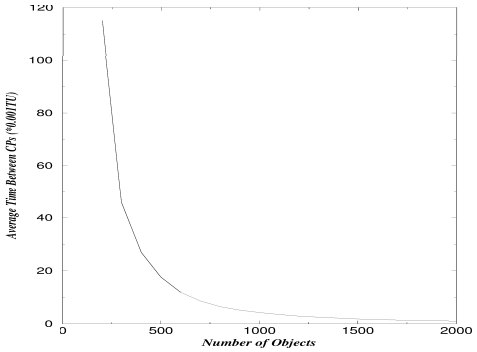

Figure 5.5: $\alpha = 0.05\%$.

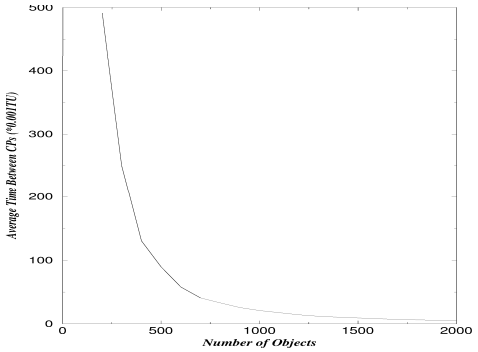

Figure 5.6: $\alpha = 0.01\%$.

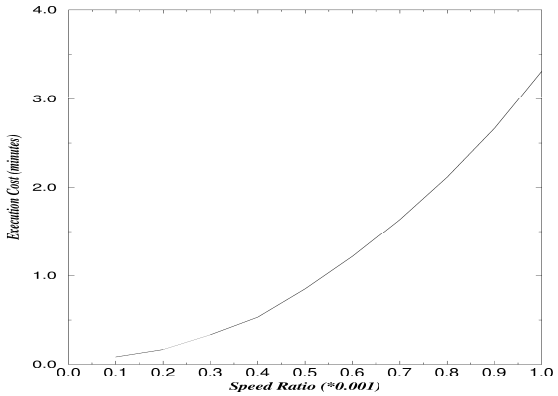

Figure 5.7: Execution time of the module which computes and stores cross points (N=1000).

maximum value we tried was 2000. This is because of the (discouragingly) huge number of resulting intersection points we got. In these graphs, we calculated the number of cross point events that occur within the first 100 TUs of our simulation. These exclude newly coming objects during those 100 TUs each bringing its own set of intersection points. As an example, for 2000 objects and $\alpha = 0.1\%$ we obtain about 100000 intersections; this is 1000 intersections per unit time. What is worse, the curve looks like a square function. This leads to a very small time lapse between one intersection and the next which we call *average time between cross points*. The values for this parameter as a function of N are shown in Figures 5.4, 5.5, and 5.6. At N=2000 and $\alpha = 0.1\%$ the average is around 10^{-4} TUs. It increases by an order of magnitude (to 10^{-3} TUs) when α drops by an order of magnitude (0.01%). The problem is that the values of N experimented with are very low and it is difficult to imagine the size of the set of intersection points at N=20000, not to mention our target of 100000 objects. Given that graphs have the shape of a quadratic function it is easy to see that the intersections set size will be impractical to handle both in space and time. We have not succeeded in finding a closed formula for the expected number of intersections in a system with N objects. We know however that it is $\mathcal{O}(N^2)$ and that the constant factor in the big-Oh notation depends mainly on α, the speed ratio. Finding such a formula would help a lot in understanding whether or not the difficulties of the cross points method are inherent but it turns out to be a challenging task.

As the number of intersections is quite large, it is only natural that merely computing and storing them consumes a lot of space and time resources. We

are more interested in the latter cost because storage space may still be affordable and space requirements could be reduced by decreasing the lookahead interval ΔT. In Figure 5.7, we present execution time of the module which computes the intersections between objects. The number of objects used was 1000 and we measure as a function of α which ranges from 0.0001 to 0.001 (rather optimistic). We note however that we use a linked list to store the cross points and found that the resulting insertion time of a single cross point is very expensive. The values are therefore exaggerated but still indicative. Our first conclusion is thus that cross points need at least a logarithmic time insert indexing structure such as the height-balanced AVL tree. We then also require that retrieving the next cross point (in the processing phase) take $\mathcal{O}(1)$ which was the case with our linked list (next cross point at the head of the list). Still however doing this may not be enough to make the method practical.

For example, the length of ΔT does not affect the complexity of the cross points computation algorithm. In other words, a smaller ΔT does not relieve us from the requirement to examine all possible pairs (hence the quadratic complexity is imposing). And no obvious subset of pairs of objects may be possibly known *not* to intersect in some near future time interval δt. This also gives us a rough criteria for the choice of ΔT. Namely, it should be as big as we afford memory to store cross points data and might be limited only by the constraint of a reasonable cross point retrieval time (preferably $\mathcal{O}(1)$ as mentioned above).

All these observations, together with the fact that we did not implement a height-balanced tree (just a normal binary tree) for the study, discouraged us from doing any experiments on query processing. However, it is easy to see before hand that insertion, deletion, and update overheads as well as query processing performance all benefit from the $\mathcal{O}(\log N)$ complexity. For a value of N as large as 100000 (our dream value), $\log N = 17$ which is roughly the actual number of floating point operations and comparisons needed to perform an insertion or a deletion plus the cost of rotations that keep the height balance (which are local most of the time). We then expect the cross points method to scale very well to large values of N if the problematic overhead of cross point management is somehow circumvented. The above observations pushed

us to reconsider the foundations of the method and go more deeply into its anatomy. We came up with a collection of ideas which we embody in the following critique of the method.

5.3 A Critique of the Cross Points Method

Our speculation about the difficulties experienced with the cross points method and the search for ways to circumvent them led us to believe that it is inherently inefficient with respect to the problem it tries to solve. This belief is based on the following important observation. While we would ultimately like to support range queries of different kinds, we are taking pains to record every event at which trajectories of any pair of objects cross each other. From a purely pragmatic standpoint, this information is not necessary for answering range queries. It is difficult to envisage an application wherein the event of two objects' attributes taking the same value at the same time would be a useful piece of information. Note that this the generalized understanding or definition of a cross point. To translate this to our favorite example of moving vehicles, we are recording every instance at which a vehicle overtakes another and every instance at which two vehicles come across each other when they are on the same route (running in opposite directions). Nevertheless, all we are demanding to know is which vehicles will be in a given segment of the route during a given time slice (this is the general two-dimensional range query). Taking this to a higher level of abstraction, we are dealing with the relationship between objects which when viewed mathematically has a quadratic cardinality in the system size N. We thus have unnecessarily raised both the space and time complexity of our problem when it did not require so. Remember that the management of cross points is the auxiliary overhead that was needed to keep the main indexing structure (the height-balanced AVL tree) functioning properly. We do not deny that by just looking at the binary tree storing attribute order, we have both an elegant structure and a very efficient one indeed with $\mathcal{O}(N)$ space overhead and $\mathcal{O}(\log_2 N)$ time overhead for all operations and queries supported. However, the price we pay for that is too high. We attempted to remedy this by trying to find a more efficient way to manage cross points.

We then had the impression that the cross points computation problem does not yield itself easily to complexity reduction. There apparently is little room for optimizations and the problem is inherently quadratic. To appreciate this more, consider the naive approach. We exhaustively examine all the N^2 pairs of objects (except of course an object and itself) and compute at which point in time the pair intersects. It might be in the future and it could have been in the past. If the intersection is in a reasonably 'near future' (e.g., within the specified lookahead interval ΔT) then it is stored or appended to the list of cross points taking into account its position in time relative to the other cross points. We can immediately spot the following two sources of inefficiency in this naive algorithm.

1. We are uselessly computing the cross point events which take place in the far future. Having discovered that an intersection is too far in the future for us to bear the cost of storing it, we just ignore it. Nevertheless, we will compute it again in the coming period when ΔT time has elapsed. In fact, if it is very far in the future, we may do the same computation several times (i.e. more than two) before we come to store it.

2. We are analogously, uselessly storing the cross point events which took place in the past. What is even worse, we in fact compute intersection events which never took place. Using the moving vehicles analogy, assume a moving vehicle suddenly 'popped up' (i.e., registered to our system) moving at a relatively high speed. Then the naive algorithm will assume it had to overtake or come across all the objects that are currently behind it (depending on their direction of motion and their velocities). However, it could simply have just entered the current road or highway from a junction. In other words, we are acting as if each object begins its journey along the attribute dimension from one of its endpoints.

Our attempts to reduce the 'search space' or rule out the useless pairs which are known not to intersect in the future failed to materialize into an algorithm. We remark however that by sorting the objects along the attribute dimension and also sorting then according to the absolute values of their slopes, we could know which subset of the N objects will meet in the future. To explain this

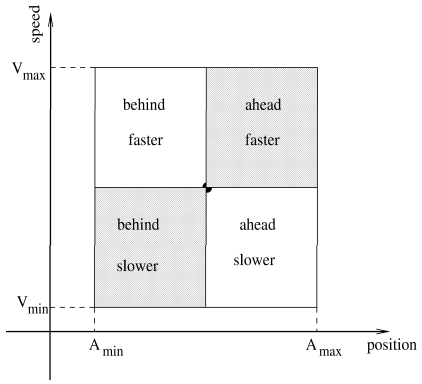

Figure 5.8: Subset of vehicles to come across a given vehicle in the future. The selected vehicle is at the center and the shaded area is for vehicles which will not meet it. The vehicles in this graph move in the same direction as the designated one in the center.

