The Dissertation Committee for Vasilis Samoladas
Certifies that this is the approved version of the following dissertation:

On Indexing Large Databases for Advanced Data Models

Committee:

Daniel P. Miranker, Supervisor

Annamaria B. Amenta

James C. Browne

Donald S. Fussell

Dimitris Georgakopoulos

On Indexing Large Databases for Advanced Data Models

by

Vasilis Samoladas, M.Sc., Eng.Dipl.

Dissertation

Presented to the Faculty of the Graduate School of
The University of Texas at Austin
in Partial Fulfillment
of the Requirements
for the Degree of

Doctor of Philosophy

The University of Texas at Austin

August 2001

On Indexing Large Databases for Advanced Data Models

Publication No. _____

Vasilis Samoladas, Ph.D.
The University of Texas at Austin, 2001

Supervisor: Daniel P. Miranker

In the last decade, the relational data model has been extended in numerous ways, including geographic information systems, abstract data types and object models, constraint and temporal databases, and on-line analytical processing. We study the indexing requirements of these data models. In many cases, these requirements are fulfilled by efficient techniques for multidimensional range search. Previous techniques for multidimensional range search, such as the R-tree and its variants, are based on ad hoc assumptions on the nature of the workloads they index, and have been known to suffer from reduced scalability and robustness. We adopt an alternative approach; our study focuses on techniques that provide worst-case performance guarantees, and thus overcome these deficiencies.

Indexability, proposed by Hellerstein, Koutsoupias and Papadimitriou, is a novel memory model for external memory. In indexability, the complexity of indexing is quantified by two parameters: storage redundancy and access overhead. Indexability focuses on the inherent trade-off between these two parameters. We study multidimensional range search under indexability. Our results are of two kinds; indexing schemes for various problems, and corresponding lower bounds. We develop indexing schemes for interval management, multidimensional arrays, and various types of planar range search. We derive a lower-bounds theorem for arbitrary indexing schemes, and apply it to multidimensional range search, proving most of our indexing schemes to be optimal.

We then leverage our theoretical work to the design of access methods. We solve the long-standing open problem of an optimal external-memory priority search tree. Our structure, the EPS-tree, is based on indexability results. We also explore dynamization, and develop techniques with optimal amortized and worst-case cost. We implement and evaluate experimentally our access method. Our experiments demonstrate that EPS-trees achieve excellent search and update performance, comparable to that of B+-trees on one-dimensional datasets. Our experiments with large datasets demonstrate the scalability and robustness of our techniques. We also affirm the relevance of space-I/O trade-off in achieving high indexing performance.

We conclude that the EPS-tree is an efficient, robust access method for a wide range of problems. Its success affirms the merits of systematic use of redundancy, and nominates indexability as a prominent methodology.

Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction

The adoption of a new data model by large-scale database systems requires efficient indexing techniques for implementing its language features on secondary storage. Such was the case for the the relational model, in the early 70's. Before its advent, the data models of choice were the hierarchical, and the network (CODACYL) data model. Both models exposed to the application the physical layout of disk-resident data. The relational model [Cod70] departed from this practice by hiding the actual structure of secondary storage. Thus, efficient associative access to the data became imperative. Almost concurrently, in 1970, Bayer and McCreight discovered the B-tree [BM70, BM72], and solved the most critical case of associative access: one-dimensional range searching of ordered sets of keys of type string, numeric, date, etc. It is widely believed that relational databases would not have enjoyed the popularity they enjoy, had the B-tree not accompanied them with such superb timing.

Analogous events took place in the mid-80's, when the development of structures such as the grid file [NHS84], the kd-B-tree [Rob84], and of course the R-tree [Gut85] launched the era of Geographic Information Systems

(GIS), by supporting efficient multi-dimensional range searching for geographic data. In subsequent years, a plethora of techniques was developed, including extensions of the R-tree, (R+-tree [SRF87], R*-tree [BKSS90]), techniques based on z-orders and other space filling curves, various quad-trees, and various special-purpose structures.

However, the multidimensional range search problem has proved to be vastly more challenging than one-dimensional range search. Even for the simplest multidimensional range search problems, no technique is widely acknowledged to be superior. Instead, there is a variety of data structures for conceptually similar range search problems. Choosing the best structure is a matter of the type and statistics of the data to be indexed, and also depends on the implementation complexity that can be afforded—since some of the best structures tend to have rather intricate implementations.

This situation is in sharp contrast with the ubiquity of the B-tree (and its variants) for one-dimensional range searching. where it is possible, and has actually been the practice in real systems, to use the same B/B+-tree implementation to answer any problem which can be stated a one-dimensional range search. Whether the data is numeric, representing sales figures, or strings representing last names, the same data structure is used to support point and range searching. This ubiquity can be summarized in the following statement: the B-tree exhibits optimal search performance, using optimal disk space. Of course, this statement, must be qualified by a definition of the cost model under which the B-tree is optimal.

1.1 New data models

The generality of the relational model is probably its most appealing feature. However, the implementation of this model by the majority of commercial database systems is somewhat restrictive. Initially motivated by On-Line Transaction Processing (OLTP), with a strong focus in business applications, the database vendors created complex monolithic systems, highly optimized to serve their main markets, but with weak support for other data-intensive tasks. This bias is reflected throughout a typical database product. SQL is arguably one of the most inadequate software development languages, totally lacking in modularity and extensibility, but actually quite adequate for business accounting software. Transaction support is also geared heavily towards a limited class of applications. This situation has been denounced repeatedly by both researchers and practitioners on technical grounds, but marketing concerns seem to override all technical arguments.

This situation has created a rather segmented database industry, with one large segment providing the mainstream database products for business accounting, and a number of smaller segments serving the needs of various other data-intensive applications, such as Geographic Information Systems (GIS), scientific and engineering applications, on-line analytical processing for business data (OLAP), computer-aided design (CAD), computed-aided software engineering (CASE), document management, workflow applications etc. Many of these peripheral segments build their products upon mainstream database platforms, but only use them as glorified storage servers, and employing only the most basic functionality.

Each of the many application-specific extensions of the relational model

is based on its own augmentation of the basic relational query language–typically some SQL dialect. Prime directions of augmentation include the introduction of new data types (such as geometric types), new semantics for relations (e.g. versions), more flexible storage management (such as datacubes for OLAP), additional modularity enhancements (such as classes, inheritance and encapsulation), novel transaction semantics (e.g. temporal extensions that can access previous database states), and interoperability (for data warehousing and e-commerce).

This variety of application domains reflects aspects of the real world, and is not a problem of itself. However, its consequences in the engineering of database systems have been somewhat problematic. Ideally, the different semantic models would be implemented as little more than translation layers, translating application semantics into programs over a small, robust set of primitive data management operations. These primitive operations would be implemented at a low-level layer which should be oblivious to the high-level semantics of the application at hand. In reality, the technology is far from this desirable state. Only the basic relational database technology is currently commoditized. Any extensions to the basic relational model, such as the ones mentioned above, currently require extensive specialized support at the lowest layers of the system.

1.2 Challenges in indexing for new data models

With respect to indexing in particular, specialized indexing support is currently the rule, in almost all application areas. This situation is reflected in the thrust of the indexing research, both academic and industrial. Indexing is one of the most active research areas in the database field. However, much of the effort is in developing special-purpose index structures, driven by the needs and characteristics of particular applications. This trend has some undesirable consequences. The major consequence is increased costs. Developing specialized solutions for small markets increases development costs. Also, the specialized solutions tend to be less stable, less optimized, and less robust, and thus incur higher operational costs.

In order to address the increased costs, most database vendors have recently provided low-level extensibility support for their general-purpose database products. This support is usually in the form of Application Programming Interfaces (APIs) to the storage management layer of their products, and has come under names such as "data blades" or "data cartridges". Although such facilities are helpful, the development cost of index structures is still significant.

Another source of complexity is the proliferation of data structures for conceptually similar problems. A thorough survey of spatial access methods by Gaede and Günther [GG98], lists over 50 spatial access methods (SAMs). Furthermore, despite a large research effort in this area, no clear winner has emerged. According to [GG98],

"Even for experts, it becomes more and more difficult to recognize

[...these access methods'] merits and faults...".

Furthermore, most of these techniques are not known to have robust performance over a wide range of problem parameters. Shifting needs in the same application domain, such as increased amounts of data, or new types of queries, may render previously efficient methods virtually useless.

This situation motivated research into some fundamental issues. On direction was the study of implementation abstractions, suitable for indexing. The pioneering work in this respect was the development of *GiST*, the Generalized Search Tree [HNP95] of Hellerstein, Naughton and Pfeffer. GiST implements parameterized search and update procedures, where the user can customize these operations by applying domain-specific heuristics. The success of GiST is not in improving access performance, but in substantially lowering the implementation cost of index methods.

The search and update operations of GiST, although not unduly restrictive, impose some limits onto the kinds of search that they allow. Thus, the developers of GiST recognized the need for a "theory of indexability", a theoretical framework that would "...describe whether or not trying to index a given dataset is practical for a given set of queries."

In the early 1990s, a few researchers pioneered a new approach into multidimensional indexing. Motivated by maturing techniques from computational geometry, they sought to develop a new breed of indexing techniques, with provably good performance for fundamental multi-dimensional search problems. Quickly it became apparent, that provably good performance was achievable, but often at a price; increased storage cost for the index structure. Also, some of the initial techniques developed would not meet the theoretically optimal performance expectations. Finally, these techniques had substantially

higher implementation complexity than their heuristic counterparts. For these reasons, these techniques have not yet been adopted into practice.

The call for a "theory of indexability" was answered by Hellerstein, Koutsoupias and Papadimitriou [HKP97], with the introduction of a new model of indexing, which strived to reconcile the theoretical work with some of the practical concerns that it had raised. The new model was particularly suited for the study of lower bounds, which was the focus of most of the results in [HKP97], but it also introduced a bold departure from previous theoretical models. Whereas previous models incorporated both the locality and the search aspects of indexing, indexability focused on the locality exclusively, abstracting away the search aspects of the problem.

1.3 Our thesis

In this dissertation, we develop the theory of indexability, with an emphasis on its application to multidimensional search. Our work includes both a broad theoretical study of indexing within this theoretical framework, but also extends to the application of the theoretical corpus towards practical solutions. Our results can be summarized as follows:

- We adopt indexability as a general, structured external memory model, suitable for indexing. Indexability introduces the *workload* as an abstraction of range search, and the *indexing scheme* as an abstraction of access method.

- We apply indexability to the important problem of two-dimensional range search. We develop indexing schemes for all special cases of this

problem. These include interval management, three-sided search, orthogonal segment intersection, orthogonal range search, and rectangle intersection.

- We also apply indexability to other problems of practical interest, including indexing multidimensional arrays, and indexing set-valued attributes.

- We develop a general theory of indexability lower bounds. We apply our theory to two-dimensional range search, and prove our indexing schemes to be optimal, or, in one case, almost optimal. We obtain similar results for multidimensional arrays and set-valued attributes.

- Leveraging our results on indexability, we solve the long-standing open problem of optimal three-sided range search in external memory. Our access method, the EPS-tree, is an adaptation of McCreight's priority search tree [McC85] to external memory. The EPS-tree achieves optimal search performance. We also develop update techniques with asymptotically optimal amortized and worst-case costs.

- We perform the first thorough experimental evaluation of an asymptotically optimal access method for multi-dimensional range search. Such techniques have been proposed before for other problems (e.g. [KRV+93, VV96b, RS94, AV96]), but were not evaluated empirically. Thus, they were considered by many (e.g. [ST99]) of mainly theoretical interest. Our results on the EPS-tree indicate that it exhibits excellent performance, scalability to large datasets, and robustness under various query distributions.

It is our thesis that provably efficient indexing methods are a valuable alternative for multi-dimensional range search, suitable for the increased indexing requirements of new data models. A critical aspect of most of these techniques is the disciplined use of redundancy. In this context, indexability is a valuable tool for the development of such indexing methods.

Chapter 2

Overview of Multidimensional Range Search

The basic range search problem is stated as follows: given any finite set of objects I, selected from a (possibly infinite) domain $D \supseteq I$, a (possibly infinite) set of ranges R, and an incidence relation $p \subseteq D \times R$, we wish to preprocess I into a data structure, such that, for a given range $r \in R$, all records $x \in I$, such that $(x, r) \in p$, can be retrieved efficiently. We often write $(x, r) \in p$ in its predicate form $p(x, r)$. Thus, a range search problem is defined by the triple (D, R, p).

Range search is of fundamental importance to databases; it corresponds to the generalized *select* operator of relational algebra, or, as it is sometimes called, θ-selection. The basic problem statement can be further elaborated, in various ways that we will examine next.

One generalization of the problem, introduced by Fredman [Fre80, Fre81], is as follows: let $(S, \oplus)$ be a semigroup, that is, let $\oplus$ be an associative and

commutative operation over S. Also, let f be a mapping from D to S. For a given range $r \in R$, we wish to compute the semigroup sum

$$\bigoplus_{x \in I \wedge p(x,r)} f(x)$$

This generalization can model the computation of *range aggregates*, such as the well-known SQL aggregates MIN, MAX, SUM, AVG, and COUNT. The basic range search problem—sometimes called the *range-reporting problem*, is stated by taking $S = 2^D$, $\oplus$ to be set union, and $f(x) = \{x\}$ to turn an object x into a singleton set.

Another distinction is between static vs. dynamic range search. In the static case, the dataset I is considered fixed. In the dynamic case, the data structure must be accompanied by algorithms, that support *updates* of the dataset I, that is, insertion of new elements, or deletion of existing elements. Special cases also apply. For example, in some cases we are interested in insertions only. In other cases, we are interested in *replacements*, that is, a deletion followed by an insertion.

Data structures for solving range search problems, are always specified within a particular *memory model*. In turn, the choice of memory model dictates the cost measures that determine the quality of the data structure. Typical cost measures reflect appropriate notions of space cost or time cost. Main memory data structures are typically described in the RAM model, or in the pointer machine model. Magnetic disks are modeled by block-oriented memory models, sometimes called *external* memory models.[1] External memory models are usually parameterized by the *block size*. This parameter is

[1]This is slightly abusive terminology, since, strictly speaking, external memory includes non-blocked devices, such as tapes.

introduced as a succinct way of reflecting technological aspects of magnetic disks, and in particular the high latency cost associated with disk head movement. In these models, space is measured in disk blocks, and time is measured in number of block I/O operations. A popular model is the *Parallel Disk Model* (PDM), introduced by Vitter and Shriver [VS94].

Arguably the most important, and certainly the most studied area of range search, is geometric range search, where the domain D and the range set R are spaces of geometric objects (points, lines, circles, rectangles, etc.), and the incidence relation p is some geometric relation (containment, intersection, alignment, etc.).