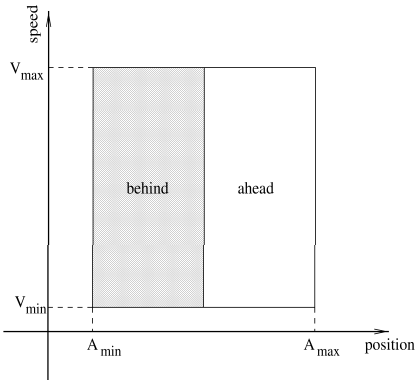

Figure 5.9: The same situation for vehicles moving in the opposite direction to that of the given vehicle. Speed is irrelevant here; it will meet with all vehicles currently ahead of it and with none of the vehicles behind it.

we again resort to the moving vehicles analogy. We just remind the reader that velocity corresponds to the slope or rate of change of the attribute value in the linear equation $f(t) = at + b$ associated with every dynamic attribute. The idea is best explained in Figures 5.8 and 5.9. We plot the objects in the position-velocity space[1]. For any arbitrary object in this space, we want to know which objects will intersect in the future. The space for objects moving in the same direction as its own is depicted in Figure 5.8 and the second case for objects moving in the direction opposite to its own is shown in Figure 5.9.

Unfortunately, this does not culminate in any computation which allows us to determine the forthcoming intersections in less than N^2 complexity. However, it seems we have looked from too narrow an angle to hope for a solution when we considered a single object's prospects. From a global system perspective, we would like to know about the criteria (if any) for predicting the forthcoming cross point events. A more useful and precise question would be the following: given a snapshot of the system at some point in time, is there a mathematical expression for the next pair or m pairs of objects that will

[1]The sign of the slope which corresponds to the direction of motion might be conceived of as the third dimension. Instead, we simply use two graphs.

intersect? Consider the objects in sorted order along the attribute dimension. Let o_i and o_{i+1} be two adjacent objects at positions a_i and a_{i+1} moving with 'velocities' (or slopes) v_i and v_{i+1} respectively. By looking at the sign of their slopes (which corresponds to the direction of motion for vehicles) we could know whether or not they will intersect in the future. If this is the case, then the time t_{CP} at which their intersection event takes place is given by the formula

$$t_{CP} = \left| \frac{a_{i+1} - a_i}{v_{i+1} - v_i} \right| \tag{1}$$

This is just the distance separating the two objects divided by their relative speeds. The pair of objects which will intersect next is simply the pair with the lowest t_{CP}. This could only lead us to an $\mathcal{O}(N)$ algorithm for finding the next cross point and an $\mathcal{O}(mN)$ algorithm for finding the next m cross points in order which ironically turns out to be worse than the $\mathcal{O}(N^2)$ naive algorithm.

We have thus provided a critique and an explanation of the difficulties and disadvantages of the cross points method. We also described a few of our attempts to introduce improvements on the method which, though did not culminate with success have resulted in a better understanding of its anatomy. We hope we were nor 'unjust' to the method and that we did not miss any important observation which might reverse the judgment.

Chapter 6

THE FP-INDEX

6.1 Introduction

In Chapter 4, we studied the use of the PR quadtree for dynamic attribute indexing. The study revealed high storage requirements which in turn affect query processing performance. This compelled us to investigate new ways of indexing that do not have this disadvantage. The result of our investigation was the development of a novel indexing method that is at the same time much simpler and more efficient across most performance parameters when compared to the quadtree. The purpose of this chapter is to present the details of the method. We also contribute a new algorithm for processing continuous queries that is optimal in the number of disk accesses required. With this algorithm we could even reach an average of less than one disk access per continuous query. We start by giving the motivation behind the method in Section 6.2. Section 6.3 gives a description of our new access method; we called it the *Fixed Partitioning Index* (FP-index). In Section 6.4, we digress briefly on the practical values for α and the length of a single time unit; this is a necessary background for our subsequent analysis and estimations. Section 6.5 provides an analysis of build cost and Section 6.6 describes the primary and secondary memory requirements of the index with comparisons to the quadtree. Section 6.7 gives insertion cost and presents a technique called *delayed insert* for improving it. In Section 6.8, we analyze the expected cost of range queries when using our index; both

continuous and instantaneous queries are considered. Section 6.9 gives the details of our novel optimization algorithm for continuous queries. Finally, we conclude the chapter in Section 6.10.

6.2 Motivation

We have noted earlier that one of the drawbacks of the quadtree approach is an excessive number of copies (or a high duplication ratio D) that directly affects all kinds of operations and both kinds of queries performed on the tree. However, we must remark that there is no way to avoid duplication which is inherent to our problem. Since we are monitoring an attribute over time, we must have more than a single copy of its corresponding object. What is then a reasonable number of copies? When does duplication start to involve unwarranted redundancy? Let us return to the specific picture of things in our quadtree.

We found that a trajectory spends a long portion (if not all) of ΔT crossing a single attribute interval (ΔT is the length of the time period between index reconstructions). If it takes n time intervals to cross that single attribute interval, then we are uselessly storing $n-1$ copies because all of the n copies convey one piece of information; namely that a given object crossed a given attribute interval between some time t_{enters} and another time t_{leaves}. We derived earlier the expression for the number of quadrants crossed by a trajectory which spans m attribute intervals over n time intervals, which was $n+m-1$. We could as well have used a single copy for every attribute interval covered which means m copies (then also $n-1$ redundant copies). This is the minimum we can afford if the subdivision of the attribute space is imposed on us (given, that is). The conception of the proposed method of dynamic attribute indexing is inspired by the foregoing observation. The fact that an object spends most of its time in a single attribute interval suggests to us that we might exclude the time dimension altogether from our indexing information and use a unidimensional index. The details of the method are outlined in the next section.

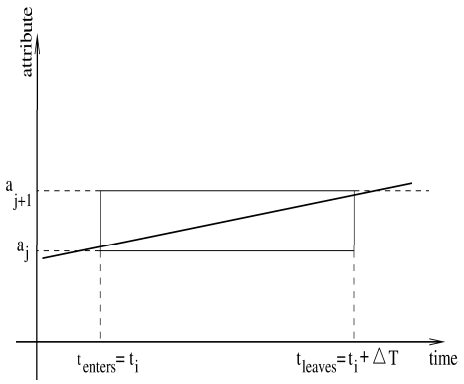

Figure 6.1: Non-transitory objects.

6.3 The FP-Index

The main idea of the FP-index is to keep the technique of indexing over consecutive time intervals of length ΔT (known as *sessions*) but index only on the single attribute dimension. We still regenerate and destroy our index every ΔT time units. We partition in a static way our attribute space $[a_0 \ldots a_0 + \Delta A)$ into E equal-sized attribute intervals each of length $\delta a_E = \Delta A/E$. We will call E the *partition size*. If between t_i and $t_i + \Delta T$, a trajectory crosses m such (consecutive) intervals then we need m copies of the corresponding object, one in each of the m intervals. Furthermore, given an interval and an object crossing it, besides storing the three object parameters $(< ID, a, b >)$ we also add the time t_{enters} at which the object has entered the interval and the time t_{leaves} at which the object left the interval (they satisfy $t_i \leq t_{enters}, t_{leaves} \leq t_i + \Delta T$).

According to when objects might enter and leave the attribute interval, we may distinguish three cases. These cases correspond to two categories of interval crossing. Objects might have entered the attribute interval at some time earlier than t_i, the beginning of the i^{th} session. Then they spend all the session inside this same attribute interval $(a_j < f(t) < a_{j+1}, \forall t : t_i \leq t \leq t_i + \Delta T)$. These objects are assigned to an independent category which we call the category of *non-transitory objects* depicted in Figure 6.1. They have $t_{enters} = t_i$ and $t_{leaves} = t_i + \Delta T$ although they actually entered before t_i and will leave after $t_i + \Delta T$ $(t_{leaves} > t_i + \Delta T)$. We choose this assignment for query processing purposes which make comparisons easy during the filtering of query data. On the other extreme are objects which enter the interval after t_i and

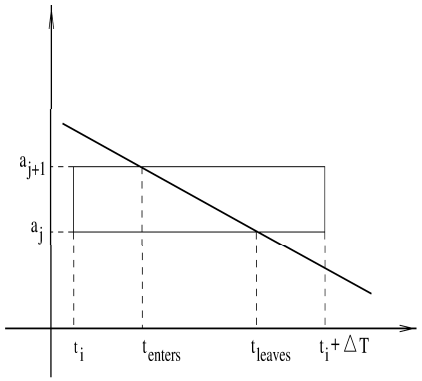

Figure 6.2: First kind of transitory objects: Trajectory enters the given attribute interval after t_i, the beginning of the session and leaves it before its end (at $t_i + \Delta T$).

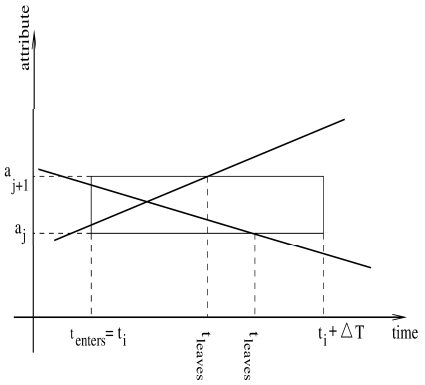

Figure 6.3: Second kind of transitory objects: Trajectory entered the given attribute interval in an earlier session (before t_i) and leaves it before the end of the current session (at $t_i + \Delta T$).

leave it before $t_i + \Delta T$ ($t_i < t_{enters}, t_{leaves} < t_i + \Delta T$) as shown in Figure 6.2. These belong to the second category which we call *transitory objects* because they do not 'live' in the interval for the length of the session but pass through it temporarily. Finally the third case of interval crossing whose objects are also transitory occurs when an object has entered the interval at some time before t_i but leaves the interval before the end of the session ($t_{leaves} < t_i + \Delta T$). This case is shown in Figure 6.3. For these objects we choose the assignment $t_{enters} = t_i$ although in reality $t_{enters} < t_i$ in order to simplify computations and filtering as mentioned above. In our application, the variables t_{enters} and t_{leaves} are needed in the processing of instantaneous queries which cover a thin time segment that must be compared with them to check for overlap. Having said this, let us go back to the FP-index.

This is a linear array A of E entries where the i^{th} entry is responsible for indexing the i^{th} interval. As such, it relies on a *Fixed Partitioning* of the attribute space hence the name FP-index. Figure 6.4 shows the memory and disk portions of the FP-index. Each entry $A[i]$ is a record $< FirstPage_i, LastPage_i >$ with two fields. $FirstPage_i$ is a pointer to the first disk page that contains data about the objects which cross the i^{th} attribute interval during the current session. Since a single attribute interval may easily be crossed by more trajectories than can be stored in a single disk page, $A[i]$ must point us to a set

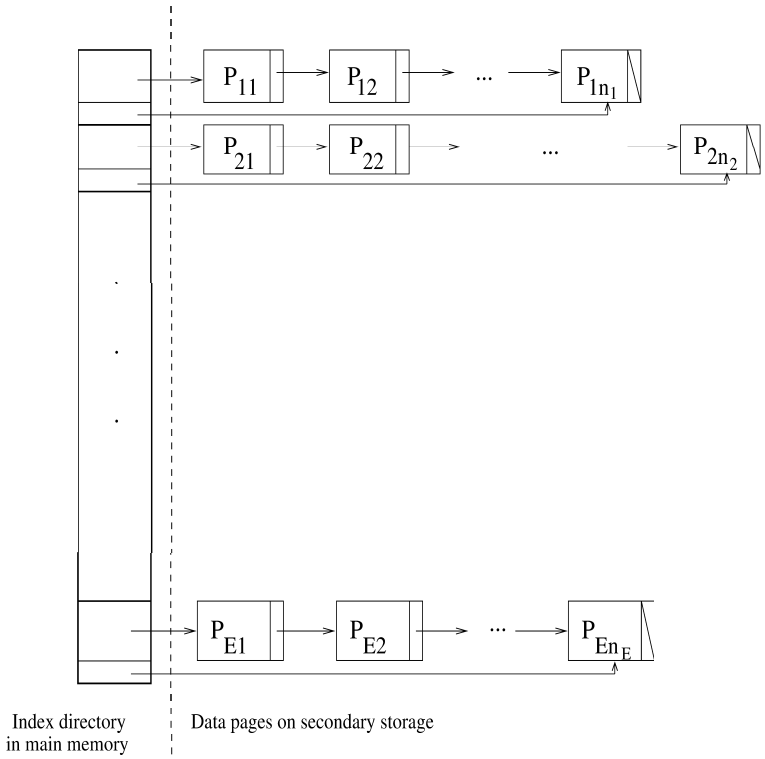

Figure 6.4: The FP-Index.

of disk pages. We implement this by including a field in every disk page that contains a pointer to the next disk page in the chain (of pages) corresponding to the current attribute interval. In order to facilitate insertion, we include the field $LastPage_i$ which points to the last disk page of the chain. Notice that we make no distinction between transitory and non-transitory objects in the index. All of them are stored together arbitrarily without regard to any information about time except the order in which they were inserted into the index during reconstruction. To gain more insight into the method, let us make some analysis and reconsider the amount of duplication involved.

Let $\overline{v}$ denote the average speed of objects in the system. In the FP-index, an object has as many copies m as the number of attribute intervals it crosses in a time ΔT. The distance covered by an object in a session is $\overline{v}\Delta T$; thus we have a duplication ratio D as follows.

$$D = \left\lceil \frac{\overline{v}\Delta T}{\delta a_E} \right\rceil \tag{1}$$

To ease our job, we will remove the ceiling from Equation 1; this results in an error less than 1/2. We will see later that our D is in the order of a few tens; in

fact even if it is smaller than that we still consider decimal fractions negligible. Then we have $D = \frac{\overline{v}\Delta T}{\delta a_E}$. Remember that the speed ratio α was defined by the formula $\alpha = \frac{\overline{v}}{\Delta A}$ as the fraction of distance traveled by an object moving with mean speed $\overline{v}$ in a single time unit with respect to ΔA. Then we have $\overline{v} = \alpha \Delta A$. Substituting this in Equation 1 we obtain

$$D = \frac{\alpha \Delta A \Delta T}{\delta a_E} \tag{2}$$

However, by definition $E = \frac{\Delta A}{\delta a_E}$. Substituting by E in Equation 2 we get the following neat closed form for the average number of duplicates.

$$D = \alpha E \Delta T \tag{3}$$

We have thus a deceptively simple[1] arithmetic equation that brings the four main parameters of our index together. These are the duplication ratio D, the speed ratio α, the partition size E, and the index regeneration period ΔT. It also provides us with a few hints on the mechanics of our method. Given a fixed α, the higher E the higher the number of duplicates we will have. This is intuitive since a high E means a fine partitioning of the attribute space that causes every trajectory to cross a larger number of intervals in a session (though still the same distance). D also increases when we increase ΔT a hint that gives us the factor that will deprive us from enjoying a long ΔT. This is also intuitive since given more time an object travels more distance hence more intervals. Related to this observation is an interesting special case of Equation 3; the case when the product αE equals 1. This yields the elementary equality

$$D = \Delta T \tag{4}$$

It says that we can 'enjoy' as many time units between index reconstructions as we can afford duplicates of our objects; indeed a pattern that we did not expect. We have to say first that this special case is not contrived or theoretical; given that α is a fixed application parameter, one simply chooses $E = \frac{1}{\alpha}$ and checks if it makes a reasonable D. In the next section, we present a practical case where it holds. Furthermore, the product αE is not a meaningless mathematical product. Since α is the fraction of distance traveled in a single time unit, αE is just the average number of attribute intervals crossed in a single time

[1] In fact this is one of the simplest possible four-variable equations.

unit. Equation 3 is very important and dominates the rest of this chapter; all subsequent analysis, estimations, and comparisons are dependent on it in one way or another. Before proceeding to work out its consequences, we introduce two quantities that are characteristic to the method. The first quantity is the number of index points P_E that we expect to have in a single attribute interval when our attribute space is partitioned into E intervals. It is readily computed as $P_E = (N \times D)/E$ where the product $N \times D$ gives us the total number of index points generated. The second quantity is presented through the following definition.

Definition 7 *An* **Interval Weight** w *is the number of data pages needed to store the index elements that belong to that attribute interval:*

$$w = \left\lceil \frac{N \times D}{E \times B} \right\rceil$$

where E is the partition size, N is the system size, D is the average number of duplicates per object in the system, and B is the page size (in records).

The value of B in our application is 204. This is obtained by dividing page size (4096 bytes) by record size which requires 20 Bytes in our case. Note that $w = \lceil \frac{P_E}{B} \rceil$. We also use $E = 1000$ as a fixed value in all that follows. We think it is a reasonable choice between the two extremes of high and low values. An unnecessarily big E yields an excessive number of copies which is the problem we set out to avoid in the first place. A small E is bad for query processing because the query answer becomes only a small subset of what we have to retrieve to compute it. We will also see that in practice it brings about reasonable values with the remaining three parameters (see the next section). It remains to experiment with values of α and ΔT and see what the resulting performance parameters look like. We will do this in the framework of a hypothetical application domain that justifies our selection of particular values for E, α, and ΔT; we hope this application is typical. Since this gets us nearer to real applications, we first have to understand better what α and time units stand for in practice. We see this in the next section.

6.4 The Nature of α and Time Units

We have all along been using α and ΔT in an abstract way. We want to know how large or small α is in a real application. Since α is the fraction of distance traveled in a single time unit, we first have to know how long is a single time unit. Is 1 TU equal to an hour? a minute? or a second? (or less?). We can examine the effect of each possibility in the context of an application. We consider a stretch of distance that is 1000 Km long and vehicles moving along it in both directions (directions correspond to negative and positive slopes) with an average speed $\overline{v} = 60$ Km/hour. We check the three cases one by one. If 1 TU = 1 hour, then objects move a distance of 60 Km out of 1000 Km in a single time unit; this yields $\alpha = 0.06$ and $\alpha E = 60$ (remember E is fixed at 1000). Then Equation 3 reduces to $D = 60\Delta T$ and it is up to us now to choose ΔT thereby monitoring the resulting D. If we let $\Delta T = 1$ TU then we regenerate the index once in an hour and accept to have 60 copies of each object (i.e., each object crosses 60 intervals of length 1Km in an hour). Using the same calculations, if 1 TU = 1 minute then $\alpha = 0.001$, $\alpha E = 1$ and we have the interesting special case mentioned earlier where $D = \Delta T$. Again it is up to us to choose ΔT and this time we choose $\Delta T = 60$ TUs which gives $D = 60$. We obtain the same number of copies per object to store but we also get the same regeneration period (60 minutes = 1 hour); we continue with the last case. If 1 TU = 1 second then $\alpha = 0.00001666$ and $\alpha E = 0.01666$ so that Equation 3 evaluates to $D = 0.01666\Delta T$. This time we choose $\Delta T = 3600$ TUs which again gives $D = 60$ and the same period of one hour (3600 seconds) between index regenerations. What does all this mean? First the selection of the length of a single time unit is immaterial, it worked neither for nor against us due to the following reason. The shorter is a single time unit, the smaller is the resulting α (since objects travel less distance when time is shorter). But then we counter this by assigning ΔT a proportionately larger number that keeps the real period length the same and yields the same D. Second the fact that α fluctuated from 0.06 to 0.00001666 without affecting the nature of our indexing problem suggests that we can always play with values (mainly ΔT) to get a tolerable duplication ratio D or at least find a compromise between ΔT and D. The reader might object that our choice of $\overline{v}$ (60 Km/hr) and ΔA (1000

Km) already determined everything and the rest was a change of units; this might be true. In general, we feel that a highly turbulent and dynamic system (i.e., a big α) or a very sluggish one probably need to be treated as special cases and pose a different indexing problem to be solved. Whatever the case, we will continue with the above context and in subsequent sections we will use the second alternative for the length of a single time unit, that is 1 minute. We chose this because it gives our special case $D = \Delta T$ and also relieves the reader from making any calculations to derive D from ΔT or vice versa in the examples that follow. For this we ask the reader to keep in mind that $D = 30$ means both that we incur 30 copies per object and that we generate the index every 30 minutes. In our attempt to understand the performance of the FP-index we use the values 30, 10, 5, and 1 for D. The latter value means that we may even afford a single copy per object in the whole index if we are able to regenerate it in a relatively short time every minute. We will also test four values of N which are 10000, 25000, 50000, and 100000. We start by looking at regeneration or build cost.