Of interest to databases, and particularly so for supporting new data models, is *multi-attribute search*. There, the domain D and range space R are composed as cartesian products of domain and range spaces, and the incidence relation p is composed as a boolean conjunction. More precisely, let relation $p_i \subseteq D_i \times R_i$ be seen as an incidence relation over domain D_i and range space R_i, for $i = 1, \ldots, d$. Then, let $D = D_1 \times \ldots \times D_d$, and $R = R_1 \times \ldots \times R_d$. Also, let $C(q_1, \ldots, q_d)$ be a d-ary boolean function, and define the incidence relation $p \subseteq D \times R$ as

$$p(x, r) \equiv C\big(p_1(x_1, r_1), \ldots, p_d(x_d, r_d)\big),$$

where $x = (x_1, \ldots, x_d)$ and $r = (r_1, \ldots, r_d)$.

In the most common case in database applications, (D_i, R_i, p_i) will be a simple, one-dimensional range search problem, where D_i is a totally ordered type (integer, string, date etc.), $R_i = D_i \times D_i$, is the set of pairs over D_i, and

$$p_i(x, (y_1, y_2)) \equiv y_1 < x < y_2.$$

When the boolean function $C(q_1, \ldots, q_d)$ is the conjunction

$$C(q_1, \ldots, q_d) = \bigwedge_{i=1}^{d} q_i,$$

the problem is known as *multidimensional range search*.

It is not hard to see that, the case of arbitrary boolean function C can be reduced to multidimensional range search. Also, by appropriate transformations, a number of quite diverse range search problems, including many geometric range search problems,[2] can be solved efficiently by reduction to multidimensional range search. Finally, multidimensional range search generalizes two important types of database search, *partial match queries* and *exact match queries*.

Because of its importance, multidimensional range search is the most intensely studied case of range search, and it is also the main focus of this dissertation. We now survey the work related to this problem.

2.1 Multidimensional range search in computational geometry

In the early days of computational geometry, multidimensional range search, also known as orthogonal range search, was one of the fundamental areas of interest. This interest produced a wealth of results, of which only few can be mentioned here. A number of books (e.g. [Sam89a, Sam89b]) and surveys (e.g. [Meh84, Mat94, AE97]) cover the subject very thoroughly.

[2] In the context of geometric range search, the term used is orthogonal range search.

2.1.1 Quadtree and kd-tree

One of the earliest data structures was the quad-tree, proposed by Finkel and Bentley [FB74]. In its simplest form, the quadtree is a tertiary tree. Given a set of planar points, and a splitting point, the set is split into four subsets, corresponding to the four quadrants defined by the splitting point. The root of the tree stores the splitting point, and each of these four subsets is recursively processed, and stored in the four subtrees of the root.

The quadtree is one of the most intensely studied data structures. The survey by Samet [Sam84] reports much of the relevant work. Unfortunately, the quadtree has bad worst-case behavior. Because of their simplicity, quadtrees have been externalized in various ways (e.g. [TVM98]). They are well suited for applications where the dataset is strongly regular, such as a rasterized image.

The kd-tree, introduced by Bentley [Ben75], improves upon the worst-case query cost of quadtrees. For a set of n d-dimensional points, the kd-tree requires time $O(dn^{1-1/d} + t)$ to answer an orthogonal range query returning t points.

Let S be a set of d-dimensional points. Every node of the kd-tree is associated with a subset of S. The root is associated with S. At every node of the kd-tree, depending on its distance δ from the root, the node's associated set is split into two parts, based on coordinate $x_{(\delta \mod d)+1}$. The two subsets resulting from the split, are recursively associated with the children of the node.

The kd-B-tree, introduced by Robinson [Rob84], was one of the earliest spatial access methods proposed. It is an adaptation of kd-trees to external

memory, combining the kd-tree and the B-tree. The kd-B-tree was the point of departure for Lomet and Salzberg [LS90], who introduced the hB-tree (which stands for "holey brick" B-tree). Another extension of the kd-B-tree was the LSD-tree of Henrich, Six and Widmayer [HSW89]. The main contribution of these extensions is a more efficient process of performing updates, which increases their robustness.

2.1.2 Processing intervals

A number of techniques for processing intervals have been proposed. Often, these techniques were developed as parts of more general techniques. For example, the *segment tree* was invented by Bentley, to solve Klee's measure problem, i.e., to compute the area of the union of n (possibly overlapping) rectangles. Also, the *interval tree* of Edelsbrunner was developed to solve the rectangle intersection problem (report all pairs of intersecting rectangles, from a given set of n rectangles). The general problem of answering range queries over sets of intervals is known as *interval management*.

These data structures have been the focus of significant effort in externalization. The I/O-optimal solution to interval management was given in [KRV+93]. Their solution, called the *metablock tree*, was fairly involved, and did not allow deletion. Ramaswamy and Subramanian proposed path caching [RS94], a technique that externalizes the interval and segment tree (and also the priority search tree, discussed below). However, this technique requires non-optimal space. In particular, the interval tree obtained by path caching requires $O(\frac{n}{B} \log B)$ disk blocks. Arge and Vitter [AV96] finally succeeded in fully externalizing the interval tree, with optimal query I/O, space,

and worst-case dynamic update cost.

2.1.3 Priority search tree

Virtually all planar orthogonal range search problems, can be solved optimally in main memory, by techniques based on the *priority search tree* of McCreight [McC85]. We will present this data structure in some detail, in Ch. 6. This data structure solves optimally the dynamic case of three-sided queries: organize a set of planar points, so that the points contained in a region of the form $[x_1, x_2] \times [-\infty, y]$ can be retrieved efficiently.

Because of its importance, the priority search tree has been the focus of many externalization attempts. Icking *et al.* [IKO87] proposed a structure using optimal space, but with query cost $O(\log_2 n + t/B)$ I/Os. The XP-tree of [BG90] also uses optimal space, but the query cost is $O(\log_B n + t)$ I/Os. Using path caching [RS94], the resulting structure has optimal query cost $O(\log_B n + t/B)$ I/Os, but requires non-linear space of $O(\frac{n}{B} \log \log B)$ disk blocks, and the update cost is $O(\log_B n)$ amortized. Finally, Samoladas, and independently Arge and Vitter, solved the problem optimally in both space and time, with optimal worst-case update cost. These results are reported in this work, and also in [ASV99].

2.1.4 Range tree

The range tree was discovered independently by Bentley [Ben80], Lueker [Lue78], Willard [Wil78], and Lee and Wong [LW80]. In two dimensions, the range tree is simply a balanced search tree over the x-coordinate of the points. Furthermore, in each node u is rooted another search tree, built over the points in u's

subtree, and ordered by the y-coordinate. Extension to higher dimensions is done recursively.

As described, the range tree over a d-dimensional point set, requires space $O(n \log^{d-1} n)$, and can answer range queries in time $O(\log^d n + t)$. Willard and Lueker [WL85] develop a comprehensive set of techniques for adding range search capability to dynamic data structures. Also, trade-offs between space and time are possible. Their techniques explore the theory of range trees.

2.1.5 Filtering search

One of the most elegant techniques in range searching is that of *filtering search*. This technique was explored by Chazelle [Cha86], although it was implicitly used in a few previous results (e.g. [McC85]). The technique is only applicable in range-reporting search, where the size of the result can grow as large as the original dataset. The basic principle of filtering search is quite simple; when the output of a query is large, the search can be slow. Chazelle used the principle of filtering search to devise new data structures for a number of basic range-reporting problems. Many of the techniques presented in that paper, were later adopted in different settings, including external memory.

For the interval stabbing problem, Chazelle offered a radically new technique, which was based on clustering. Given a set S of n intervals, it is possible to construct subsets $S_1, \ldots, S_k \subseteq S$, with $\sum_{i=1}^{k} |S_i| = O(n)$, such that for any stabbing query of size t, it is enough to "filter" one of the sets S_i, and keep only those intervals that satisfy the query, at the same time guaranteeing that $t = \Theta(|S_i|)$. A number of researchers applied this idea to different domains. Ramaswamy [Ram97] combined this idea with B+-trees to support indexing

for constraint and temporal databases. Kriegel *et al.* [KPS00] evaluate the implementation of this technique in the SQL procedural language of the Oracle8i server.

Another result of [Cha86] was the first optimal data structure for two-dimensional range search, on a pointer machine. Solutions based on the range tree would achieve optimal time, but with slightly suboptimal space $O(n \log n)$. Chazelle showed that it was a simple matter to apply the principle of filtering search, to improve space to $O(n \log n / \log \log n)$, and even further, to allow for optimal space-time trade-off. This technique was adapted to external storage by [SR95, ASV99].

In the same paper, Chazelle introduced the concept of a *hive graph*. This structure was used to improve the search cost of many previous data structures. It was later generalized by Chazelle and Guibas [CG86a, CG86b] to the technique of *fractional cascading*.

Other problems solved in [Cha86] using filtering search included point enclosure, segment intersection, and k nearest neighbors.

2.1.6 Orthogonal Geometric Range Search

Most of the techniques for orthogonal range search are based on range trees. The problem has received extensive treatment for decades. Thus, we shall only survey a few of the major, or more recent results. More extensive surveys can be found in [Meh84, PS85, AE97].

The best data structures known today are due to Alstrup *et al.* [ABR00]. They improved on the previous best bounds due to Chazelle [Cha86, Cha88]. The work of [ABR00] introduces two data structures for orthogonal range

search in $\mathbb{R}^2$, one requiring $O(\log\log n+t)$ time and $O(n\log^{\varepsilon} n)$ space, and the other requiring $O((\log\log n)^2+t\log\log n)$ time and $O(n\log\log n)$ space. They also show that any data structure for $\mathbb{R}^2$, with $O(f(n)+t)$ time complexity and $O(s(n))$ space complexity, can be used to construct a data structure for $\mathbb{R}^d$, with $O\big(f(n)(\frac{\log n}{\log\log n})^{d-2}+t\big)$ time and $O(s(n)\log^{d-2+\varepsilon} n)$ space complexity.

The above results are obtained in the RAM memory model. For the more restrictive pointer machine model, Chazelle [Cha90a] shows that there exists a data structure with time complexity $O(\log^{d-1+\varepsilon} n + t)$ and space $O(n(\log n/\log\log n)^{d-1})$. In external memory, significantly less is known.

Of practical interest are restricted cases of orthogonal search in $\mathbb{R}^2$ and $\mathbb{R}^3$. The priority search tree [McC85] of McCreight handles all special problems in $\mathbb{R}^2$ optimally. For $\mathbb{R}^3$, the main problem is that of dominance search; given any query point p, retrieve all points in the data set that are dominated by p in every dimension. Chazelle and Edelsbrunner [CE85, CE87] provide solutions taking linear space and query time $O(\log^2 n+t)$. Makris and Tsakalidis [MT98] improve the bound to $O(\log n + t)$ on the RAM, and to $O(\log n\log\log n + t)$ on the pointer machine, still with linear space.

In external memory, significantly less is known. The P-range tree Subramanian and Ramaswamy [SR95] can answer two-dimensional three-sided range queries with slightly suboptimal $O(\log_B n + t/B + \mathrm{IL}^*(B))$ I/Os per query,[3] using linear space. For two-dimensional range search, a structure of $O((\frac{n}{B})\frac{\log(n/B)}{\log\log_B n})$ blocks can be used. Ravi Kanth and Singh [KS89] study the case of non-replicating structures, and provide a structure with $O((n/B)^{(d-1)/d}+ t/B)$ access cost and optimal update cost $O(\log_B n)$. They also show that this

[3] $\mathrm{IL}^*(B)$ is the iterated $\log^*$ function, i.e., the number of times we must apply $\log^*$ to B before the result becomes ≤ 2.

cost is optimal for non-replicating structures. Theirs is the only external-memory optimal technique that extends to arbitrary dimensions.

In three dimensions, the only result to date is due to Vengroff and Vitter [VV96b], who show how to index a dataset in $O(\frac{n}{B}\log\frac{n}{B})$ blocks, and achieve a query cost of $O\big((\log\log\log_B n)\log\frac{n}{B} + t/B\big)$ I/Os for dominance queries. General three-dimensional range search can be performed with the same access cost, using space $O(\frac{n}{B}\log^4\frac{n}{B})$ blocks.

2.1.7 Other geometric range search problems

Computational geometry has studied other range search problems, besides orthogonal range search. Of these the most important is arguably simplex range search, and its special case of half-space range search. A simplex in $\mathbb{R}^d$ is the intersection of $d+1$ half-spaces, e.g. a triangle is a simplex in $\mathbb{R}^2$, a quadrilateral is a simplex in $\mathbb{R}^3$, etc. The simplex range search problem is to organize a set of points in $\mathbb{R}^d$, so that all points inside a given simplex can be reported efficiently. The problem has attracted considerable attention, and has motivated some of the most elegant results in the past two decades. Here, we shall mention selectively some basic techniques, that gave rise to recent results for external memory. For some excellent surveys, as well as more complete references, see [AE97, Mat94].

The earliest technique was the *partition tree*, proposed by Willard [Wil82]. It became the basis for most linear-space data structures for this problem. The original data structure had query cost $O(n^{\log_4 3} + t)$ for $d = 2$. This was improved by a series of papers, most notably by the seminal result of Haussler and Welzl [HW87], who derived a cost of $O(n^{1-\frac{1}{d(d-1)}+\varepsilon} + t)$. Vari-

ous improvements followed, culminating with the result of Matoušek [Mat92], who obtained $O(n^{1-1/\lfloor d/2 \rfloor}\text{polylog}(n) + t)$ query cost, still with linear space. These techniques were adapted to external memory by the works of Agarwal *et al.* [AAE+98], Kollios, Gunopoulos and Tsotras [KGT99], and Agarwal, Arge and Erickson [AAE00].

Another series of techniques began with the work of Chazelle, Guibas and Lee [CGL85], who introduced to the problem the concept of *arrangements*. For the planar case, their technique achieves optimal time $O(\log n + t)$ with linear space. However, generalizing the approach to higher dimensions, introduces non-linear space, typically exponential to the problem dimension. For example, the extension of Matoušek [Mat92] can answer halfspace queries in $\mathbb{R}^d$ in time $O(\log n + t)$, but requires $O(n^{\lfloor d/2 \rfloor}\text{polylog}(n))$ space. However, the technique is useful at least in low dimensions, and has been externalized by Agarwal *et al.* [AAE+98].

2.1.8 Lower bounds

In many geometric range search problems, there is no solution with both optimal time and space complexity. This state of things was observed early on, with many researchers calling for either improved upper bounds, or lower bounds that would resolve the issue.

Some early lower bounds by Fredman [Fre80, Fre81], Yao [Yao82] and Vaidya [Vai89] indicated that indeed, optimal query time $O(\log n + t)$ with linear space was impossible for many problems. However, it was the work of Chazelle [Cha95, Cha90a, Cha90b] that provided tight lower bounds for orthogonal range search, in various memory models (for main memory).