6.5 Build Cost

We build the FP-index in the same way as we did with the quadtree method. We allocate an array in memory of E elements and then go through the trajectories one by one inserting an index element with associated information t_{enters} and t_{leaves} in every interval crossed by the trajectory in the next ΔT time units. Since in this method we select ΔT and D beforehand, we know there will be about $N \times D$ operations of this kind. After finishing this, we go through the array entries and 'pack' every B index elements together, allocate a disk page for them, and write them to the disk. We also update our index array A to record information about the first and last disk pages storing data of the current attribute interval being processed. Here again, we view build cost as the sum of an in-memory processing phase cost and the cost of the phase where pages are flushed to disk. The cost of the latter phase is computed by multiplying the number of disk pages needed by the index by the disk access time. The total number of disk accesses needed is just E times the number of

N	Disk Access Time
10000	20 seconds
25000	5 minutes
50000	21 minutes
100000	1 hr 27 min

Table 6.1: Time to reconstruct the quadtree.

disk pages per interval, that is interval weight w. Thus we have

$$\text{Disk Access Cost} = wE \tag{5}$$

We will use the value of 20 milliseconds as an estimation of the time it takes to access a disk page. In our study of quadtrees we used a minimal value of 10 msecs and a maximal value of 30 msecs; here we are using their average. To have a fair comparison, we will also repeat estimations for the quadtree regeneration time using the value of 20 msecs as a disk access time. Furthermore, we note here that in an earlier table on build cost estimations for quadtrees we used the formula $i = \lfloor \log_2(N/B) \rfloor$ to compute the order of a quadtree with N elements. This formula gives an order i that is one less than the real order in practice due to the assumption of minimal split waves and a perfect pattern of plateaus and split waves[2]. In this chapter we will use the real order for which 2^i is as follows: 32 for N=10000, 128 for N=25000, 256 for N=50000, and 512 for N=100000. We remind the reader that 2^i is the side length of the partitioned space after inserting the N objects and the resulting quadtree requires $2^i \times 2^i = 2^{2i}$ pages of secondary memory. The resulting disk access time for the four studied values of N are given in Table 6.1. We have earlier said that for $N > 25000$ the quadtree becomes impractical. We did not include the CPU portion of build cost as it is overshadowed by the high disk access cost. Before we present the tables for the FP-index, we digress briefly on an aspect of build cost not mentioned before.

It was implicitly assumed and accepted throughout the study that the time spent rebuilding the index is an idle time during which no requests of any kind

[2]We already suggested that in practice we may add one to the computed order to allow for earlier splits.

N	D=30	D=10	D=5	D=1
10000	3	1	0.5	0.1
25000	7.5	2.5	1.25	0.25
50000	15	5	2.5	0.5
100000	30	10	5	1

Table 6.2: CPU cost of reconstructing the FP-index (in seconds).

will be processed[3]. We are then required to reduce build cost as much as we can to bring system idleness to acceptable levels. A measure of the goodness of a build time is how long it is compared to ΔT. Let $T_{rebuild}$ denote the time it takes to rebuild the index. We then formalize the notion of system idleness through a parameter we call *percentage idleness* defined by the formula

$$\text{Percentage Idleness} = \frac{T_{rebuild}}{T_{rebuild} + \Delta T} \times 100 \tag{6}$$

In practice, we would tolerate different percentage idleness values according to the nature of the application which involves such parameters as heaviness of workload or desired response time (for queries). In Tables 6.2 and 6.3 we provide the estimations of CPU time and disk access time in the rebuild cost of the FP-index; we also give percentage idleness[4].

The first thing we notice when comparing to quadtree values is that there is a remarkable improvement and that disk access cost for the FP-index is substantially lower than its quadtree counterpart. The biggest value we tried for N (100000 objects) which with a quadtree-based index required one hour and a half, now takes only 5 minutes at D=30 and 1.7 minutes at D=10. For N=50000 we have 1 minute at D=10 while it required 21 minutes with quadtrees. The CPU cost of reconstruction is almost negligible for most combinations of N and D as evidenced in Table 6.2. In both component costs of reconstruction, not only does the FP-index beat the quadtree in its feasible range of N (up to 25000) but it also produces practical costs for higher values of N which are infeasible with quadtrees. For our 'dream' value of N=100000, we may in fact consider D=10 to be a feasible solution where the FP-index

[3]The index could be reconstructed in parallel but this would require twice as much memory; we rule out this consideration anyway.

[4]We ignored CPU time in computing percentage idleness as it is negligible

N	D=30		D=10		D=5		D=1	
	Access Time	Percent. Idleness	Access Time	Percent. Idleness	Access Time	Percent. Idleness	Access Time	Percent. Idleness
10000	40 sec	2.17	20 sec	3.22	20 sec	6.25	20 sec	33.3
25000	1.3 min	4.15	40 sec	6.25	20 sec	6.25	20 sec	33.3
50000	2.7 min	8.26	1 min	9.09	40 sec	11.76	20 sec	33.3
100000	5 min	14.26	1.7 min	14.52	1 min	16.6	20 sec	33.3

Table 6.3: Disk access cost of reconstructing the FP-index.

is regenerated every 10 minutes and takes 1.7 minutes to construct yielding a percentage idleness equal to 14.52%.

Now what is the effect of D? It is directly related to the storage requirements of the index. Halving D (roughly) halves the number of disk pages consumed which in turn halves build cost. Besides, percentage idleness provides us with a heuristic for choosing a suitable D that does not unnecessarily tax our memory. When we have approximately the same percentage idleness for two different values of D then we select the smaller D to save space. Another pattern we see in Table 6.3 is that percentage idleness increases as D decreases. This reveals that although decreasing D causes build cost and ΔT to decrease by a similar factor, they apparently do not decrease proportionately enough to keep the ratio $T_{rebuild}/(T_{rebuild} + \Delta T)$ constant. An extreme case is at $D = 1$ when we are busy one third of the time reconstructing (20 seconds for every minute). This is due to the fact that the index has reached a number of disk pages (1000) below which it cannot drop which corresponds to the limit of $w = 1$ even if the actual number of index points requires much less memory. In fact, the case $D = 1$ is peculiar in many aspects. In a sense it is a cherished possibility which we would like (ideally) to be able to reach in every application. But as it is associated with positive implications it also has disadvantages; we shall return to this in more depth later.

6.6 Storage Requirements and Utilization

Intuitively, since we are incurring a minimum number of disk accesses in the index reconstruction algorithm for both the quadtree and FP-index, the drop in build cost must have come from a drop in the number of disk pages consumed by the index. In fact, this is what motivated the method in the first place. The FP-index requires wE disk pages. Besides this, we will also compute utilization ratio. One way to do this is to divide the number of index points that belong to an interval which is $\frac{ND}{E}$, by the capacity of the pages allocated to an interval which is wB. Then letting U denote utilization we have

$$U = \frac{ND}{wEB} \tag{7}$$

Since $w = \lceil ND/EB \rceil$, we obtain the more expressive form

$$U = \frac{ND/EB}{\lceil ND/EB \rceil} \tag{8}$$

It is then the relation of a quantity to its ceiling. Table 6.4 presents the computed storage consumption (in Mbytes) and utilization of our FP-index for various combinations of N and D. The corresponding storage consumption values for the quadtree are 4 Mbytes at N=10000, 64 Mbytes at N=25000, 256 Mbytes at N=50000, and 1 gigabyte(!) at N=100000. In the FP-index, the highest value is at N=100000 and D=30 where only 60 Mbytes are needed (as opposed to 1 gigabyte). As an example of improvements we take N=50000 at D=10^5 which consumes 12 Mbytes compared to the quadtree's 256 Mbytes. We then can safely conclude that we succeeded in devising an index that has practical and reasonable space requirements. Furthermore, unlike execution time estimations, our count of the number of disk pages required is fairly accurate and a real application may only exhibit negligible deviations.

Does it beat the quadtree on utilization? Not really. We have seen that in practice the quadtree oscillates between utilizations of 50% and 90% and has an average utilization of 75% in theory. For the FP-index, utilization is high at high values of N and D, and low at low values of N and D. However, a poor storage utilization is associated mainly with low storage consumption so that

[5]This is our favorite value for D; it seems to be a good compromise.

N	D=30		D=10		D=5		D=1	
	Requi.	Util.	Requi.	Util.	Requi.	Util.	Requi.	Util.
10000	8	74%	4	49%	4	25%	4	5%
25000	16	92%	8	61%	4	61%	4	12%
50000	32	92%	12	82%	8	61%	4	25%
100000	60	98%	20	98%	12	82%	4	49%

Table 6.4: Storage requirements (in Mbytes) and utilization of the FP-index.