The results of Chazelle were not extended in external memory for some time. Subramanian and Ramaswamy [SR95] attempted to extend the result of Chazelle for the two-dimensional case. Unfortunately, their published proof is seriously flawed. Hellerstein, Koutsoupias and Papadimitriou [HKP97] introduced the indexability model, and published the first lower bounds, for restricted cases of two-dimensional search. Subsequently, Koutsoupias and Taylor [KT98, KT99] provided lower bounds for orthogonal range search, but within a simplified complexity framework, where the effect of the block size is not accounted for. Samoladas and Miranker [SM98] introduced a general framework for lower bounds, and proved tight bounds for restricted d-dimensional range search problems. Arge, Samoladas and Vitter [ASV99] combined this technique with the results of [KT98], and provided strong lower bounds for two-dimensional search, along with lower bounds on the space-I/O trade-off. These results have been obtained in the context of the indexability memory model. Extended discussion of this work can be found in Ch. 5 of this dissertation.

2.1.9 Persistence

Ordinary data structures are, in a sense, ephemeral; the past state of the data structure is erased, when an update is made. In persistent data structures, past states are available to access. There are two versions of persistence, depending on whether past states are also available for update. A fully persistent data structure allows past states to be updated (creating a new branch of versions at that state). In a partially persistent structure, only the most recent state can be updated (thus, the versions are linearly ordered).

Driscoll *et al.* [DSST89] introduced a general technique for making main-memory, linked data structures persistent. Previously, persistence was considered on a case-by-case basis, often within the context of more general problems. For example, Sarnak and Tarjan [ST86] use persistence to perform planar point location. Besides these applications, persistence can be used to support many of the operations in temporal databases [ST99].

The techniques of Driscoll *et al.* motivated a number of techniques for external persistent versions of the B-tree. Lanka and Mays [LM91] proposed the first, fully persistent B+-tree. Subsequent works by Becker *et al.* [BGO+96] and Varman and Verma [VV97] refined the approach, by addressing a number of practical issues, such as thrashing, and retirement of old versions to tertiary storage.

2.2 Multidimensional range search in databases

To a large extent, the initial development of multidimensional range search techniques for databases took little advantage of the advances in computational geometry. The contrast between the two bodies of work is striking; there is little, if any, geometry (in the mathematical sense) to be found in the mainstream database techniques for multidimensional range search, even in those application areas where the data is inherently geometric (e.g. geographic information systems).

Database research did incorporate some of the early works in computational geometry, most notably quadtrees and kd-trees. These techniques gave rise to a number of database methods, most notably the kd-B-tree [Rob84], the hB-tree [LS90], and the LSD-tree [HSW89]. Apart from these, which were

discussed in the previous section, the majority of the multidimensional access methods can trace their ancestry to one or more of the following three techniques: the grid file [NHS84], z-ordering [OM84], and the R-tree [Gut85]. We survey briefly the main ideas behind these techniques and their many refinements. For more complete surveys, we recommend the paper by Gaede and Günther [GG98], which focuses on spatial access methods, and also that of Salzberg and Tsotras [ST99] which concentrates on temporal access methods.

2.2.1 The grid file

The grid file is an adaptation of the idea of extendible hashing of Fagin *et al.* [FNPS79]. The original technique was proposed by Nievergelt, Hinterberger and Sevcik [NHS84]. The main idea behind the grid file is as follows: the date space is partitioned by a grid. Each cell of the grid is associated with a single bucket. Buckets can be associated with more than one cell. A query is mapped against the grid, and the buckets corresponding to the resulting cells are fetched. Insertion may cause a cell's bucket to overflow. In this case, the cell must be split. This is done by adding a new grid line, splitting a whole row or column (for 2-d data) of cells. Deletions are harder to handle.

A number of extensions to the basic technique were proposed. We selectively mention Hinrichs [Hin85], who proposed a two-level grid file, and Freeston [Fre87], who proposed the BANG file, which replaced the original grid with a more versatile method of partitioning the space.

2.2.2 Space-filling curves

A space-filling curve is a one-dimensional curve that passes through all points of a d-dimensional space. Consider the discrete planar case; the regular grid $G = [0 : n] \times [0 : n]$ contains $l = (n + 1)^2$ points. A space-filling curve is a bijective mapping f from G to $L = [0 : l - 1]$. We are interested in locality-preserving space-filling curves; that is, for points $p_1, p_2 \in G$ where $\|p_1 - p_2\|$ is small, we wish $|f(p_1) - f(p_2)|$ to be small, with high likelihood.

Given a set S of points in G, we can transform each point $p \in S$ into $f(p)$ and store the set $\{f(p)|p \in S\}$ into a B+-tree, or any other one-dimensional index. Given a query rectangle $r = [x_1 : y_1] \times [x_2 : y_2] \subseteq G$, we wish to retrieve $S \cap r$. Rectangle r can be transformed through f into a sequence of intervals $[a_1 : b_1], \ldots, [a_k : b_k]$, all subsets of L, where k is minimum. Thus, we can pose these k one-dimensional queries to the B+-tree holding our transformed dataset S, and retrieve the answer to our query.

The approach described is appealing both for its simplicity, and for the fact that it is implemented by a B+-tree, which negates the need for special index structures in the database. The drawback of the approach is that the number of one-dimensional queries k can become large, and thus performance may suffer. Thus, it is critical to select an appropriate space-filling curve, that will keep k small most of the time. The most popular candidates are the Peano curve (also known as z-order), proposed initially by Orenstein and Merret [OM84], and the Hilbert curve, proposed originally by Faloutsos and Rong [FR91]. These techniques can easily be extended to dimensions higher than 2.

2.2.3 Storing spatial objects using point access methods

Both the techniques based on the grid file, and those using space-filling curves, are natively capable of storing only multidimensional points. However, in many applications, more general spatial objects, such as lines, rectangles, or arbitrary polygons, must be stored. Of these, rectangles are the most popular, because other kinds of objects can be approximated by their *minimum bounding rectangle* (MBR). In this setting, the queries of interest are intersection queries for a given query rectangle (other types of queries are possible).

A simple approach at storing rectangles in point access methods, is to transform a d-dimensional (hyper) rectangle into a point in $2d$-space. Under this transformation, a rectangular intersection query is transformed into a $2d$-dimensional dominance query. This transformation was employed in the DOT (Double Transformation) technique of Faloutsos and Rong [FR91], in the work of Kanellakis *et al.* [KRV$^+$93] (for $d = 1$), and in a number of other works.

For objects with complicated shape, it may be desirable to decompose them into multiple rectangles, if using the object's MBR provides a crude approximation. Gaede [Gae95] studies empirically the effects of redundancy introduced by this approach.

2.2.4 R-tree and its variants

Perhaps the most popular access method for spatial objects is the R-tree, introduced by Guttman [Gut85]. In contrast to other techniques, the R-tree can store points, rectangles, line segments, and many other geometric objects. Although versatile, the original R-tree suffered from serious performance problems, both with respect to search and with respect to update operations. Thus,

the initial work was followed by a very large number of extensions, each attempting to ameliorate a particular shortcoming of the original method.

The basic idea behind the R-tree, is surprisingly devoid of any geometric, spatial, or other data-dependent considerations. The R-tree cannot fairly be called a spatial access method; rather, it is an ingenious extension of the concept of the B+-tree, based on robust engineering concepts of external memory. This generality was recognized in the Generalized Search Tree (GiST) of Hellerstein, Naughton and Pfeffer [HNP95], where the generality of the R-tree approach was formalized and brought to center stage.

Basic R-tree operation

Consider a dataset S of n objects. The type of objects is intentionally left unspecified. Partition **arbitrarily** the set S into blocks. Each block can be described by a predicate, suitable for the type of objects stored. For example, if rectangles and points are stored, the predicate may be the minimum bounding rectangle for the contents of the block.

If there are B objects per block, roughly n/B blocks are created. Consider the set of n/B predicates describing these blocks as a new dataset, and repeat the process, creating n/B^2 blocks. Keep repeating this process, until a single block is created. The result is a hierarchical structure, a balanced tree, of height $\log_B n$, occupying $O(n/B)$ disk blocks. The original n/B blocks are the leaves of the tree, and the single block of the final phase is the root.

To answer a range query, whose nature is intentionally left unspecified, work as follows; traverse recursively the tree, starting from the root. At each node visited, pose the query against the predicates stored in the node, and descend to those nodes whose predicates are not disjoint with the query.

Insertion is performed by selecting **arbitrarily** a particular path, from the root to a leaf of the tree, where the object is to be placed. If the leaf is not full, insert the object, and adjust the predicates along the selected path. If the leaf is full, split it **arbitrarily** into two leaves, containing a roughly equal number of objects. Propagate the insertion higher in the tree, similarly to the insertion for B+-trees. Deletion of an object is performed along the same lines. In the case of node underflow caused by a deletion, select a sibling of the underflowing node **arbitrarily**, and merge the two nodes into one.

We have emphasized the word **arbitrarily** a number of times in the above description, namely in the policies for tree construction (bulk loading), insertion, and deletion operations. We must qualify this statement carefully. Indeed, correctness of the basic process can be guaranteed by any policy, even one making random choices. However, search performance is very strongly related to the particular choice of policy. Inappropriate policies may result in access methods that "know the value of everything and the cost of nothing" [Aok99]. Despite a vast research effort on appropriate policies, no totally satisfactory candidates have been found, even for restricted application domains. We now survey some of the many proposals, along with the main extensions to R-trees.

The original R-tree

The original R-tree was proposed as a method for storing d-dimensional points and rectangles. The node predicates were the MBRs of the node's contents. Under these choices, one factor affecting performance, is the amount of overlap between the MBRs of the nodes. Guttman [Gut85] proposed a number of different policies, to minimize overlap during insertion, including one with

complexity $O(B)$ (linear split), and one with complexity $O(B^2)$ (quadratic split).

We must remark here, that overlap is itself a heuristic measure of the quality of a particular R-tree. Even when there is no overlap, an R-tree can have bad performance. For example, consider an R-tree storing the points of a regular grid, where each leaf node stores adjacent points belonging to a single row of the grid. Although the leaves do not overlap at all, the performance for even simple queries can be very bad.

Variants of the original R-tree

Kamel and Faloutsos [KF94] introduced the Hilbert R-tree, an elegant variant utilizing the Hilbert space-filling curve. In particular, the original partitioning of the dataset, to create the leaves of the tree, is obtained by first ordering the dataset on the (one-dimensional) Hilbert value of the keys, and then splitting the resulting sequence into blocks. The leaf where a new record is to be inserted is chosen by the same principle. Thus, the Hilbert R-tree can be seen as both an R-tree, and a B+-tree on the Hilbert values of the records.

Other notable variants include the packed R-tree of Roussopoulos and Leifker [RL85], and the sphere tree of Oosterom [Oos90].

The R+-tree

The R+-tree of Sellis, Roussopoulos and Faloutsos [SRF87] attempts to sidestep the performance problems caused by node overlap in the original R-tree. In the R+-tree, there is no overlap between the MBRs of same-level tree nodes. This can be achieved, by replicating each object into every node whose MBR intersects it. This strategy was shown empirically to improve search perfor-

mance. However, it comes with increased cost and complexity for updates. In the simplest case of no node overflow, an inserted object may have to be stored in multiple leaves. When nodes must be split, several complications can arise.

A novel aspect of the R+-tree approach, is the (ad hoc) introduction of redundancy, for the purpose of improving data clustering. This aspect relates to our own thesis, that limited redundancy can be a practical method for performance increase.

The R*-tree

Another interesting technique for addressing the overlap problem, was introduced by Beckmann *et al.* [BKSS90]. Their structure, called the R*-tree, introduces a sophisticated insertion procedure for the R-tree, called *forced reinsert*: when a node overflows, it is not split immediately; instead, a percentage of its contents is removed (the authors suggest an ad hoc value of 30%), and these entries are reinserted one by one into the tree. Even when there is no split, insertion is more sophisticated than the methods of Guttman [Gut85].

The R*-tree was empirically shown to improve search performance by up to 50%, compared to the basic R-tree. Also, space utilization was improved.

The P-tree

In many R-tree variants, the predicates describing tree nodes are (hyper) rectangles. A notable exception is the P-tree of Jagadish [Jag90]. In the P-tree, a node's predicate can be an arbitrary polyhedron. This approach can substantially improve the approximation of the contents of a node. In the experiments reported in [Jag90], it was found that 10-gons are a good choice for two-dimensional data. A drawback of the approach is that 10-gons require

more storage space than rectangles, and thus the degree of the tree nodes is reduced, leading to P-trees that are higher than corresponding R-trees.

R-tree variants for temporal data

A number of R-tree variants have been developed specifically for storing temporal data. In a temporal database, time is often represented as an additional data dimension. In particular, time periods can be represented as intervals on the temporal dimension.

An important technique is the SR-tree proposed by Kolovson and Stonebraker [KS91]. This structure is a combination of R-trees and segment trees. The main idea behind the SR-tree is that very long intervals are stored in internal nodes, instead of leaf nodes, similarly to segment trees. Again, this is done in order to reduce the overlap at the tree leaves.

Another variant for temporal queries is the bitemporal R-tree of Kumar, Tsotras and Faloutsos [KTF98], which is a partially persistent version (see §2.1.9) of the R-tree.

The performance of R-trees

Because of their importance, R-trees have been the target of extensive analytical study. It is well-known that the worst-case access cost of R-trees can be very bad. However, many researchers studied their expected performance, under reasonable assumptions on the distribution of queries and data, and the quality of the clustering.

In a seminal paper, Faloutsos and Kamel [FK94] introduce the concept of fractal dimension, as a way of describing datasets whose distribution departs from the uniform. They provide analytical and empirical results on the positive

effect on performance of datasets with fractal dimension lower than the spatial dimension of the data.

In most other works, the data distribution is assumed to be uniform. Notable analytical works include that of Pagel *et al.* [PST+93], and the work of Theodoridis, Stefanakis and Sellis [TSS00].

2.3 Final remarks

The areas of multidimensional range search and indexing are two of the broadest, multi-discipline areas in computer science. The results presented in this chapter are, by necessity, only a small part of the full volume of work in these areas. We chose to expand only in those works and concepts that play an immediate role in our work, and even so, we had to omit many important contributions. In this section, we attempt to ameliorate this situation, by citing a number of survey articles and books, where the interested reader may find more information.

The amount of theory and techniques on B+-trees is proportional to their paramount importance in databases. Although slightly outdated, the classic survey of Comer [Com79] is a thorough presentation of the main concepts in this area.

External-memory range search is part of the broader area of external memory algorithms. The most important results from this emerging area can be found in a recent survey by Vitter [Vit99].

An active area of mathematics with many applications in geometric range search, is the area of geometric discrepancy. A recent book by Matoušek [Mat99] provides a thorough coverage of the area.

Much of the material surveyed in this chapter is presented more thoroughly in a recent book by Manolopoulos, Theodoridis and Tsotras [MTT99].

Chapter 3

Theory of Indexability

In this chapter we set out a simple framework for describing indexing problems, and for measuring the efficiency of a particular indexing scheme for a given problem. The presented approach was proposed by [HKP97], under the name "theory of indexability".

Indexability has two, relatively distinct parts. First, it introduces a set-theoretic specification and terminology for range-reporting problems. Second, and most important, it proposes a new memory model for range search, and its accompanying performance measures. To a large extent, both of these parts have been adopted before in many relevant settings.

3.1 Indexing Workloads

Access methods must be evaluated in the context of a particular *workload*, consisting of a finite subset of some domain together with a set of queries. More formally, we have the following definition:

Definition 3.1. *A workload W is a tuple $W = (D, I, \mathcal{Q})$, where D is a non-empty set (the* domain*), $I \subseteq D$ is a non-empty finite set (the* instance*), and $\mathcal{Q}$ is a set of subsets of I (the* query set*).*

Example: One of the workloads discussed later, models two-dimensional arrays, where queries are arbitrary subarrays. This workload consists of the domain $\mathbb{R}^2$, the instance $I = \{(i, j) : 1 \leq i, j, \leq n\}$, and the family of "range queries" $Q[a, b, c, d] = \{(i, j) : a \leq i \leq b, c \leq j \leq d\}$, one for each quadruple (a, b, c, d) with $1 \leq a \leq b \leq n, 1 \leq c \leq d \leq n$. Notice that this is a *family of workloads*, with instances of increasing cardinality, one for each $n \geq 0$. Another family of workloads (the set inclusion queries) has as its domain, for each n, all subsets of $\{1, 2, \ldots, n\}$, and for each subset I of the domain, the set of queries $\mathcal{Q} = \{Q_S : S \subseteq \{1, 2, \ldots, n\}\}$, where $Q_S = \{T \in I : T \subseteq S\}$.

In the terminology of combinatorics, W is a simple hypergraph, where I is the vertex set, and $\mathcal{Q}$ is the edge set. We do not use this terminology here, choosing instead to define terms more natural for databases. There is no analog of the domain D in hypergraphs. We could have dropped it from our definition, but it is suggestive of a parameterization of workloads (for example, all two-dimensional range-query workloads have the same domain). It is worth noting that the hypergraph abstraction has been used in related work to measure the quality of existing indexing schemes on particular workloads [SKH99].

3.2 Indexing Schemes

Definition 3.2. *An indexing scheme $\mathcal{S} = (W, \mathcal{B})$ consists of a workload $W = (D, I, \mathcal{Q})$, and for some positive integer B a set $\mathcal{B}$ of B-subsets of I, such that*

$\mathcal{B}$ *covers* I.

We refer to the elements of $\mathcal{B}$ as blocks, and to $\mathcal{B}$ as the set of blocks. We refer to B as the block size. Notice that an indexing scheme is a simple, B-regular hypergraph with vertex set I.

3.3 Performance Measures

We now define two performance measures on indexing schemes, *redundancy* and *access overhead*, which relate to the performance of an indexing scheme in terms of space and query I/O cost, respectively. In both cases, the measures are normalized by the ideal performance (linear space and size of the query, respectively). In the following definitions, let $\mathcal{S} = (W, \mathcal{B})$ be an indexing scheme of block size B on workload $W = (D, I, \mathcal{Q})$, and let $N = |I|$.

3.3.1 Storage Redundancy

Definition 3.3. *The redundancy* $r(x)$ *of* $x \in I$ *is defined as the number of blocks that contain* x*:*

$$r(x) = |\{b \in \mathcal{B} : x \in b\}|$$

The redundancy r of $\mathcal{S}$ is then defined as the average of $r(x)$ over all objects:

$$r = \frac{1}{N} \sum_{x \in I} r(x)$$

It is easy to see that the number of blocks is $|\mathcal{B}| = \frac{rN}{B}$.

3.3.2 Access Overhead

Definition 3.4. *A set of blocks* U covers *a query* Q*, iff* $Q \subseteq \bigcup U$*.*

Definition 3.5. *A cover set* C_Q *for query* Q *is a minimum-size set of blocks that covers* Q*.*

Notice that a query may have multiple cover sets.

Definition 3.6. *The access overhead* $A(Q)$ *of query* Q *is defined as*

$$A(Q) = \frac{|C_Q|}{\left\lceil \frac{|Q|}{B} \right\rceil}$$

where C_Q *is a cover set for* Q*.*

It is easy to see that $1 \leq A(Q) \leq B$, since any query Q will be covered by at least $\lceil \frac{|Q|}{B} \rceil$ and at most $|Q|$ blocks.

We now define the access overhead A of indexing scheme $\mathcal{S}$, to be the maximum of $A(Q)$ over all queries.

Definition 3.7. *The access overhead* A *for indexing scheme* $\mathcal{S}$ *is*

$$A = \max_{Q \in \mathcal{Q}} A(Q)$$

Notice that, although the redundancy is defined as an average (over all data items), the access overhead is a maximum (over all queries). This is less inconsistent and arbitrary than it may seem at first. By averaging over all data items we capture the true space performance of the indexing scheme, while averaging time performance over all queries would be much less defensible; queries are generally not equiprobable, and guarantees, and thus worst-case analysis, are desirable in the context of query response time.

3.3.3 Some Trivial Bounds and Trade-offs

Based on standard properties of databases and disks, we assume that the number of objects N is always much greater than the block size B, although B is not limited in any concrete way.

For any indexing scheme $\mathcal{S}$, the minimum possible redundancy is 1, when $\mathcal{B}$ is a partition of I, and the maximum sensible redundancy is $\binom{N-1}{B-1}$, when $\mathcal{B} = \binom{I}{B}$. For $\mathcal{S}$ having maximum redundancy, A is exactly 1, which is minimum; in that case, every query Q can be covered by a set of disjoint blocks whose union contains Q. Also, for $r = 1$ it is easy to devise a problem where $A = B$, which is maximum (e.g. $\mathcal{Q} = \binom{I}{B}$).

3.4 Discussion

There are many similarities between indexability and previously proposed models. For example, Nodine, Goodrich and Vitter [NGV96], adopt similar cost metrics to study the performance of graph search in external memory. Also, indexability's definition of a workload has been used before, notably by Fredman [Fre80], in the context of studying range search. We discuss issues related to indexability and other competing models for external memory.

A striking aspect of indexability is that it totally omits any aspects of search. For a model applied to range search, this is at least surprising. This is not standard practice for memory models; on the contrary, most memory models focus almost exclusively on search, in the sense that they carefully specify the conditions under which various parts of the memory can be accessed. For example, the RAM model allows access to arbitrary locations, and

allows arithmetic operations on memory addresses. The pointer machine disallows such operations. There, access to a memory location has to be through a special token, the pointer.

In indexability, there are no provisions for modeling the computation of a query's cover set. This is reflected in the definition of indexing schemes, as well as in the definitions of the cost metrics. This omission can be justified in at least two ways.

First, experience with existing access methods indicates that, in practice, search does not introduce additional I/O costs in query processing. In real systems, there is usually enough main memory available, to fit short descriptions for all disk blocks. For example, it is typical for most internal blocks of a B+-tree to reside in a main-memory buffer. Some access methods, such as the grid file, go as far as *requiring* that a catalog fit in main memory. Thus, from a performance point of view, search incurs only CPU costs in practice.

Second, in many range search problems, devising an efficient search scheme seems to be less challenging than devising a good clustering of the data on disk. In fact, given a good clustering, it is often straightforward to construct a search scheme on top of it.

Thus, the omission of search is not as arbitrary as it may seem at first. It offers the advantage of focusing the model on what seems to be the salient aspect of most range search problems, that of locality.

The choice of indexability cost metrics restricts the scope of the model to range reporting problems only. For example, indexability is not suitable for modeling some kinds of search. Notably, nearest-neighbor search under indexability is a trivial problem, because the answer to a query is a single record, and thus the access overhead of every query can be optimal $A = 1$,

with redundancy $r = 1$. The locality issues involved in nearest-neighbor search cannot be captured in this model.

Another limitation incurred by the definition of workload, is that indexability cannot model problems where the answer is a general function of the dataset. Query results are restricted to be subsets of the dataset. Thus, indexability cannot model general range queries over a semigroup.

On the other hand, because of its limitations, indexability is particularly amenable to combinatorial analysis. This is a major advantage, as combinatorics offers a vast arsenal of techniques to a wealth of problems. Thus, despite its limitations, indexability is a useful model for studying a large number of interesting problems.

Chapter 4

Indexing schemes

We now investigate the construction of indexing schemes for some fundamental two-dimensional range search problems. However, we first introduce the indexability model, and some handy definitions, by studying the trivial case of one-dimensional range queries.

4.1 Linear indexing scheme

Let $\mathscr{D}$ be an ordered domain, with $\prec$ as the order relation. Let $\mathcal{I}$ be a finite subset of $\mathscr{D}$. We wish to answer range queries over $\mathcal{I}$, i.e., given $a, b \in \mathscr{D}, a \prec b$, to retrieve all $x \in \mathcal{I}$ such that $a \prec x \prec b$.

Definition 4.1. *A sequence of blocks $b_1, \ldots, b_K$ is linear for order $\prec$, if for all $1 \leq i < j \leq K$,*

$$x \in b_i \wedge y \in b_j \Rightarrow x \prec y$$

When the order relation $\prec$ is implied, we omit it and simply speak of a *linear sequence of blocks*.

Definition 4.2. *A set $\mathcal{I}$ is packed into a linear sequence of blocks iff the sequence has length $\lceil |\mathcal{I}|/B \rceil$.*

Now we estimate the access overhead for packed linear sequences of blocks.

Proposition 4.1.1. *A range query retrieving t elements requires at most $\lceil \frac{t-1}{B} \rceil + 1$ blocks from a packed linear sequence.*

Proof. Fix some t. If some query requires more than $\lceil \frac{t-1}{B} \rceil + 1$ blocks to be answered, it has more than t elements. Indeed, trivially, the (at least) $\lceil \frac{t-1}{B} \rceil + 2$ blocks will form a subsequence of the linear sequence (else some blocks are not required). The first and last of these contribute at least an element each, and the middle ones contribute exactly B elements each. Thus, the query has at least $t+1$ elements. □

By the above proposition we conclude that the access overhead for a packed linear sequence is $A = 2$. Now, consider the case where some linear sequence of blocks is not packed. The following proposition applies:

Proposition 4.1.2. *For a linear sequence of blocks, such that every k adjacent blocks have at least B/l elements each, the access overhead is $A \leq k(l+1)-1$.*

Proof. Let a query require c blocks. For some m and u, $0 \leq u < k$, $c = km+u$. The number of elements t returned by the query is $t \geq mB/l + u$. We have

$$\begin{aligned} t &\geq \frac{B}{kl}km + u \\ &= \frac{B}{kl}c - \Big(\frac{B}{kl} - 1\Big)u \end{aligned}$$

Thus, $c \leq kl\frac{t}{B} + u$, which proves the proposition. □

Linear sequences with the property of the above proposition are frequently used as parts of indexing schemes.

4.2 Interval queries

At first glance, intervals do not appear to be two-dimensional objects, since they represent ranges of a totally ordered domain. However, it is often helpful to identify an interval with a 2-d point on the plane, and a set of intervals with a set of 2-d points. We shall consider open intervals (x, y), where x is the startpoint and y is the endpoint. We call x and y the bounds of the interval.

For a set of open intervals $\mathcal{I}$, we wish to retrieve all intervals $(x, y) \in \mathcal{I}$ which intersect a given query interval (a, b). Two intervals intersect iff

$$x < b \wedge a < y \tag{4.1}$$

or, in words, iff the startpoint of each is before the endpoint of the other. A useful observation is that Eq. 4.1 can be decomposed into two conditions:

$$(a < x < b) \vee (x < a < y) \tag{4.2}$$

The left term of the disjunction is simply a range query on the startpoint of the intervals. The right term is called a point-enclosure, or *stabbing query*: given a point, retrieve all intervals that enclose it. Thus, the interval intersection problem reduces to the stabbing query problem. A geometric representation of interval queries is shown in Fig. 4.1, where the decomposition of an interval intersection query becomes plainly visible.

Stabbing queries are a rather simple workload, and can serve as a good example with which to demonstrate some of the basic techniques for constructing indexing schemes. For this reason, we shall provide multiple solutions,

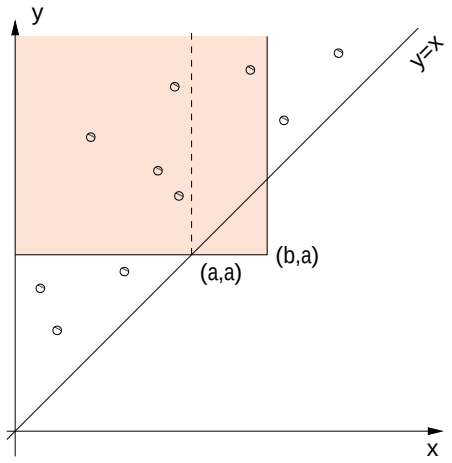

Figure 4.1: Interval intersection as two-dimensional range search. Each point (x, y) corresponds to an interval. Query interval (a, b) corresponds to the upper-left quadrant of point (b, a). A stabbing query at point a corresponds to the upper-left quadrant of point (a, a).

which will enable us to introduce techniques that will be applied later to more challenging problems.

4.2.1 The persistence approach

The concept of persistence was discussed in §2.1.9. Persistent data structures allow queries not only on the current state of a data structure, but on past states as well. Here, we provide a solution to the interval stabbing problem based on this technique.

Consider the *dynamic set maintenance* problem, defined as follows: we wish to dynamically maintain a set of blocks which stores a dynamic set of keys, i.e., a sequence of sets of keys, where each set in the sequence is derived from the previous one by either an insertion or by a deletion of a key. At each

time, we wish to retrieve the full set. In a database, a relation which undergoes insertions and deletions is an example of our problem. At each time, we wish to retrieve the whole relation. Mathematically, we can represent this problem by a finite sequence S_t of sets, such that $|S_t \triangle S_{t+1}| = 1$.[1] To ensure that a key is not re-inserted once it is deleted, we require that for each key x, and each three times $t_1 \leq t_2 \leq t_3$,

$$x \in S_{t_1} \wedge x \in S_{t_3} \Rightarrow x \in S_{t_2}$$

Observe that a persistent solution for the above problem is in fact a (static) solution for interval intersection. Let $\mathcal{I}$ be a set of n intervals. We will maintain dynamically a subset of $\mathcal{I}$. The sorted sequence s_i of all $2n$ interval bounds will define the sequence of insertions and deletions. Thus, if s_i is the startpoint of an interval, it corresponds to its insertion, and if s_i is the endpoint it corresponds to its deletion. Clearly, a stabbing query over the set of intervals $\mathcal{I}$ corresponds to a query of some state of the dynamic set.

We now analyse a simple solution for persistent dynamic set management, within the indexability framework. The queries of our workload are represented by the sets S_t. Naturally, we will require that the constructed indexing scheme must cover every query S_t with at most $A\lceil|S_t|/B\rceil$ of its blocks.

To construct our indexing scheme, we maintain a set E of *active blocks*. Initially, E contains a single empty block. We also define $\alpha_t(b) = b \cap S_t$ to be the set of *active elements* of block b at time t. We iterate over all insert/delete operations in chronological order. The maintenance of E at time t is done as follows:

[1]For sets A, B, $A \triangle B = A \cup B - A \cap B$ is the symmetric difference of A and B.