N	D=10	D=5	D=1
10000	2	1	0.2
25000	5	2.5	0.5
50000	10	5	1
100000	20	10	5

Table 6.5: Actual memory utilization of the FP-index (in Mbytes).

the index is not big anyway. This is an important and positive observation though. The fact that when we have very low space requirements we also have a very small fraction of that space being actually used, suggests that the real amount of memory consumed by the index points alone is deceptively low. So low in fact, that we can venture to put it in main memory.

To see how far this was possible, we computed the actual memory requirements of the FP-index for D=10, D=5, and D=1. The results are shown in Table 6.5. Note that when the index moves to memory we are no longer constrained by disk page size B and could simply store the index points corresponding to each attribute interval in an array of fixed size. The reader may judge from the table which values of N we could realistically tolerate in main memory. However, it seems reasonable to us that values of N around 10000 yield an acceptable in-memory index. In fact, if the application at hand does not have many attributes to index we may consider N=50000 feasible which at D=5 requires 5 Mbytes[6]. It is not difficult to see the consequences of memory residence of the FP-index on query processing performance and especially insertion which becomes $\mathcal{O}(m)$ where m is the mean number of intervals

[6]This is only a subjective guess.

crossed by a trajectory (in a session) mentioned at the beginning of this chapter. Query processing reduces to simple array access and filtering. Let us note before moving to another topic that we consider this to be one of the major achievements[7] of the FP-index. System sizes which proved unmanageable with the cross points method and which the quadtree managed with difficulty and strain on memory and computational resources, are now tolerable even in main memory. As stated earlier, this was one of the major motivations that guided inception of the FP-index. A second motivation was to economize on query cost and a third was to reduce insertion, deletion, and update costs. Before moving on to discussing how far the FP-index succeeds in realizing these goals, we digress briefly on the topic of comparing its primary memory requirements with those of the quadtree.

6.6.1 Primary Memory Consumption

In this section we calculate how much memory the nodes of a quadtree require and compare it with the FP-index. Remember that the disk pages contain raw data while the indexing nodes which we store in main memory provide just information for navigation and search and pointers to disk pages at the leaf level. We have earlier refrained from treating this subject as the main memory requirements of index nodes were overshadowed by the secondary memory consumption of the data pages. Since we are here in the context of comparing quadtrees performance with that of our FP-index over several parameters, this brief discussion is in place.

A single record of the *quadnode* contains 12 fields each requires 4 bytes yielding a total of 48 bytes per node. It then remains to compute the number of such nodes. We will compute this number for regular quadtrees; those whose corresponding space is partitioned into a $2^i \times 2^i$ grid for some i[8]. If $s = 2^i$ is the side length of the grid, then we denote by L_s the number of nodes (leaf and internal) of the quadtree indexing an $s \times s$ partitioned space (remember that the partition is caused by insertions and splits and not vice versa). In this

[7]It has yet to be validated through experimental studies though.
[8]The order i of a quadtree, as defined in Chapter 4, coincides with its height.

N	s	L_s	Memory Consumption
10000	32	1365	64 Kbytes
25000	128	21845	1 Mbyte
50000	256	87381	4 Mbytes
100000	512	349525	16 Mbytes

Table 6.6: Primary memory consumption of quadtree nodes.

quadtree there are s^2 data pages pointed to by s^2 leaf nodes. Then we have

$$\begin{aligned} L_s &= s^2 + \frac{s^2}{4^1} + \frac{s^2}{4^2} + \ldots + \frac{s^2}{4^i} \\ &= s^2(\sum_{l=0}^{i} \frac{1}{4^l}) \\ &= \frac{4}{3}(1 - \frac{1}{4^{i+1}})s^2 \end{aligned}$$

The fraction $\frac{1}{4^{i+1}}$ becomes negligible very quickly (e.g., it equals 0.0002441 at i=5). Neglecting this fraction yields the simpler approximative formula

$$L_s \approx \frac{4}{3}s^2 \tag{9}$$

We use Equation 9 to compute the actual memory consumption for the four values of N we have been using in the study. The result is given in Table 6.6. Notice how consumption quickly becomes in megabytes. As opposed to these values, the primary memory consumption of the FP-index is hardly noticeable. It is an array of E records each consuming 8 bytes; at E=1000 it requires less than 8 Kbytes. Furthermore, its requirements of primary memory depends only on E and is totally independent from N. In summary, compared to the quadtree index, the FP-index's primary memory consumption is negligible. We continue with other aspects of performance.

6.7 Insertion Cost

One of the dramatic improvements we have achieved is in reducing insertion cost. In all circumstances, it requires one disk access for each interval crossed by the new trajectory during the time that remains until the end of the session.

We can do this because for each interval we have a pointer in our index A to the last page that contains data of that interval. This makes the number of disk accesses per insertion bounded by m, the mean number of intervals crossed in ΔT time. Besides the resulting lower cost of insertion which we analyze below, there is also the advantage of independence from N, the system size. The average cost of insertion which we denote by C_{insert} is then the same for an application with 5000 objects as for one with 100000 objects. It only depends on E and the average speed $\overline{v}$. On the other hand, C_{insert} doubles between one plateau and the next in the quadtree as was explained in Chapter 4 and is thus badly affected by an increase in N. For the FP-index a more accurate estimate of the average number of disk accesses would be $m/2$. This is because objects come at different times of the session and according to the position in time at which they appear they will have time to cross between 1 and m attribute intervals till the end of the session.

As a first optimization measure, we could refrain from inserting objects which come 'too late' in the session when there remains little time before reconstructing the index. Inserting such objects will hardly have any function other than burdening us with extra disk accesses if the remaining time is not enough to answer queries that involve them or if the resulting inaccuracy is negligible. On the other hand, we can still do better if we use a simple technique which we call *delayed insert*. The idea of this technique is to gather in memory a few insertion requests for every attribute interval and then flush their corresponding objects to disk together hence incurring a single disk access for them. We implement this as follows. We use a vector T whose length is the partition size E. Whenever a new object comes, we compute as usual which attribute intervals it will cross in the remaining time; assume it crosses m intervals. Then instead of seeking our index A and retrieving the m relevant page numbers to finish the insertion, we simply insert those m points in array T in their correct positions ($T[i]$ will receive the index point which falls in the i^{th} attribute interval). If during this insertion, some array element $T[i]$ reaches some threshold number of points q then we flush its contents to the disk. We call q the *queue size*. Vector T can then be implemented as an $E \times q$ two-dimensional array. We can immediately see that delayed insert with queue size q divides the average insertion cost per object by q. Ideally, if $q = m$ we

reach the cost of one single disk access per insertion as opposed to a single disk access per interval crossed. Unfortunately, there is an important limitation of delayed insert that may deprive us from such a luxury. Delaying insertion results in the following two kinds of misinformation.

1. *At the system scale.* There is a fraction of objects which have effectively entered the system but which do not figure in our index. A range query submitted at any time will then be missing a few objects; namely those which are still in the queue. Depending on the criticality of such information we may tolerate different error rates.

2. *At the object scale.* Since insertion proceeds by interval and no longer by object, we may have part of an object's points flushed to the index and the other part still pending in the queue. That is because an object's associated m intervals will usually not fill up at the same time. This might be less harmful to the accuracy of the answers computed for range queries. This is because it suffices that one index point associated with the new object (there are m of them) be already flushed to disk for a range covering its interval to include it in the answer.

We expect that at low values of q the error rate might be negligible and we may tolerate a small amount of inconsistency with reality. We have in mind values of q below 5. In fact, even using $q = 2$ is better than ignoring the technique altogether as it will cut in half the overall cost of insertions which is a significant gain in the long run. The simplicity and ease of implementation of delayed insert policy is another point in favor. Finally, we note that the cost of a deletion is $w \times m$. This is because once we know an object belongs to some attribute interval $A[i]$ $(1 \leq i \leq E)$ we have to retrieve all the w associated data pages to search for it while for insertion we just needed to access the last page of the chain. The cost of an update is the sum of insertion and deletion costs.

6.8 Range Query Performance

The case of range queries is slightly different from that of insertions although the improvement achieved in disk access cost is also substantial when compared with quadtree performance. Here, when a range covers any attribute interval, all the chain of disk pages associated with that interval (which is the interval weight w) have to be retrieved to answer the query. The cost is then no longer independent of N as was the case with insertion. Fortunately, the system size affects query cost in a much less severe way in the FP-index than it does with quadtrees where each split wave doubles the cost. We provide a direct comparison of continuous query costs in the FP-index and the quadtree index. We shall talk later about instantaneous queries. For the comparison to be fair, we use a standard hypothetical continuous query that covers 1% of the indexed space. When $E = 1000$, this coincides with 10 attribute intervals of the FP-index. However, this does not correspond to a rounded integer multiple of attribute intervals in the quadtree index (intervals do not have the same length in both indices). To remedy this, we add an extra interval in the quadtree case which mathematically means taking the ceiling. When we feel this was unfair to the quadtree method, we also add one or two intervals to the range length of the FP-index so as to make comparison as accurate as possible. The number of intervals into which the space gets partitioned in the case of the quadtree is not fixed but equal to 2^i where i is the quadtree order. In Table 6.7, we give the disk access cost of the continuous query mentioned above. Table 6.8 shows this cost in the FP-index for the four values of D we used earlier. It also contains an extra column for the percentage drop in cost achieved when comparing with the corresponding quadtree value. With the exception of one entry, all other percentages indicate that the number of disk accesses required drops by at least two thirds. In fact, the cost drops to one tenth its original value in most cases. Again, this was possible thanks to our success in eliminating redundancy and reducing substantially the required number of copies in the design of the FP-index. This dramatic percentage improvement translates into a tolerable number of disk accesses per query as the values in Table 6.8 suggest. For example, the values at D=30, D=10, and D=5 average to 40 disk accesses which with an average access time of 20 msecs

N	D	Disk Accesses
10000	32	32
25000	128	128
50000	256	768
100000	512	2560

Table 6.7: Disk access cost of continuous queries in the quadtree method.