Case 1: The operation at t is the insertion of key x.

I1. If some block in E has free space, add x to it.

I2. If no block has free space, but some block b has fewer than $B/2$ active elements (i.e., $|\alpha_t(b)| < B/2$), replace b by a new block containing $a_t(b) \cup \{x\}$.

I3. If none of the above conditions apply, create a new block containing x.

Case 2: The operation is a deletion of key x.

D. Let b be the block that contains x. If $|\alpha_t(b)| \leq B/4$, and some other block b' also has $|\alpha_t(b')| \leq B/4$, then remove both b and b' from E and replace them with a new block containing $\alpha_t(b) \cup \alpha_t(b')$.

We claim that the access overhead A of the indexing scheme is small, and also that the redundancy r is small. Our argument is based on the following invariant:

Invariant 4.2.1. *At every time t, the following three apply:*

1. *The union of all blocks in E covers S_t.*
2. *The blocks of E are pairwise disjoint.*
3. *We say that a block* underflows*, iff $|\alpha_t(b)| < B/4$. There is at most one block in E that underflows.*

Proof. The parts of the invariant are easily proved by induction. They are true initially, since there is only one initial empty block in E. Assume they are true

at step t. At step $t+1$, (1) and (2) are trivially true. For (3), let the operation be an insertion; (3) will still be true if **I1** or **I2** apply, since the number of active elements of all blocks can only increase. Also, if **I3** applies, then by the condition of **I3** there was no underflowing block in E, thus the newly created block is the only underflowing block. Finally, if a deletion occurs at $t+1$, then this deletion can cause the block containing the deleted element to underflow. But then, if there is another underflowing block, the two will be immediately merged, and thus there will again be at most one underflowing block. □

Now, we show that the access overhead A is at most 5. To see this, observe that our invariant guarantees that to cover the query corresponding to time t, we need only to use the blocks contained in E at time t. Let K_t be the number of blocks in E at time t. As all but one contain at least $B/4$ active elements, we conclude that

$$|S_t| \geq (K_t - 1)\frac{B}{4}$$

and thus query S_t is covered with an access overhead of at most 5.

The redundancy will also be $r = O(1)$ for our indexing scheme. The argument is as follows: A new block can be created by steps **I2**, **I3** and **D**. In each case, the new block will be at least half-empty. Thus, at least $B/2$ insertions following the creation, can be satisfied by the space in the new block. We conclude that the number u of blocks created by steps **I2** and **I3** is at most $2n/B$. Now, let v be the number of blocks created by step **D**. Since step **D** creates a new block that replaces two previous ones, we can conclude that $v \leq u - 1$. The total number of blocks is $u + v \leq 4n/B$, and thus the redundancy is $r \leq 4$.

In this section we have not attempted to obtain a good indexing scheme, and thus the constants of redundancy and access overhead are high. In subsequent sections, more sophisticated techniques will provide better constants.

4.2.2 A partitioning approach

We now develop a different approach to construct an indexing scheme for interval stabbing. For one-dimensional points, a simple partitioning of the sorted dataset provides an optimal indexing scheme. In this section, we explore a partitioning technique for interval stabbing queries. However, instead of partitioning the dataset, we will partition the set of queries. The origins of this technique can be found in [Cha86].

We begin with some useful definitions.

Proposition 4.2.1. *For a sequence $\{s_i\}$ of $n \geq 1$ elements from an ordered domain $\mathscr{D}$, there exists an element $a \in \mathscr{D}$ such that there are at most $n/2$ elements of $\{s_i\}$ strictly less than a and at most $n/2$ elements of $\{s_i\}$ strictly greater than a. The element a is called a bisector of the sequence.*

Proof. Sort $\{s_i\}$ into sequence $x_1, x_2, \ldots, x_n$. Then, $x_{\lceil n/2 \rceil}$ is a bisector. □

A strictly increasing sequence $\{x_i\}$, $1 \leq i \leq n$, of length $n \geq 0$, defines $n+1$ regions on $\mathscr{D}$. For $n = 0$ the region is $\mathscr{D}$ itself. We denote these regions as

$$r_i = \{x \in \mathscr{D} \,|x_i < x < x_{i+1}\} \ , \ 0 \leq i \leq n$$

(where $x_0 = -\infty$ and $x_{n+1} = +\infty$). We refer to such a set of regions as a *decomposition*, and we say that sequence $\{x_i\}$ *induces* the decomposition.

Consider the dataset as a set $\mathcal{I}$ of n open intervals over some ordered domain $\mathscr{D}$, and some decomposition of $m+1$ regions, induced by $\{x_i\}$, $1 \leq i \leq m$. We say that interval $(x, y) \in \mathcal{I}$ covers region r_i, iff $r_i \subseteq (x, y)$. For $0 \leq i \leq j \leq m$, let $C_{i,j} \subseteq \mathcal{I}$ be those elements of $\mathcal{I}$ which cover all regions $r_i, r_{i+1}, \ldots, r_j$, and let $S_{i,j} \subseteq \mathcal{I}$ be those intervals that intersect some of the regions $r_i, r_{i+1}, \ldots, r_j$. Note that $C_{i,j} \subseteq S_{i,j}$ and also the equations

$$\begin{aligned} S_{i,j} &= \bigcup_{k=i}^{j} S_{k,k} \\ C_{i,j} &= \bigcap_{k=i}^{j} C_{k,k} \end{aligned}$$

For dataset $\mathcal{I}$, for block size B and any $\lambda > 1$, we consider a decomposition induced by $\{x_i\}, 1 \leq i \leq m$, with the following two properties:

$$\lambda\left(|C_{i,i}| + B\right) \geq |S_{i,i}| + B \qquad \text{for } 0 \leq i \leq m \tag{4.3}$$

$$\lambda\left(|C_{i,i+1}| + B\right) \leq |S_{i,i+1}| + B \qquad \text{for } 0 \leq i < m \tag{4.4}$$

For this decomposition, construct an indexing scheme for $\mathcal{I}$ as follows:

1. For each i, $1 \leq i \leq m$, construct packed blocks by storing all intervals that enclose point x_i.

2. Also, construct a packed linear sequence of $\lceil n/B \rceil$ blocks, storing all intervals in ascending order of their startpoint.

A query defined by a closed interval $[a, b]$, is answered as follows: there is a unique i such that $x_i \leq a < x_{i+1}$. Retrieve the blocks of the intervals intersecting x_i (provided $i > 0$), and also retrieve all intervals whose startpoint lies between x_i and b, from the packed linear sequence.

We now use Eqs.(4.3) and (4.4) to analyze this indexing scheme.

Eq. (4.3) suffices to guarantee that the access overhead of any query will be small. Indeed, consider query $[a, b]$ and let i be such that $x_i \leq a < x_{i+1}$. As stated, we will retrieve all $\lceil U_1/B \rceil$ blocks storing the U_1 intervals containing x_i, and also all $\lceil (U_2 - 1)/B \rceil + 1$ blocks of the packed linear sequence, containing all intervals with endpoints between x_i and b. Some of the intervals from these blocks will not belong to the query, namely those intervals (x, y) with $x_i \leq y < a$. There are at most $|S_{i,i}| - |C_{i,i}|$ such intervals. Let t be the number of intervals in the query. Then,

$$U_1 + U_2 \leq t + |S_{i,i}| - |C_{i,i}|.$$

From Eq. 4.3, we get

$$U_1 + U_2 \leq t + (\lambda - 1)(|C_{i,i}| + B),$$

and since $t \geq |C_{i,i}|$, we have

$$U_1 + U_2 \leq \lambda t + (\lambda - 1)B.$$

If V is the total number of retrieved blocks,

$$\begin{aligned} V &= \lceil \frac{U_1}{B} \rceil + \lceil \frac{U_2 - 1}{B} \rceil + 1 \\ &\leq \frac{U_1 + U_2 + 3B - 1}{B} \\ &\leq \frac{\lambda t}{B} + \lambda + 2 \\ &\leq 2(\lambda + 1) \left\lceil \frac{t}{B} \right\rceil \end{aligned}$$

and thus the access overhead is

$$A \leq 2\lambda + 2 \tag{4.5}$$

Eq. (4.4) suffices to guarantee that the redundancy is small. The set of intervals containing point x_i is $S_{i-1,i-1} \cap S_{i,i}$. Thus, the total space required will include the $\lceil n/B \rceil$ blocks of the packed linear sequence, as well as K blocks for storing the containing intervals for each x_i, where

$$K = \sum_{i=1}^{m} \left\lceil \frac{|S_{i-1,i-1} \cap S_{i,i}|}{B} \right\rceil \leq m + \frac{1}{B} \sum_{i=1}^{m} |S_{i-1,i-1} \cap S_{i,i}| \tag{4.6}$$

blocks. To bound the above sum, observe that

$$\begin{aligned} S_{i-1,i-1} \cap S_{i,i} &= (S_{i-1,i-1} \cap S_{i,i} - C_{i-1,i-1}) \cup \\ &\quad (S_{i-1,i-1} \cap S_{i,i} - C_{i,i}) \cup \\ &\quad C_{i-1,i} \\ &\subseteq (S_{i-1,i-1} - C_{i-1,i-1}) \cup (S_{i,i} - C_{i,i}) \cup C_{i-1,i} \end{aligned}$$

(since $S_{i-1,i-1} \cap S_{i,i} \supseteq C_{i-1,i-1} \cap C_{i,i} = C_{i-1,i}$). Notice that $S_{i,j} - C_{i,j}$ is the set of intervals with at least one endpoint in $r_i \cup \ldots \cup r_j$. Thus,

$$\begin{aligned} K &\leq m + \frac{1}{B} \sum_{i=1}^{m} \Big(|S_{i-1,i-1} - C_{i-1,i-1}| + |S_{i,i} - C_{i,i}| + |C_{i-1,i}| \Big) \\ &\leq m + \frac{4n}{B} + \frac{1}{B} \sum_{i=1}^{m} |C_{i-1,i}| \\ &\leq m + \frac{4n}{B} + \frac{1}{B} \sum_{i=1}^{m} \Big(\frac{|S_{i-1,i}| - |C_{i-1,i}|}{\lambda - 1} - B \Big) \\ &\leq \Big(4 + \frac{2}{\lambda - 1} \Big) \frac{n}{B} \end{aligned}$$

from which we conclude that

$$r \leq 5 + \frac{2}{\lambda - 1} \tag{4.7}$$

We must also show that decompositions which satisfy properties (4.3) and (4.4) do exist. The argument is by a two-phase construction, including a *splitting* phase and a *merging* phase.

Assume any initial decomposition, induced by $\{x_i\}$, $1 \leq i \leq n$. Let some region r_i of this decomposition violate Eq. (4.3). Proposition 4.2.1 implies that there exists some $x \in \mathscr{D}$ that bisects the set of $2|S_{i,i} - C_{i,i}|$ endpoints of the intervals in set $S_{i,i} - C_{i,i}$. Use this x to *split* region r_i, and thus replace r_i by two new regions. Repeat the splitting process until all regions satisfy Eq. (4.3). Indeed, the extreme decomposition into $2n+1$ regions, where $S_{i,i} - C_{i,i} = \emptyset$ for all i, does satisfy Eq. (4.3), since $\lambda > 1$, so the splitting process will terminate.

Now, given any decomposition such that all regions satisfy Eq. (4.3), process this decomposition as follows: repeatedly select adjacent regions r_i and r_{i+1} that violate Eq. (4.4). *Merge* these two regions, by replacing them with a new region representing their union. Since the initial decompositions violated Eq. 4.4, the new region satisfies Eq.(4.3). Also, the size of the decomposition has decreased by 1. The extreme case of a decomposition with a single region will trivially satisfy Eq.(4.4). So the merging phase also terminates, and it will yield a desired decomposition.

In general, the order of the phases cannot be reversed. Also, note that the splitting phase can be bypassed by assuming that its end result is the extremal decomposition induced by the ordered sequence of all interval endpoints in the dataset. Then, the merging phase can process the regions serially from (say) left to right, and obtain a desired decomposition.

The preceding analysis is summarized in the following theorem:

Theorem 4.3. *Given set $\mathcal{I}$ of n intervals, and a desired access overhead $A \geq 4$, there exists an indexing scheme for interval intersection queries with redundancy*

$$r \leq 5 + \frac{4}{A-4}$$

The performance of the indexing scheme described in this section is not superior to that of the previous section. Yet, the construction is interesting theoretically, because of the separation of the argument along Eqs. 4.3 and 4.4.

4.3 Intersecting segments

Let $\mathscr{D}_H$ and $\mathscr{D}_V$ be two totally ordered domains. We call $\mathscr{D}_H$ the *horizontal domain*, and $\mathscr{D}_V$ the *vertical domain*. A *horizontal segment* is a triple $(x_1, x_2, y) \in \mathscr{D}_H \times \mathscr{D}_H \times \mathscr{D}_V$, with $x_1 < x_2$. Similarly, a vertical segment $(x, y_1, y_2) \in \mathscr{D}_H \times \mathscr{D}_V \times \mathscr{D}_V$ has $y_1 < y_2$.

The orthogonal segment intersection problem is defined as follows: given a set $\mathcal{I}$ of n horizontal segments, and a query defined by vertical segment (a, b_1, b_2), retrieve all segments $(x_1, x_2, y) \in \mathcal{I}$ such that

$$x_1 < a < x_2 \quad \text{and} \quad b_1 < y < b_2$$

A geometric depiction is shown in Fig. 4.2. In order to provide full generality

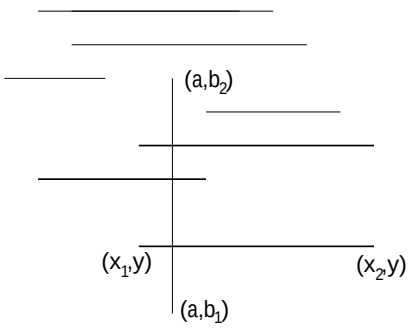

Figure 4.2: A set of horizontal segments, and a vertical segment intersecting three of them.

to our problem, we will assume that both domains $\mathscr{D}_H$ and $\mathscr{D}_V$ contain positive and negative infinities, i.e., each has a minimal and a maximal element. Thus, horizontal and vertical segments can be infinite in one or both directions. With this addition, interval intersection generalizes a number of other problems, among them interval stabbing and interval intersection.

We describe a solution to this problem based on the idea of persistence. That is, we consider the problem of inserting and/or deleting elements from a linear sequence of blocks on the order of $\mathscr{D}_V$. Domain $\mathscr{D}_H$ will serve as the temporal domain. Informally, a horizontal segment (x_1, x_2, y) is added to the linear sequence at a time corresponding to x_1, and is deleted at a time corresponding to x_2.

More precisely, the set of n horizontal segments $\mathcal{I}$ is processed as a sequence $\{s_i\}$ of $2n$ insertion/deletion operations. To horizontal segment (x_1, x_2, y) corresponds an insertion operation $\mathbf{ins}(x_1, x_2, y)$ and a deletion operation $\mathbf{del}(x_1, x_2, y)$. Define function τ as:

$$\begin{aligned} \tau(\mathbf{ins}(x_1, x_2, y)) &= (x_1, x_2, y) \\ \tau(\mathbf{del}(x_1, x_2, y)) &= (x_2, x_1, y) \end{aligned}$$

Operations are sorted in $\{s_i\}$ in ascending lexicographic order of τ, i.e.,

$$i < j \Rightarrow \tau(s_i) \dot{\leq} \tau(s_j)$$

(where $\dot{\leq}$ is the lexicographic order over $\mathscr{D}_H^2 \times \mathscr{D}_V$).

To each integer $0 \leq t \leq 2n$ corresponds a state S_t, which is a subset of $\mathcal{I}$. State S_{t+1} is defined by state S_t and operation s_{t+1} as follows:

1. If $s_{t+1} = \mathbf{ins}(x_1, y, x_2)$, let $S_{t+1} = S_t \cup \{(x_1, x_2, y)\}$.

2. If $s_{t+1} = \mathbf{del}(x_1, x_2, y)$, let $S_{t+1} = S_t - \{(x_1, x_2, y)\}$.

A persistent range query is defined as follows: for every t and range (a, b) with $a, b \in \mathscr{D}_V, a < b$, retrieve all elements $(x_1, x_2, y) \in S_t$ with $a < y < b$. Each segment intersection query over $\mathcal{I}$, defined by a vertical segment (τ, a, b), can be transformed to some t and range (a, b), so that the answer to the segment intersection query is the same as the answer the range query (a, b) on S_t. Note that the converse need not be true, if segment endpoints are not in general position.

To construct an indexing scheme, we iterate in order over the sequence of operations $\{s_i\}$. Through the iteration we maintain a linear sequence E of blocks for the order of $\mathscr{D}_V$. Initially, E contains a single, empty block. At time t, the sequence is updated to reflect the insert/delete operation s_t. For a block b, $\alpha_t(b)$ denotes the set of *active* elements of b at time t, i.e., the set $b \cap S_t$. The maintenance of E is done as follows:

1. If $s_t = \mathbf{ins}(x_1, x_2, y)$, locate a block b in the sequence, corresponding to y. If,
   (a) b has space available, add (x_1, x_2, y) to it, else,
   (b) if $|\alpha_t(b)| < B/2$, replace b in E by a block containing $\alpha_t(b) \cup \{(x_1, x_2, y)\}$, else,
   (c) replace b by two blocks, which split evenly between them the set $\alpha_t(b) \cup \{(x_1, x_2, y)\}$.
2. If $s_t = \mathbf{del}(x_1, x_2, y)$, let b be the block b containing (x_1, x_2, y). Then,
   (a) if $|\alpha_t(b)| = 0$ and b is not the only element of E, remove it from E. Else,

(b) if, for one of the (at most) two adjacent blocks of b, say block b', it is $|\alpha_t(b)| + |\alpha_t(b')| \leq B/2$, merge b and b' into a new block containing all active elements of the merged blocks. Replace the merged blocks with the new one in E.

We now show that the blocks produced by the above algorithm, form an efficient indexing scheme for segment intersection. First, observe that the algorithm is correct, i.e., that E is indeed at all times a linear sequence. Also, at every step t, the contents of all blocks subsume S_t. To answer a query (a, b) over S_t, we use the blocks that were contained in E at step t of the construction.

At all times t, every two adjacent blocks in E contain at least $B/2$ active elements. This can be shown by considering the various cases of the construction algorithm. Since E is a linear sequence, by Prop. 4.1.2, the access overhead is $A \leq 6$.

A newly created block (by step 1b,1c or 2b) will have at least $B/2$ free space initially. Thus, the blocks created by steps 1b and 1c can be attributed to at least $B/2$ insertion operations. Since there are n insertion operations, the number of blocks created by insertions is at most $2n/B$. When a block is created by step 2b (merging), the size of E decreases by one, and thus there are at most $2n/b$ blocks created by this step as well. We conclude that $r \leq 4$.

The presented solution can be modified to improve redundancy, to the detriment of access overhead. This is done by modifying the processing of deletions. For some $k \geq 2$ (with case $k = 2$ corresponding to the original), step 2b is modified to be as follows:

2b-1 repeatedly replace each maximal subsequence of E of length at least

k, and such that the blocks in the subsequence collectively contain at most $B/2$ active elements, by a single block containing all their active elements.

Each replacement of the above kind will reduce the length of E by $k-1$, and thus there will be at most $\frac{2n}{(k-1)B}$ blocks produced by this step. However, the access overhead is now (by Prop. 4.1.2) $A \leq 3k$, and thus we obtain the following general result:

Theorem 4.4. *Given a set $\mathcal{I}$ of n horizontal segments, there exists an indexing scheme for segment intersection queries with redundancy*

$$r \leq 2 + \frac{6}{A-3}$$

for any access overhead $A \geq 6$.

4.3.1 Three-sided queries

The three-sided query problem is to index a set of points in the plane such that given a, b, c with $a < b$, to retrieve the points (x, y) such that $a < x < b$ and $y < c$. A geometric representation of this problem is shown in Fig. 4.3. The problem is a special case of segment intersection, but we study it separately because of its practical applications. In particular, we derive improved bounds for redundancy and access overhead.

The improvement comes from a modification of the algorithm for segment intersection. We select the negative direction on the y-axis to be our temporal dimension. That is, each point in the dataset is a segment which *starts* at $-\infty$. This choice enables us to improve the construction of an indexing scheme as follows: instead of starting with an initial state S_0 which is

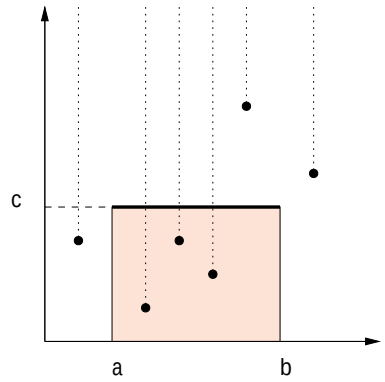

Figure 4.3: A three-sided query over a set of points, defined by limits a,b and c. The points can be thought as vertical segments extending to $+\infty$, and the query as a horizontal segment (a, b, c).

empty, that is, by a linear sequence of blocks E which contains only a single, empty block, we begin with an initial state $S_0 = \mathcal{I}$, i.e., E initially is a *packed* linear sequence, of n/B blocks. The algorithm then constructs additional blocks by processing a sequence of deletions. Since no insertion is performed, the only modification involves step 2b of the process of the previous section, which for $k \geq 2$ is modified as follows:

2b-2 repeatedly replace each maximal subsequence of E of length at least k, such that the blocks in the subsequence collectively contain fewer than B active elements, by a single block containing all their active elements.

Based on the analysis of the previous section, the following theorem can be shown:

Theorem 4.5. *For a set $\mathcal{I}$ of n points on the plane, there exists an indexing scheme for three-sided queries with redundancy*

$$r \leq 1 + \frac{2}{A-2}$$

for any access overhead $A \geq 4$.

Proof. For a chosen value of k, the access overhead is (by Prop. 4.1.2) $A \leq 2k$, and the redundancy is $r \leq 1 + \dfrac{1}{k-1}$. □

The important improvement in the constants of theorem 4.5 is a result of the fact that the process has been reduced to a persistence problem involving only deletions, which implies merging. Merging can be easily modified to guarantee both good space utilization and small additional space overhead. If the choice of time were made opposite, i.e in the positive direction of the y-axis, a sequence of insertions would have be performed, which would not improve the redundancy bounds significantly.

4.4 Four-sided queries

The four-sided query problem is the canonical range search problem in two dimensions: index a set $\mathcal{I}$ of n points on the plane, so that given a query rectangle with lower-left corner (a, b) and upper-right corner (c, d), all points (x, y) such that $a < x < c$ and $b < y < d$ can be efficiently retrieved.

A solution for this problem is based on the solution derived previously for three-sided queries. Our approach is based on the principle of divide-and-conquer. In order to simplify our analysis, we will derive an asymptotic expression of the redundancy, for a fixed access overhead A, but with small hidden constants. We will also assume that numbers are rounded off nicely into integers. This will only affect our analysis by small constant factors.

For some integer $c > 1$ to be determined later, divide the plane into c horizontal stripes, so that each stripe contains n/c points of the dataset. Now,

consider a query whose rectangle intersects more than one of these stripes. This query is decomposed into a number of one-dimensional range queries (at most $c-2$ of them) and into two three-sided queries. This is depicted in Fig. 4.4.

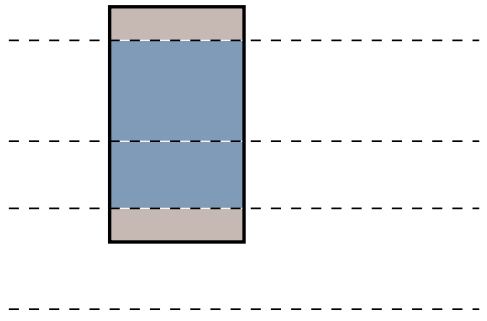

Figure 4.4: The horizontal dashed lines divide the plane into 5 stripes. A rectangle is decomposed into 2 one-dimensional queries and two three-sided queries.

If the points inside each stripe are organized in an indexing scheme which allows three-sided queries in both orientations ($\sqcap$-like and $\sqcup$-like ones), then every query whose rectangle intersects more than one of the stripes can be answered efficiently.

To handle all queries, we recursively divide each stripe into c sub-stripes. This process continues until the stripes contain at most AB points each. Those points are organized into A blocks, and a query falling inside such a stripe will be answered by accessing all A blocks. The depth of this hierarchical subdivision is $\log_c \frac{n}{AB}$.

Each horizontal stripe created by this recursive subdivision must be organized so as to answer both $\sqcap$-like and $\sqcup$-like three-sided queries. Let A_1 be the access overhead for these queries. Note that the indexing scheme for

three-sided queries of §4.3.1 will also answer one-dimensional range queries, as it contains a packed linear sequence over all points. We now estimate constants c and A_1, which yield the desired access overhead A. Let a query which retrieves T points be decomposed into $c-2$ one-dimensional queries, each of which retrieves T_i points, $1 \le i \le c-2$, and two three-sided queries which retrieve $T_\sqcap$ and $T_\sqcup$ points respectively. The number of blocks accessed is at most

$$\begin{aligned} & \sum_{i=1}^{c-2}\Big(\left\lceil \frac{T_i - 1}{B} \right\rceil + 1\Big) + A_1 \left\lceil \frac{T_\sqcap}{B} \right\rceil + A_1 \left\lceil \frac{T_\sqcup}{B} \right\rceil \\ \le \;& 2(c-1) + \frac{1}{B}\sum_{i=1}^{c-2} T_i + A_1 \frac{T_\sqcap}{B} + A_1 + A_1 \frac{T_\sqcup}{B} \\ \le \;& \Big(2(c-2) + 3A_1\Big)\left\lceil \frac{T}{B} \right\rceil \end{aligned}$$

Thus, A_1 and c must be selected so that

$$A = 2(c-2) + 3A_1 \tag{4.8}$$

Now we compute the redundancy for whole indexing scheme. Each of the $\log_c \frac{n}{AB}$ levels of our hierarchical subdivision will contain the whole dataset, and each will require at most

$$\frac{n}{B}\Big(1 + \frac{4}{A_1 - 2}\Big)$$

blocks, as can be seen by Theorem 4.5. The redundancy is thus,

$$r \le \Big(1 + \frac{4}{A_1 - 2}\Big) \log_c \frac{n}{AB} \tag{4.9}$$

We can substitute $A_1 = (A + 4 - 2c)/3$ in the above equation, to obtain

$$r \le \frac{A - 2c + 10}{A - 2c + 2} \cdot \frac{\log(n/AB)}{\log c} \tag{4.10}$$

The value of c which yields minimum redundancy r can be obtained by solving the equation $\dfrac{\partial r}{\partial c} = 0$. After manipulations (see Appendix A) we obtain

$$A = 2(c-2) + 4\sqrt{c \ln c + 1} - 2 \tag{4.11}$$

By contrasting Eqs.(4.8) and (4.11) we see that for minimum redundancy, it must be

$$A_1 = \tfrac{4}{3}\sqrt{c \ln c + 1} - \tfrac{2}{3}$$

Thus, we can assert that $c = \Theta(A)$, and from Eq.(4.9) we have the following theorem:

Theorem 4.6. *For a set $\mathcal{I}$ of n points on the plane, there exists an indexing scheme for four-sided range queries, with redundancy*

$$r = O\Big(\frac{\log(n/B)}{\log A}\Big)$$

The above result is in a sense negative. This is an indexing scheme for which the redundancy grows with the size of the dataset. Unfortunately, as we will show later, the above relationship is optimal.

4.5 Multidimensional Arrays

We now turn our attention to a restricted form of range queries, namely those where the points are arranged on a dense rectangular grid. Another statement of the problem is to index an array, so that subarrays can be accessed efficiently. To make this restricted case more interesting, we consider arrays of some fixed dimension $d \geq 1$.