N	D=30		D=10		D=5		D=1	
	Disk Access	Percent. Drop	Disk Access	Percent. Drop	Disk Access	Percent. Drop	Disk Access	Percent. Drop
10000	20	37.5	10	68.75	10	68.75	10	68.75
25000	32	75	16	87.5	10	92.18	10	92.18
50000	96	87.5	36	95.3	24	96.87	10	98.7
100000	150	94.14	50	98	30	98.82	10	99.6

Table 6.8: Disk access cost amd improvement achieved for continuous queries in the FP-index.

will take 0.8 seconds. Access time is even more reasonable at lower N. For example, at N=25000 and D=5 it is 0.2 seconds. As to the performance of instantaneous queries, given the absence of any indexing information related to the time dimension from our index, they will require the same number of disk accesses as a continuous query that covers the same attribute range. This is the main drawback of our FP-index and in this respect the quadtree remains always superior with its cost of a single disk access per interval covered by the query. However, we believe that this is not serious enough to compromise the applicability of the method in practice especially for the lower values of N given the big improvement achieved in continuous (hence instantaneous) query performance.

Another relief is that if we find a way to batch the processing of continuous queries and do it in an efficient way at the beginning of sessions, then we will have more time for answering instantaneous queries and thus improve overall response time. The next section explores this new topic in depth.

6.9 An Optimization Algorithm for Continuous Queries

In this section, we present the algorithm which minimizes the disk access cost of continuous query processing. Section 6.9.1 introduces the necessary concepts. In Section 6.9.2, we present the algorithm and in Section 6.9.3 we analyze its performance.

6.9.1 Introductory Notions

We have earlier said that a continuous query coming at time t_{now} will be evaluated over the time starting from t_{now} and till the end of the session during which it was submitted. From then on it will be reevaluated every ΔT time units after every reconstruction of the index until it is explicitly deleted. We then have a set of queries with their associated upper and lower limits to be evaluated at the beginning of each session. An important observation intrinsic to the nature of range queries is that ranges may overlap. Furthermore, the higher the number of such queries and the wider their ranges, the more the amount of overlap increases. The heart of our optimization algorithm is the exploitation of overlap to reduce to a minimum the total number of disk accesses required to answer the set of continuous queries. Since overlap is a central notion in our exposition, we shall seek to quantify it more concretely in a way that emphasizes our goals. For this, we first need the following definition.

Definition 8 *Given a session s, an attribute space partitioned into E equal-sized intervals, and a set of continuous range queries over this space, the* **popularity** *p_i of the i^{th} attribute interval ($1 \leq i \leq E$) is the number of queries whose range contains or partially overlaps with it.*

An interval that is not covered by any query has a popularity of 0. We then express overlap in terms of p_i. We could simply choose the sum $\sum_{i=1}^{E} p_i$ as an indicator of overlap or the ratio $(\sum_{i=1}^{E} p_i)/E$ which actually means average popularity. However we choose the following expression which though looking

less natural agrees more with intuition.

Definition 9 *We characterize overlap of range queries over an attribute space partitioned into E equal-sized intervals by the* **Overlap Index** *which we denote by OI and express using the formula*[9]

$$OI = \sum_{i=1}^{E} (p_i - 1)[p_i \neq 0]$$

We subtract one from each non-zero popularity and take the sum over the E intervals. The boolean expression $[p_i \neq 0]$ is used because an interval of popularity 0 does not contribute to overlap. Similarly, an interval of popularity one being covered by a single query range should not contribute to overlap hence the justification of $p_i - 1$. Hereafter, we say that a query *references* an attribute interval or that attribute interval is *referenced by* a query if the range of the query contains or partially overlaps with that query. Let us turn to our algorithm.

6.9.2 The Algorithm

The main idea is that if an attribute interval is covered by many queries, we should still bring its associated data pages to the buffer only once. While its data pages are in the buffer, we use them to compute answers for the referencing queries. What we will do is then the following. We will go through the attribute intervals one by one and bring into the buffer the data pages of all intervals of non-zero popularity. Once all referenced intervals' data pages are in the buffer, they can be used to compute the answers of all the queries. For this we will temporarily assume that the buffer is large enough to hold the disk pages associated with all the referenced intervals. We use this assumption mainly to ease our presentation; later we suggest remedies to this problem that will allow us get rid of it. To be able to describe the algorithm, we need some terminology and notation related to attribute intervals and queries. This is given below.

[9]The square braces are what D. E. Knuth called Iversonian notation after its originator Kenneth Iverson. In [GKP89] he says 'the idea is simply to enclose a true-or-false statement in brackets and to say that the result is 1 if the statement is true, 0 if the statement is false'.

- d_i: Number of data pages associated with the i^{th} attribute interval.
- p_i: Popularity of the i^{th} attribute interval.
- A^i_{min}: Starting point of the range of the i^{th} range query.
- A^i_{max}: Ending point of the range of the i^{th} range query.
- N_Q: Total number of range queries to be processed in the current session.
- $QID_{i,j}$: Identity number of the j^{th} range query that references the i^{th} attribute interval.
- C_i: Total number of data pages containing data required to answer the i^{th} range query.

Next, we present our data structures. We will make use of five arrays; let us justify each of them. The first is our prominent FP-index which we called array A. It is needed to retrieve from disk the data pages associated with intervals of non-zero popularity. The second is the array which contains the continuous queries to be processed which we call QL (for Queries List). The third is an array which stores the popularities of each of the E intervals; we call it array P. For the time being, popularities are needed to distinguish zero popularity intervals from non-zero popularity ones and determine accordingly which data pages will be needed and which will not. The fourth is an array that stores for each interval the IDs of the queries referencing it. We call it the *interval references array* and denote it by R. Obviously array R's contents will have to be computed by our algorithm. The fifth and final array is one that records for every query which buffer pages contain the data pages needed for answering it; we call it array B. This array is the one that will be used for query processing and it may be seen as the output of our optimization algorithm. The array R of interval references is needed only as an intermediate data structure that allows us to compute the contents of array B. Below, we give a summary of these arrays in more formal notation.

- A: The FP-index; $A[i] =< FirstPage_i, LastPage_i > \ (1 \leq i \leq E)$.
- QL: The queries array; $QL[i] =< QID_i, A^i_{min}, A^i_{max} > \ (1 \leq i \leq N_Q)$.

```
for i ←1 to N_Q do
    b_i= First attribute interval covered by QL[i]'s range.
    e_i= Last attribute interval covered by QL[i]'s range.
    /* update interval references and popularity array */
    for j ← b_i to e_i do
        P[j] ← P[j] + 1
        R[j] ← R[j] ∪ {QID_i}
    endfor
endfor

/* compute the contents of the B array */
for i ←1 to E do
    if P[i] ≠ 0 then
      Transfer the chain of pages starting at A[i].FirstPage
      to buffer positions b_i1, b_i2, ..., b_iC_i
      for j ←1 to p_i do
          QID_i,j ← R[i, j]
          for k ←1 to C_i do
              B[QID_i,j] ← B[QID_i,j] ∪ {b_i,k}
          endfor
      endfor
    endif
endfor
```

Figure 6.5: The optimization algorithm for continuous query processing.

- P: The popularities array; $P[i] = p_i$ $(1 \leq i \leq E)$.
- R: The interval references array; $R[i, j] = QID_{i,j}$ $(1 \leq i \leq E$ and $1 \leq j \leq p_i)$.
- B: Array of query references to buffer pages; $B[i] = \{b_{i1}, b_{i2}, \ldots, b_{iC_i}\}$ $(1 \leq i \leq N_Q)$.

The algorithm is given in Figure 6.5. It has two separate phases driven by the two independent loops. The first phase computes popularities of attribute intervals (i.e., the p_i's) and fills the interval references array R appropriately. It does this by going through the list of queries' ranges and computing for every query the intervals it references. Each such reference is then recorded in array R and the associated popularity is incremented. The second phase is

dedicated to computing the contents of array B which as mentioned before is the algorithm's output and may also be seen as containing a map of the buffer upon which depends the batch execution of the set of continuous queries. The job of the outer loop is to go through the list of intervals one by one and bring from disk to the buffer all the data pages associated with intervals of non-zero popularity. We let the pages of the i^{th} interval reside in buffer positions $b_{i1}, b_{i2}, \ldots, b_{iC_i}$. It now remains to record these positions for each query whose range covers the i^{th} interval; this is the task of last two nested loops. This way, each query will know later where on the buffer its associated data resides. Once our optimization algorithm finishes its job, we are ready to process queries. To process the i^{th} query, we read all buffer pages mentioned in $B[i]$ and merge them. We then filter out duplicates and get the answer. Let us analyze the performance of the algorithm.

6.9.3 Performance Analysis

In this section, we analyze the performance gain achieved by the algorithm described above and suggest ways to remove the assumption of a large buffer. Let I denote the total number of intervals referenced by the set of continuous queries in a session. Then I may be expressed as follows.

$$I = \sum_{i=1}^{E} [p_i \neq 0] \tag{10}$$

First, notice that $I \leq E$ no matter how large is the set of queries to be processed. Second, wI is the resulting optimal number of disk accesses. If E, the partition size is given and fixed then there is no way to avoid accessing a disk page that was referenced by one or more queries; we simply cannot compute a correct query answer without it. We forward this observation in the form of a proposition.