For a given d-dimensional array with N^d elements ($N \geq B$), we fix parameter c, to be determined later, and we consider all d-tuples of nonnegative integers $(i_1, \ldots, i_d)$, such that $\sum_{k=1}^{d} i_k = \log_c B$. Each such tuple is called a *shape*, and it is used to partition the array into subarrays of dimensions $c^{i_1} \times \cdots \times c^{i_d}$. Each subarray contains B elements, and is thus stored in a block. Overall, the number of blocks K created thus is

$$K = \binom{\log_c B + d - 1}{d - 1} \frac{N^d}{B}$$

since there are $\binom{\log_c B + d - 1}{d-1}$ shapes. The redundancy is therefore

$$r = \binom{\log_c B + d - 1}{d - 1} \tag{4.12}$$

Let $X_1 \times \cdots \times X_d$ denote the dimensions of some subarray that we wish to retrieve. The subarray is covered by blocks of the same shape $(i_1, \ldots, i_d)$, and in particular by a (hyper)cube of $f_1 \times \cdots \times f_d$ of blocks of this shape. Thus, the total number of blocks is

$$f = \prod_{k=1}^{d} f_k \tag{4.13}$$

It is easy to see (by an argument similar to Prop. 4.1.1) that

$$f_k \leq \left\lceil \frac{X_k - 1}{c^{i_k}} \right\rceil + 1$$

and thus

$$\begin{aligned} f &\leq \prod_{k=1}^{d} \left(\left\lceil \frac{X_k - 1}{c^{i_k}} \right\rceil + 1 \right) \\ &\leq \prod_{k=1}^{d} \left(\frac{X_k}{c^{i_k}} + 2 \right) \end{aligned} \tag{4.14}$$

Define the scaling factor λ as

$$\lambda = \left(\frac{\prod_{k=1}^{d} X_k}{B} \right)^{1/d}$$

and let

$$\hat{X_k} = \frac{X_k}{\lambda} \quad \text{for } 1 \leq k \leq d$$

From the definition of λ we have

$$\prod_{k=1}^{d} \hat{X_k} = B \tag{4.15}$$

By substitution in Eq. 4.14 we obtain

$$f \leq \prod_{k=1}^{d} \left(\lambda \frac{\hat{X_k}}{c^{i_k}} + 2 \right) \tag{4.16}$$

We now determine the shape of the blocks as follows: let $\hat{i_k} = \lfloor \log_c \hat{X_k} \rfloor$. By Eq.(4.15), $\sum_{k=1}^{d} \hat{i_k} \leq \log_c B$, thus there exists some shape $(i_1, \ldots, i_k)$ which dominates tuple $(\hat{i_1}, \ldots, \hat{i_d})$ in every coordinate. Pick any such shape. For the chosen shape, the relation $\frac{\hat{X_k}}{c^{i_k}} \leq c$ holds for all k, and by substitution into Eq.(4.16),

$$f \leq (\lambda c + 2)^d \tag{4.17}$$

If $\lambda c < 2$, the query is covered by at most 4^d blocks. Assume $\lambda c \geq 2$. Eq.(4.17) yields

$$\begin{aligned} f &\leq (2\lambda c)^d \\ &\leq (2c)^d \frac{\prod_{k=1}^{d} X_k}{B} \\ &\leq (2c)^d \left\lceil \frac{\prod_{k=1}^{d} X_k}{B} \right\rceil \end{aligned}$$

So, for every access overhead $A \geq 4^d$, we have $c = \frac{A^{1/d}}{2} \geq 2$. The restriction on the access overhead is slightly non-optimal, but cannot be removed for

the type of indexing scheme constructed here. Indeed, consider a query of dimensions $B^{1/d} \times \cdots \times B^{1/d}$, strategically located so that in each dimension (of length $B^{1/d}$) it intersects two block lengths of the shape $(B^{1/d}, \ldots, B^{1/d})$. It can be shown easily that all other shapes will be at least as bad for such a query. Thus, independent of the choice of c, the access overhead will always be $A \geq 2^d$. Our stricter restriction of $A \geq 4^d$ is a result of roundoff steps in our analysis, but removing it would greatly complicate our formulas.

By combining our choice of c with Eq.(4.12), we obtain the following theorem.

Theorem 4.7. *Let W be a workload whose instance consists of the elements of a d-dimensional array of size $N \times \cdots \times N$, and whose set of queries is the set of all subarrays. For any access overhead $A \geq 4^d$, there exists an indexing scheme of redundancy*

$$r = \binom{d\frac{\log B}{\log A - d} + d - 1}{d - 1}$$

For d a fixed constant, the redundancy is

$$r = \Theta\left(\log_A^d B\right) = \Theta\left(\left(\frac{\log B}{\log A}\right)^d\right)$$

but the hidden constant is of order $O(d^d)$. This is an instance of the infamous "curse of dimensionality", where space exponential in d is required to achieve reasonable access overhead.

4.6 Point enclosure queries

The most practical planar range search problem, apart from range queries over points, is intersection queries over rectangles. In this case, the dataset is a set

of rectangles with sides parallel to the axes. A query, defined by a rectangle ρ, must retrieve all rectangles of the dataset that intersect ρ.

The problem can be stated as a two-dimensional interval problem, where each rectangle is defined by two intervals, u_x and u_y, and, given two query intervals q_x and q_y, we must retrieve all rectangles such that u_x intersects q_x and u_y intersects q_y. This view of the problem implies that a query can be decomposed, along the lines of §4.2, into a union of four queries, namely:

- a four-sided range query,
- two segment intersection queries, and
- a point enclosure query.

Of these, all except point enclosure queries have already been studied. Thus, we turn our attention to point enclosure queries, namely, given a dataset of rectangles, and a query defined by a point p, retrieve all rectangles which contain p.

A fundamental observation is that this problem is, in a sense, dual to the four-sided query problem. Let $\mathscr{R}_2$ be the set of all planar rectangles, and let $\sqcap \subseteq \mathbb{R}^2 \times \mathscr{R}_2$ be the intersection relation, i.e., for point $p \in \mathbb{R}^2$ and rectangle $\rho \in \mathscr{R}_2$, $p \sqcap \rho$ holds if, and only if, ρ encloses p. With this relation, the four-sided query problem is defined by a finite set of points $\mathcal{I} \subset \mathbb{R}^2$, and the a query Q_ρ, defined by rectangle ρ, is

$$Q_\rho = \left\{ p \in \mathbb{R}^2 \mid p \sqcap \rho \right\} \cap \mathcal{I}$$

Dually, the point enclosure problem is defined by a finite set of rectangles $\mathcal{J}$, and the query Q_p defined by point p is

$$Q_p = \left\{ \rho \in \mathscr{R}^2 \mid p \sqcap \rho \right\} \cap \mathcal{J}$$

Note that in this sense, the segment intersection problem is dual to itself. Also, the interval stabbing problem, studied previously, is dual to the one-dimensional range search problem. Under this light, let us examine the indexing schemes developed for these problems. The linear indexing scheme of §4.1 partitions the key space into (equal) regions, and stores each region separately (into a block). In contrast, the indexing scheme of §4.2 partitions the *query space*, by partitioning the axis into disjoint intervals.

Inspired by this view of the previous techniques, we now examine the indexing scheme of §4.4, hoping to derive a dual solution for point enclosure queries. That indexing scheme works by defining a sequence of $O(r)$ layers. The layers form a hierarchical subdivision of the *key space*, of degree $O(A)$. Each key is stored *once in every layer*. Conversely, a query is *decomposed into* $O(A)$ *subqueries on a single layer*.

We are then seeking an indexing scheme for point enclosure, with the following characteristics: the indexing defines a sequence of $O(A)$ layers. The layers form a hierarchical subdivision of the *query space*, of degree $O(r)$. Each key is *replicated into* $O(r)$ *copies, on a single layer*. Finally, a query is mapped to *a single region in every layer*.

Having stated our requirements, we now see that a solution derived purely from duality arguments, does indeed exist. For simplicity, we do not repeat the detailed analysis of §4.4, but use asymptotic cost expressions instead.

Using $d-1$ horizontal lines, partition the plane into d horizontal stripes, so that each stripe contains a roughly equal number of rectangle corners. Let a rectangle ρ intersect $k \geq 1$ of these horizontal lines, and thus intersect $k+1$ horizontal stripes. The rectangle will be associated with every stripe it

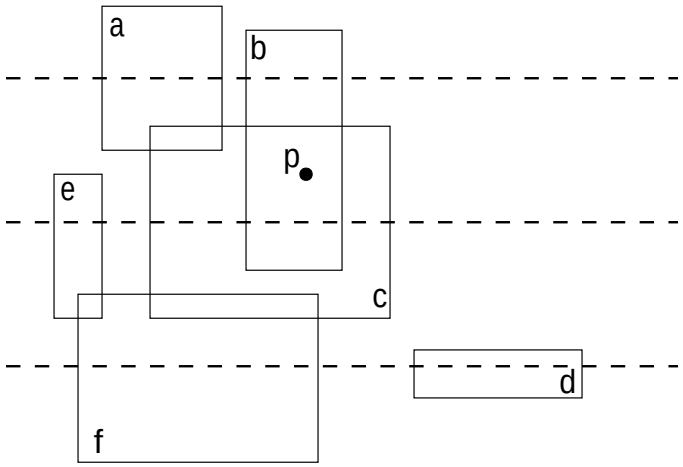

Figure 4.5: A decomposition of the plane into $d = 4$ stripes, by three dashed lines. The rectangles marked a, b, c, d, e, f, intersect at least two stripes each. A query at point p is associated with a single stripe.

intersects. To handle rectangles that do not intersect any of the $d - 1$ lines, we further subdivide each region into d subregions, and continue recursively until the regions contain $O(B)$ rectangle corners, in which case we store the whole region in a single block. Thus, the total number of layers is

$$\log_d(n/B) \tag{4.18}$$

Given a query defined by point p, decompose it into $\log_d(n/B)$ subqueries, one for each layer. The query corresponds to a single horizontal stripe in each layer. Each stripe is organized by splitting the intersecting rectangles into three sets, U, L, and M. U contains all rectangles whose upper side fall inside the stripe. Similarly, L holds all rectangles whose lower side falls into the stripe. M contains the rest, those whose vertical sides span the full height of the stripe. Now, the query at point p within the stripe can be answered as the union of three queries as follows:

- A segment intersection query over the contents of U, where the horizontal segments are the upper sides of the rectangles in U, and the query segment is the infinite vertical upward ray emanating from p.

- Another segment intersection query over the contents of L, where the queries are the lower sides of the rectangles in L, and the query segment is the infinite vertical downward ray emanating from p.

- Finally, an interval stabbing query over the contents of M, where the keys are the one-dimensional projections of the vertical sides of the rectangles of M on the y-axis, and the query point is the y-coordinate of p.

All three types of queries can be answered with $O(1)$ access overhead, using indexing schemes of $O(1)$ redundancy.

For any given query at point p, which retrieves T_i rectangles from layer i, $i = 1, \ldots, \log_d(n/B)$, the total number C of blocks required is

$$C = \sum_{i=1}^{\log_d(n/B)} O(1) \left\lceil \frac{T_i}{B} \right\rceil = O(\frac{T}{B} + \log_d(n/b))$$

and thus, $A = O(\log_d \frac{n}{B})$.

Also, let $s = O(n/B)$ be the number of all stripes in all layers, and let n_i denote the number of rectangles stored in stripe i. Each rectangle is stored in at most d stripes, so the total number K blocks required is

$$K = \sum_{i=1}^{s} O(\frac{n_i}{B}) = O(d\frac{n}{B} + s) = O(\frac{dn}{B})$$

and thus $r = O(d)$.

We conclude that

Theorem 4.8. *For a set $\mathcal{I}$ of n planar rectangles with sides parallel to the axes, there exists an indexing scheme for point enclosure queries, with access overhead*

$$A = O\big(\frac{\log(n/B)}{\log r}\big)$$

Contrast the trade-off of the above theorem with the trade-off

$$r = O(\log_A(n/B))$$

for four-sided queries (from Theorem 4.6). Interestingly, the two trade-offs are dual with respect to A and r. Although the construction of this section has made this development more-or-less expected, as a general rule this is a most amazing fact. We postpone further discussion of this matter, until after we have studied lower bounds on these trade-offs, in chapter 5.

4.7 Discussion

We have studied orthogonal range search on the plane, by constructing indexing schemes for some fundamental problems. The techniques we have used are well-understood techniques from computational geometry, with origins in main-memory range search. We have contributed to this knowledge in two ways; first, we provided external versions of these techniques, that are independent of the search component of the problem, which is the paragon of main-memory analysis. Second, we have in all cases strived to explore the space/access-cost trade-off which is inherent in these problems.

The relationship between locality and search is a fundamental concept highlighted by our work. For main-memory data structures, the issue of locality is equally important, but its expression is more intricate, because it seems

hard to separate it from the issues of search. In indexability, the concept of search is non-existent, and thus the basic combinatorial aspects of locality become more apparent. Equally important, indexability analysis highlights the effect of a variable block size, i.e., an additional parameter to the problem, motivated by technological considerations, which generalizes the concept of storage in ways not relevant to a pure main-memory approach.

The main contribution of indexability is arguably the explicit study of the space/access-cost trade-off. Although this fundamental concept is used widely in databases, it has typically been applied in ad-hoc ways. Within indexability, the issue is brought to center stage, and it receives rigorous study under formal assumptions. This is not a purely theoretical exercise. There are numerous techniques, developed in a main-memory setting, which require super-linear storage, if the access cost is to be low. Such techniques are unlikely to be adopted into database practice, because even a logarithmic factor of storage redundancy can become prohibitively expensive. [2] Thus, if databases are to take advantage of such techniques, there must be opportunity to reduce the storage requirements in a disciplined manner, which does not cancel the desirable access cost advantages of the techniques.

Finally, we note that for all the indexing schemes developed in this chapter, it is straightforward to develop corresponding search structures, and thus derive static indexes for these problems. Static indexes can be useful in databases, but their use is limited in special applications. However, it will turn out that indexing schemes are very useful in deriving dynamic data structures as well. We shall study these issues in some detail in chapter 6, where much

[2] This is not as big a problem in main-memory data structures, which have a short lifetime, and in general work under different operational assumptions.

of the development of this chapter will be used as foundation.

Chapter 5

Lower Bounds on Indexing

We have studied a number of different workloads for various types of two-dimensional range queries. In many of the cases studied in the previous chapter, the access overhead and the redundancy were independent of the size of the dataset. A notable exception was the four-sided range search problem, where the indexing scheme developed in §4.4 had redundancy logarithmic in n/B. Also, the indexing scheme for arrays (§4.5) required redundancy polylogarithmic in $\log B$. It is natural to ask whether these redundancy requirements show a failing of our techniques, or whether the indexing problems are indeed hard.

In this chapter we develop a systematic approach for deriving lower bounds on the trade-off between access overhead and redundancy for a given workload.

5.1 A General Theorem for Lower Bounds

In this section we provide a combinatorial analysis of indexing schemes, which will culminate to our Redundancy Theorem. We first state and prove a set-theoretic result that is of central importance to our proof. Note that this theorem is not specific to indexing schemes, but is in fact a theorem in extremal set theory.

Theorem 5.1. *Let $S_1, S_2, \ldots, S_a$ ($a \geq 1$) be non-empty finite sets, $S = S_1 \cup S_2 \cup \ldots \cup S_a$ be their union, and $L \leq |S|$ be a positive integer. Let k denote the maximum integer such that there exist k pair-wise disjoint sets $P_1, P_2, \ldots, P_k$, so that for all i, $1 \leq i \leq k$,*

1. *$|P_i| = L$, and*
2. *$P_i \subseteq S_j$ for some j, $1 \leq j \leq a$.*

or $k = 0$ if no such sets exist. Then,

$$k \geq \left\lceil \frac{|S| - a(L-1)}{L} \right\rceil \tag{5.1}$$

Proof. Let $P = P_1 \cup P_2 \cup \ldots \cup P_k$. By the maximality of k, for all $1 \leq i \leq a$, $|S_i - P| \leq L - 1$, else we could create an additional set P_{k+1} using L elements from $S_i - P$. Thus,

$$|S| - kL = |(S_1 \cup \ldots \cup S_a) - P| \leq \sum_{i=1}^{a} |S_i - L| \leq a(L-1)$$

from which the theorem follows. □

We now apply the above theorem to the domain of indexing schemes. First we define a new concept, *flakes*.