Proposition 1 *Given a fixed partition size E of the attribute space, the above algorithm processes any set of continuous queries using an optimal number wI of disk accesses where w is the interval weight and I is given by*

$$I = \sum_{i=1}^{E} [p_i \neq 0]$$

How many disk accesses does this algorithm save us compared to the previous approach? For each referenced i^{th} interval ($p_i > 0$), the previous approach incurred wp_i disk accesses while our new optimal algorithm incurs only w disk accesses. The gain is then $w(p_i - 1)$ when $p_i > 0$. Thus, the total gain is given by the sum $\sum_{i=1}^{E} w(p_i - 1)[p_i > 0]$. This is the product of interval weight and overlap index hence we get

$$\text{Disk Cost Gain } = w \times OI = w(\sum_{i=1}^{E}(p_i - 1)[p_i > 0]) \tag{11}$$

This means that for a fixed I, the higher the number of queries the more gain we achieve and the lower the average cost of a single query. The latter cost may even drop below a single disk access if there are 'enough' queries. When does this happen? If the set of N_Q queries span I attribute intervals, then exactly wI disk accesses are incurred. It thus suffices to have $N_Q > wI$ for this to happen. We have therefore managed through the use of batch processing and exploitation of query overlap to devise a query processing algorithm that incurs less than a single disk access per (continuous) query. Now, let us return to our initial assumption on buffer size and see how we can avoid it.

We assumed the buffer big enough to hold all of the wI referenced pages which is expressed by the constraint $BF \geq wI$. Since I is always bounded from above by E and we cannot guarantee any bound lower than that, our real constraint is in fact $BF \geq wE$. This is very unrealistic as it plainly requires the buffer to hold the whole data in the worst case. In order to circumvent this, we need to find a method that allow *incremental query processing* or a way to decompose I. In fact, we will do both. In our context, incremental query processing is the ability to compute the answer of a query without the restriction that all its referenced pages reside in the buffer at the same time. Unless we impose an upper bound on the maximum length of attribute space that could be queried, this ability is necessary. For the second goal of decomposing I (or reducing it somehow), we introduce the following term.

Definition 10 *An* **Interval Cluster** *(or simply cluster) is a set of adjacent attribute intervals $IV_k, IV_{k+1}, \ldots, IV_{k+l-1}$ of non-zero popularity surrounded from the left and right by intervals of popularity zero; i.e., popularities satisfy*

$p_{k-1} = 0$, $\forall i : k \leq i < k + l \;\; p_i > 0$, *and* $p_{k+l} = 0$. *We say* l *is the* **cluster length** *and* $w \times l$ *the* **cluster size**.

The condition $BF \geq wI$ must be satisfied only if all of the I referenced pages are contiguous. If there are any gaps then we have our I references partitioned into a set of clusters $\mathcal{C} = \{C_1, C_2, \ldots, C_\gamma\}$ where γ is the number of such clusters. Our algorithm needs a very simple change to recognize gaps. We just need to make $p_i = 0$ a stopping condition for bringing further pages into the buffer. We then process all the queries that caused or formed the current cluster (with their interval references). Having done this, we could move on to the next cluster, bring its pages to the buffer, and compute the associated queries' answers in a totally independent manner. A useful and important observation is that the set of queries causing cluster C_i which we call S_i is disjoint from the set of queries that cause cluster C_j $(i \neq j)$; formally $S_i \cap S_j = \phi$. This requires the assumption that if a query conjunctively references two disjoint segments of the attribute space, it will be decomposed into two separate subqueries with different QIDs. Its desired consequence is that the pages of a given cluster could take the places in the buffer of the pages of the previous cluster without causing problems. It also reduces the amount of main memory needed for query processing[10] since it allows us to divide the set of queries into γ mutually disjoint subsets each corresponding to the cluster it caused and evaluate each subset in series. Let l_{max} denote the maximum cluster length; we have the lighter constraint on buffer size

$$BF \geq w \times l_{max} \tag{12}$$

Let us then see how to process incrementally a query that involves more data pages than that our buffer can handle. For such queries, we just bring as many pages as we can afford, run through their data filtering out duplicates, and append the result to the query answer. We then do the same with the remaining sets of pages until we exhaust all of them. Finally, we make a last pass over the answer removing duplicates that were not recognized due to independent processing of chunks of pages. This algorithm could easily be extended to handle multiple queries that reference adjacent intervals. In this way, we have solved the problem of cluster sizes that are bigger than BF.

[10]This involves gathering relevant index points and filtering duplicates.

We propose then to go through the array P of popularities and derive from it the set of clusters and their sizes. We then process clusters which fit in the buffer in batch mode and process the rest using the incrementality technique explained above. The gain in the number of disk accesses remains the same.

6.10 Summary

In this chapter, we proposed a new indexing method for dynamic attributes that was inspired by the causes of space and time inefficiency in the quadtree approach. This method uses only the attribute dimension in indexing and boils down to a very simple unidimensional array. Although we have not yet done any experimental performance studies, the simple analysis provided here, shows that it is promising and that it beats the quadtree method in the following aspects:

1. The cost of reconstructing the index every ΔT time units.
2. The primary memory requirements of the index.
3. The disk storage requirements of the associated data.
4. The average number of duplicates needed per object.
5. The disk access cost of the insertion operation.
6. The disk access cost of continuous query processing.
7. The overall simplicity of the method and ease of implementation.

Furthermore we contributed an optimal algorithm for continuous queries that relies on batch processing at the beginning of sessions and the exploitation of query range overlap. It reduces the overall cost of the set of continuous queries to a small and tolerable number of disk accesses that is bounded from above, independent of the query load, and proved optimal.

Given that most improvements found by analysis are substantial in terms of percentage, we hope that experimental evaluation will only confirm our

expectations and that discrepancies are unlikely to reverse our conclusions on the virtues of the method. Our FP-index is thus worth studying.

Chapter 7

CONCLUSION

We have thus reached the end of this work. We would like to recapitulate upon what we have done and project into future work. This thesis was started with a proposal to study and compare the quadtree method and the cross points method. We believe that we succeeded in providing a detailed performance study of both of the methods so that very little (not to say nothing) remains to be said about their behavior in the chosen context and supported query types. Moreover, where any of the two methods fell short of being practical or of exhibiting reasonable performance, we contributed (whenever possible) our own algorithms and suggested modest techniques of improvement. When we failed to do so, as was the case with the cross points method, we tried to explain why the problem was inherently difficult. Finally, we distilled the experience gained through the study of both methods into a novel solution: a new indexing technique we called the FP-index which is promising.

The experimental performance study of both the quadtree and the cross points method counts as a contribution in its own right. Although it was overshadowed by many of our additions, we think it was not trivial anyway. It consisted of the development of a simulation model with associated parameters. This in turn required us to gather a thorough understanding and a holistic view of what is at stake in designing and assessing the worth of any new access method. As an example minor contribution, we distilled our understanding into a list of fourteen parameters that sum up the desirable properties in any access

method regardless of the nature of data it is meant for (the list is in Chapter 3). We then implemented the model to be able to conduct the simulation experiments and this involved implementing the quadtree with its associated operations as well as the implementation of a buffer manager. Overall, the performance study constituted more than half of our work.

In the quadtree method, we contributed an optimal algorithm for reconstructing the index. It has optimal complexity in the sense that it requires exactly one computation to place an index point in its right place and so no redundant computations are involved and overall CPU time is minimal. It also requires an optimal number of disk accesses as it relies on computing the contents of the index's data pages in memory and then transferring them together to the disk. We derived an analytical formula which allows us to predict the final shape of the resulting index given the system size N. In fact, the optimal quadtree reconstruction algorithm is based on it and would not have existed without it. We also provided a mathematical analysis of the average storage utilization of the quadtree which gave us the value of 75%. This at least served to improve our understanding of the mechanics of the method.

In the cross points method, we concluded that the method is impractical for values of N above 2000 due to the quadratic complexity of the cross points algorithm and the resulting impractical execution time of the cross points computation function in a real sample run. We then contributed an informal criticism of the method in which we conclude that the quadratic complexity of the cross points computation algorithm is intrinsic to the problem and thus insurmountable. As such, we could not go any further beyond the naive algorithm. We enriched the critique by presenting a few trails of thought and speculations about ways to improve upon the naive algorithm which we hope added more credibility to our arguments and improved the general understanding of how the method works.

We did not provide any comparative study between the quadtree and the cross points method. Since the early stages of our work, we sensed that if the cross points method is to make use of an in-memory data structure, there would later be little common ground for comparing the two methods. Our attempts to seek points of commonality did not culminate in success. In reality, we do

feel that the methods are fundamentally different. Nevertheless, it was clear to us right from the beginning that if the auxiliary overhead associated with the cross points method is acceptable and tolerable then it will indeed beat the quadtree approach over most parameters (if not all). This is because all the operations and query processing have complexity $\mathcal{O}(\log N)$ and no disk access costs are involved. As we could not circumvent the high cost of cross points management, the quadtree method is the declared winner.

Finally, we mention the contribution most valuable to us; this is the FP-index method. First, we have to say that although it was inspired by some of the disadvantages observed in the behavior of the quadtree index, it is not a mere extension nor a refinement of it. Rather, it is a radically different approach based on partitioning the attribute space in a fixed manner and indexing only on the attribute dimension. We have worked out all the important details of the method. Furthermore, through arguments and simple mathematical analysis we argued for the superiority of its performance relative to the quadtree across the majority of performance criteria considered. In the area of query processing, we started from the observation that the ranges of queries may overlap and devised an intelligent and optimal continuous query processing algorithm that exploits overlap. It potentially reduces the cost of a single continuous query to less than a single disk access. We strove to make our presentation of the FP-index as complete as possible missing only a few simulation experiments to substantiate our claims based largely on simple analysis and common sense. We strongly believe however that the arguments we presented are enough proof of the superiority of the FP-index and that even sizable error will not reverse judgment over most performance parameters. Add to this the fact that some conclusions are not dependent upon any estimations and are thus not prone to change in real experiments. As such, the experimental study of the FP-index is left for future work.

For additional future work, we would like to explore the possibility of an access method that does not need to be reconstructed periodically. We have been uneasy about the prospects of remaining idle more than 10% of the time just to destroy the current index and reconstruct the next one. We also consider this to be a disadvantage to the three methods presented in this thesis. It would

be nice to find a technique whereby the access method 'moves' in a more smooth and graceful way along the time dimension rather than have itself destroyed and regenerated. Ideally, we would like it to emulate the animal populations' evolution through time. An access method with such a property will also be more suitable for handling persistent queries which require referral to all past states up to a certain time point in the past (see Chapter 2). Adapting current access methods to handle persistent queries is also a future research problem.

Bibliography

[AK93] R. Alonso and H. F. Korth. Database System Issues in Nomadic Computing. In *ACM SIGMOD International Conference on the Management of Data*, pages 388–392, Washington, DC, May 1993.

[BGO+93] B. Becker, S. Geschwind, T. Ohler, B. Seeger, and P. Widmayer. On Optimal Multiversion Access Structures. In *Workshop on Advances in Spatial Databases*, pages 123–141, Singapore, June 1993.

[BKK96] S. Berchtold, D. Keim, and H. Kriegel. The X-Tree: An Index Structure for High-Dimensional Data. In *22nd Very Large Data Bases Conference*, pages 28–39, Bombay, India, September 1996.

[BKS96] T. Brinkhoff, H. Kriegel, and B. Seeger. Parallel Processing of Spatial Joins Using R-Trees. In *12th International Conference on Data Engineering*, pages 258–265, New Orleans, Louisiana, February 1996.

[BKSS90] N. Beckmann, H. Kriegel, R. Schneider, and B. Seeger. The R^*-Tree: An Efficient and Robust Access Method for Points and Rectangles. In *ACM SIGMOD International Conference on the Management of Data*, pages 322–331, Atlantic City, New Jersey, May 1990.

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

[DH95] M. Dunham and A. Helal. Mobile Computing and Databases: Anything New? *SIGMOD Record*, 24(4):5–9, December 1995.

[Ege93] M. J. Egenhofer. What's Special About Spatial? Database Requirements for Vehicle Navigation in Geographic Space. In *ACM SIGMOD International Conference on the Management of Data*, pages 398–402, Washington, DC, May 1993.

[EWK90] R. Elmasri, G. Wuu, and Y. Kim. The Time Index: An Access Structure for Temporal Data. In *16 International Very Large Data Bases Conference*, pages 1–12, Brisbane, Australia, August 1990.

[GB90] O. Günther and A. Buchmann. Research Issues in Spatial Databases. *ACM SIGMOD Record*, 19(4):61–68, December 1990.

[GB91] O. Günther and J. Bilmes. Tree-Based Access Methods for Spatial Databases: Implementation and Performance Evaluation. *IEEE Transactions on Knowledge and Data Engineering*, 3(3):342–356, September 1991.

[GKP89] R. Graham, D. E. Knuth, and O. Patashnik. *Concrete Mathematics.* Addison-Wesley, Reading, Massachusetts, 1989.

[Gre89] D. Greene. An Implementation and Performance Analysis of Spatial Data Access Methods. In *5th International Conference on Data Engineering*, pages 606–615, Los Angeles, California, February 1989.

[Gut84] A. Guttman. R-Trees: A Dynamic Index Structure for Spatial Searching. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, pages 47–57, Boston, Massachusetts, June 1984.

[Hen96] A. Henrich. Improving the Performance of Multi-dimensional Access Structures Based on k-d Trees. In *12th International Conference on Data Engineering*, pages 68–75, New Orleans, Louisiana, February 1996.

[IB92] T. Imielinski and B. R. Badrinath. Replication and Mobility. In *Second IEEE Workshop on Management of Replicated Data*, 1992.

[IB93a] T. Imielinski and B. R. Badrinath. Data Management for Mobile Computing. *ACM SIGMOD Record*, 22(1), 1993.

[IB93b] T. Imielinski and B. R. Badrinath. Mobile Wireless Computing: Solutions and Challenges in Data Management. Technical Report DCS-TR-296, Department of Computer Science, Rutgers University, New Brunswick, New Jersey, 1993.

[IB94] T. Imielinski and B. R. Badrinath. Mobile Wireless Computing: Challenges in Data Management. *Communications of the ACM*, 37(10):18–28, October 1994.

[KSSS89] H. Kriegel, M. Schiewitz, R. Schneider, and B. Seeger. Performance Comparison of Point and Spatial Access Methods. In *1st Symposium on Design and Implementation of Large Spatial Databases*, pages 89–114, California, July 1989.

[LM91] S. Lanka and E. Mays. Fully Persistent B^+-trees. In *ACM SIGMOD International Conference on the Management of Data*, pages 426–435, Denver, Colorado, May 1991.

[Lom87] D. B. Lomet. Partial Expansions for File Organizations with an Index. *ACM Transactions on Database Systems*, 12(1):65–84, March 1987.

[Lom88] D. B. Lomet. A Simple Bounded Disorder File Organization with Good Performance. *ACM Transactions on Database Systems*, 13(4):525–551, December 1988.

[Lom91] D. B. Lomet. Grow and Post Index Trees: Role, Techniques, and Future Potential. In *2nd Symposium on Advances in Spatial Databases*, pages 183–206, Zurich, Switzerland, August 1991.

[LS89] D. Lomet and B. Salzberg. Access Methods for Multiversion Data. In *ACM SIGMOD International Conference on the Management of Data*, pages 315–324, Portland, Oregon, June 1989.

[LS90a] D. Lomet and B. Salzberg. The hB-Tree: A Multiattribute Indexing Method with Good Guaranteed Performance. *ACM Transactions on Database Systems*, 15(4):625–658, December 1990.

[LS90b] D. Lomet and B. Salzberg. The Performance of a Multiversion Access Method. In *ACM SIGMOD International Conference on the Management of Data*, pages 353–363, Atlantic City, New Jersey, May 1990.

[Mul94] K. Mulmuley. *Computational Geometry: An Introduction Through Randomized Algorithms.* Prentice Hall, Englewood Cliffs, New Jersey, 1994.

[OU97] Özgür Ulusoy. Real-Time Data Management for Mobile Computing. Submitted for publication, 1997.

[PS85] F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction.* Springer-Verlag, New York, 1985.

[Sal88] B. Salzberg. *File Structures: An Analytic Approach.* Prentice-Hall, Englewood Cliffs, NJ, 1988.

[Sal94] B. Salzberg. On Indexing Spatial and Temporal Data. Technical report, College of Computer Science, Northeastern University, Boston, Massachusetts, May 1994.

[Sam84] H. Samet. The Quadtree and Related Hierarchical Data Structures. *ACM Computing Surveys*, 16(2):187–260, June 1984.

[Sam89] H. Samet. *The Design and Analysis of Spatial Data Structures.* Addison-Wesley, Reading, Massachusetts, 1989.

[Sat96] M. Satyanarayanan. Fundamental Challenges in Mobile Computing. Technical report, School of Computer Science, Carnegie Mellon University, Pittsburgh, New Jersey, 1996.

[SC91] J. Srinivasan and M. J. Carey. Performance of B-tree Concurrency Control Algorithms. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, pages 416–425, Denver, Colorado, May 1991.

[SK96] K. Sevcik and N. Koudas. Filter Trees for Managing Spatial Data Over a Range of Granularities. In *22nd Very Large Data Bases Conference*, pages 16–27, Bombay, India, September 1996.

[SL91] B. Seeger and P. Larson. Multi-Disk B-Trees. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, pages 436–445, Denver, Colorado, May 1991.

[SRF87] T. Sellis, N. Roussopoulos, and C. Faloutsos. The R^+-Tree: A Dynamic Index for Multidimensional Objects. In *13th Very Large Data Bases Conference*, pages 507–518, Brighton, England, September 1987.

[SW95] A. P. Sistla and O. Wolfson. Temporal Triggers in Active Databases. *IEEE Transactions on Knowledge and Data Engineering*, 7(3):471–486, June 1995.

[SWCD97] A. P. Sistla, O. Wolfson, S. Chamberlain, and S. Dao. Modeling and Querying Moving Objects. In *13th International Conference on Data Engineering*, pages 422–432, Birmingham, UK, April 1997.

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

[Tam96] R. Tamassia. Strategic Directions in Computational Geometry. *ACM Computing Surveys*, 28(4):591–606, December 1996.

[WCD+97] O. Wolfson, S. Chamberlain, S. Dao, L. Jiang, and G. Mendez. Cost and Imprecision in modeling the Position of Moving Vehicles. Submitted for publication, 1997.

[WJ96] D. White and R. Jain. Similarity Indexing with the SS-Tree. In *12th International Conference on Data Engineering*, pages 516–523, New Orleans, Louisiana, March 1996.

[Wol93] O. Wolfson. Data Allocation in Mobile Computing: A Project Description. In *IEEE Workshop on Advances in Parallel and Distributed Systems*, pages 89–94, October 1993.

[Wol96] O. Wolfson. Personal Communication. August 1996.

[ZMR96] J. Zobel, A. Moffat, and K. Ramamohanarao. Guidelines for Presentation and Comparison of Indexing Techniques. *ACM SIGMOD Record*, 25(3), September 1996.