Definition 5.2. *Let $\mathcal{S} = (W, \mathcal{B})$ be an indexing scheme on workload $W = (I, \mathcal{Q})$. A flake is any set of objects $F \subseteq I$ such that for some query Q and some block b, $F \subseteq Q \cap b$.*

We now have the following lemma on flakes:

Lemma 5.1.1 (Flaking Lemma). *Let $\mathcal{S}$ be an indexing scheme, A be the access overhead, and $2 < \eta < \frac{B}{A}$ be a real number. Then, any query Q with $|Q| \geq B$ will contain at least $(\eta - 2)A\frac{|Q|}{B}$ flakes of size $\lfloor \frac{B}{\eta A} \rfloor$.*

Proof. Choose a cover set for Q, say $C_Q = \{b_1, \ldots, b_a\}$, of size a. Let $S_1, \ldots, S_a$ be defined as $S_i = Q \cap b_i$ for $1 \leq i \leq a$. We have $a = A(Q)\left\lceil \frac{|Q|}{B} \right\rceil \leq A\left\lceil \frac{|Q|}{B} \right\rceil$. From Theorem 5.1 we know that the number k of flakes of size $\lfloor \frac{B}{\eta A} \rfloor$ is at least

$$
\begin{aligned}
k &\geq \left\lceil \frac{|Q| - a(\lfloor \frac{B}{\eta A} \rfloor - 1)}{\lfloor \frac{B}{\eta A} \rfloor} \right\rceil \geq \frac{|Q| - a(\lfloor \frac{B}{\eta A} \rfloor - 1)}{\lfloor \frac{B}{\eta A} \rfloor} \\
&\geq \frac{|Q| - A\left\lceil \frac{|Q|}{B} \right\rceil (\lfloor \frac{B}{\eta A} \rfloor - 1)}{\lfloor \frac{B}{\eta A} \rfloor} \geq \frac{|Q| - A\left(\frac{|Q|}{B} + 1\right)(\lfloor \frac{B}{\eta A} \rfloor - 1)}{\lfloor \frac{B}{\eta A} \rfloor} \\
&\geq \frac{|Q| - A\left(\frac{|Q|}{B} + 1\right)(\frac{B}{\eta A} - 1)}{\frac{B}{\eta A}} \geq \frac{\eta A|Q|}{B} - A\left(\frac{|Q|}{B} + 1\right) \\
&\geq \frac{\eta A|Q|}{B} - 2A\frac{|Q|}{B} \\
&= A(\eta - 2)\frac{|Q|}{B}
\end{aligned}
$$

The last inequality follows from $|Q| \geq B$. ∎

The Flaking Lemma cannot be improved beyond a constant factor. To see this, multiply the number of flakes by the flake size:

$$(\eta - 2)A\frac{|Q|}{B} \cdot \frac{B}{\eta A} = \frac{\eta - 2}{\eta}|Q|,$$

thus, the percentage of elements in the query that participate in some flake is constant, $\frac{\eta-2}{\eta}$.

We now prove a technical tool from extremal set theory, known as Johnson's bound [Joh62]:

Lemma 5.1.2 (Johnson's bound). *Let S be a finite set, and $S_1, S_2, \ldots, S_k$ be subsets of S, each of size at least $\alpha|S|$, such that the intersection of any two of them is of size at most $\beta|S|$. If $\beta < \frac{\alpha^2}{2-\alpha}$, then $k < \alpha/\beta$.*

Proof. The law of inclusion/exclusion implies that

$$\begin{aligned} |S| &\geq |S_1 \cup \ldots \cup S_k| \\ &\geq \sum_{i=1}^{k} |S_i| - \sum_{1 \leq i < j \leq k} |S_i \cap S_j| \\ &\geq k\alpha|S| - \binom{k}{2}|S| \end{aligned}$$

which reduces to

$$\beta k^2 - (2a + b)k + 2 \geq 0 \tag{5.2}$$

If the above inequality is false for some integer l, then it must be $k < l$ (because existence of $l' > l$ sets S_i would imply existence of l sets, a contradiction).

Now if Eq.(5.2) has two real roots, $k_1 < k2$, whose distance is $k_2 - k_1 > 1$, then the interval (k_1, k_2) will contain some integer, and thus it must be $k \leq k_1$.

Let $\Delta = (2\alpha + \beta)^2 - 8\beta$. For the distance of the two roots to be more than 1, it must be

$$\frac{\sqrt{\Delta}}{\beta} > 1$$

thus

$$(2\alpha + \beta)^2 - 8\beta > \beta^2$$

which yields

$$\beta < \frac{\alpha^2}{2 - \alpha}$$

Then, k is smaller than the smaller root, namely

$$k \leq \frac{2\alpha + \beta - \sqrt{\Delta}}{2\beta} = \frac{\alpha}{\beta} + \frac{1}{2} - \frac{\sqrt{\Delta}}{2\beta} < \frac{\alpha}{\beta}$$

which completes the proof. □

We apply Johnson's bound to flakes, to obtain the following lemma:

Lemma 5.1.3 (Packing Lemma). *Let $\mathcal{S}$ be an indexing scheme, A be the access overhead, and $f_1, \ldots, f_k$ be flakes of block b, of size at most $\frac{B}{\eta A}$, and such that for all i, j, $1 \leq i < j \leq k$,*

$$|f_i \cap f_j| \leq \frac{B}{2(\eta A)^2}$$

Then, $k \leq 2\eta A$.

Proof. We apply Lemma 5.1.2 (Johnson's bound) with $\alpha = \frac{1}{\eta A}$ and $\beta = \frac{1}{2(\eta A)^2}$. Since $\beta < \frac{\alpha^2}{2-\alpha}$, we get $k < \alpha/\beta = 2\eta A$. □

We are now ready to state and prove our main theorem.

Theorem 5.3 (Redundancy Theorem). *Let $\mathcal{S}$ be an indexing scheme for workload $W = (I, \mathcal{Q})$, and $Q_1, Q_2, \ldots, Q_M$ be queries, such that for every i, $1 \leq i \leq M$:*

1. *$|Q_i| \geq B$, and*

2. $|Q_i \cap Q_j| \leq \dfrac{B}{2(\eta A)^2}$ *for all* $j \neq i$, $1 \leq j \leq M$.

Then, the redundancy is bound by

$$r \geq \frac{\eta - 2}{2\eta} \frac{1}{|I|} \sum_{i=1}^{M} |Q_i|$$

where $2 < \eta < \frac{B}{A}$ *is any real number such that* $\frac{B}{\eta A}$ *is integer.*

Proof. The proof has two steps. First, we compute the *minimum* number of flakes associated with queries $Q_1, Q_2, \ldots Q_M$. Let this number be f_1. Then we will compute the maximum number of flakes associated with each block. Let this number be f_2. Clearly, there will be at least $K \geq f_1/f_2$ blocks in $\mathcal{B}$.

Step 1 Consider any query Q_i. By the Flaking Lemma, this query is associated with at least $(\eta - 2)A\dfrac{|Q_i|}{B}$ *distinct* flakes of size $\dfrac{B}{\eta A}$. Let F be such a flake. F is not a flake of some other query Q_j, $j \neq i$, because if it were, then it would be a subset of Q_j as well as of Q_i, and thus $|Q_i \cap Q_j| \geq \frac{B}{\eta A} > \frac{B}{2(\eta A)^2}$. We conclude that

$$f_1 = \sum_{i=1}^{M} (\eta - 2)A\frac{|Q_i|}{B} = \frac{(\eta - 2)A}{B} \sum_{i=1}^{M} |Q_i|$$

Step 2: Consider any block b, and let $F_1, F_2, \ldots, F_k$ be the flakes associated with this block. Each flake F_i is of size $\dfrac{1}{\eta A}B$. Also, for two distinct flakes F_i and F_j, $i \neq j$, $|F_i \cap F_j| \leq \dfrac{1}{2(\eta A)^2}B$, by the following argument: If the flakes are associated with the same query, then they are disjoint. If the flakes are associated with different queries, then their intersection is bounded by the intersection of these queries. Thus, we conclude that $f_2 \leq 2\eta A$.

The proof is complete, by the inequality

$$K = \frac{r|I|}{B} \geq \frac{f_1}{f_2} \tag{5.3}$$

which simplifies to

$$r \geq \frac{\eta - 2}{2\eta} \frac{1}{|I|} \sum_{i=1}^{M} |Q_i|$$

□

An intuitive understanding of the Redundancy Theorem can be found by observing that for any set of queries $Q_1, \ldots, Q_M$, it is trivial to construct a set of $\sum_{i=1}^{M} \frac{|Q_i|}{B}$ blocks, i.e., proportional to the total size of the queries, so that each query is answered with $A = 2$. This can be done by packing each individual query separately. The Redundancy Theorem states that, within a constant factor, it may be impossible to improve this naive indexing scheme, if the pairwise intersections of the queries are bounded. A set of such queries is (in some sense) *incompressible*.

We should remark briefly as to the role of parameter η in the analysis so far. Technically, the role of this parameter is to ensure that $\frac{B}{\eta A}$ is integer. Strictly speaking, the parameter can be removed, as for example in the following Corollary of of the Redundancy Theorem:

Corollary 5.1.1. *Let $\mathcal{S}$ be an indexing scheme with access overhead $A \leq \sqrt{B}/4$, and let $Q_1, Q_2, \ldots, Q_M$ be queries, such that for every i, $1 \leq i \leq M$:*

1. *$|Q_i| \geq B$, and*
2. *$|Q_i \cap Q_j| \leq \frac{B}{16A^2}$ for all $j \neq i$, $1 \leq j \leq M$.*

Then, the redundancy is bounded by

$$r \geq \frac{1}{12|I|} \sum_{i=1}^{M} |Q_i|.$$

Proof. Let $\eta_1 = 12/5$ and $\eta_2 = 2\sqrt{2}$. We first show that there exists $\eta \in [\eta_1, \eta_2]$ such that $\frac{B}{\eta A}$ is integer. This follows from $\frac{B}{\eta_1 A} - \frac{B}{\eta_2 A} = (\frac{1}{\eta_1} - \frac{1}{\eta_2})\frac{B}{A} \geq (\frac{1}{\eta_1} - \frac{1}{\eta_2})16 > 1$.

Using such a η in Theorem 5.3, the second premise becomes

$$|Q_i \cap Q_j| \leq \frac{B}{16A^2} = \frac{B}{2(\eta_2 A)^2} \leq \frac{B}{2(\eta A)^2}$$

and the factor $\frac{\eta-2}{2\eta}$ of the conclusion becomes

$$\frac{\eta - 2}{2\eta} \geq \frac{\eta_1 - 2}{2\eta_1} = \frac{1}{12}.$$

□

Thus, it is not necessary for this parameter to appear in lower-bound formulas. Yet, as a matter of form, the existence of η in such formulas does not introduce significant clutter, and we have chosen to retain it.

To apply the Redundancy Theorem in specific workloads, the goal is to select a large number of incompressible queries, whose total size is parameterized by the access overhead A. Then, the Redundancy Theorem immediately yields a lower bound on the trade-off between redundancy and access overhead. In the following, we apply this technique to some problems of interest.

5.2 Multidimensional arrays

In this section we apply the Redundancy Theorem to d-dimensional arrays, where the set of points consists of the elements of an $N \times N \times \ldots \times N$ array, and the set of queries consists of all subarrays. First, we examine the 2-dimensional case, and then we generalize to d dimensions.

5.2.1 2-d arrays

To apply the Redundancy Theorem, we must identify queries $Q_1, Q_2, \ldots, Q_M$, each of size at least B, and with pairwise intersections at most $B/2(\eta A)^2$. We consider only queries of size $c^j \times \frac{B}{c^j}$, for $j = 0, 1, \ldots, \log_c B$. For each aspect ratio, we will partition the $N \times N$ space, obtaining a total of $M = \frac{N^2}{B}(1 + \log_c B)$ queries of size B each. Before we apply the theorem, we compute parameter c.

Let j and j' be integers $0 \leq j < j' \leq \log_c B$, and Q_j and $Q_{j'}$ be queries of dimensions $c^j \times \frac{B}{c^j}$ and $c^{j'} \times \frac{B}{c^{j'}}$ respectively. Their intersection will have dimensions at most $c^j \times \frac{B}{c^{j'}}$, and will contain at most $\frac{B}{c^{j'-j}} \leq \frac{B}{c}$ elements. Thus, we take $c = 2(\eta A)^2$.

We are now ready to apply the Redundancy Theorem. From the theorem,

$$\begin{aligned} r &\geq \frac{\eta - 2}{2\eta}\frac{MB}{N^2} && (5.4)\\ &= \frac{\eta - 2}{2\eta}\frac{1}{N^2}\left(B\frac{N^2}{B}(1 + \log_c B)\right) \\ &= \frac{\eta - 2}{2\eta}(1 + \log_c B) \\ &\geq \frac{\eta - 2}{2\eta}\log_c B \\ &= \frac{\eta - 2}{2\eta}\frac{\log B}{\log(2\eta^2 A^2)} && (5.5) \end{aligned}$$

and thus we have shown that

Theorem 5.4. *Let W be a workload whose instance consists of the elements of an $N \times N$ array, and the set of queries is the set of all subarrays. For any indexing scheme,*

$$r \geq \frac{\log B}{8 \log A + 20}$$

Proof. Straightforward for $\eta = 4$, from Eq.(5.5). □

5.2.2 d-dimensional arrays

We can generalize the above technique to d-dimensional arrays. We consider queries of size B, with dimensions $c^{j_1} \times c^{j_2} \times \ldots \times c^{j_d}$, for all nonnegative integer $j_1, j_2, \ldots, j_d$, such that $\sum_{k=1}^{d} j_k = \log_c B$. For each sequence $j_1, j_2, \ldots, j_d$, we partition the d-dimensional cube into N^d/B subarrays, of dimensions $c^{j_1} \times c^{j_2} \times \ldots \times c^{j_d}$.

In order to select the appropriate value for c, we consider the size of pairwise intersections of rectangles with different dimensions. It is easy to see that $c = 2(\eta A)^2$ is applicable in this case also, guaranteeing that the intersection of any two rectangles will have size at most $\frac{B}{2(\eta A)^2}$.

We also use the well-known fact that the number of distinct sequences of d nonnegative integers, whose sum is n, is given by

$$\binom{n+d-1}{d-1}$$

(cf. Bose-Einstein distribution).

Thus, the total number of queries (each of size B) will be

$$M = \frac{N^d}{B}\binom{\frac{\log B}{\log 2(\eta A)^2} + d - 1}{d-1}$$

and for the redundancy we have

$$r \geq \frac{\eta - 2}{2\eta}\binom{\frac{\log B}{\log 2(\eta A)^2} + d - 1}{d-1}$$

For d a constant, the above quantity is a polynomial of degree $d-1$. Thus, we have shown the following theorem:

Theorem 5.5. *Let W be a workload whose instance is the set of all elements of an N^d array, and the set of queries consists of all subarrays. The redundancy of any indexing scheme of access overhead A is bound by*

$$r = \Omega\left(\binom{\frac{\log B}{\log A} + d - 1}{d-1}\right) = \Omega\left(\left(\frac{\log B}{\log A}\right)^{d-1}\right)$$

The trade-offs obtained for the array workloads do not depend on the size of the instance, but do depend on the block size B. Our results in this section indicate that the indexing scheme derived in §4.5 is optimal, when the dimension d is a fixed constant. Notice that our lower bound implies a constant factor for redundancy which is independent of d. Thus, the $O(d^d)$ hidden constant for the indexing scheme of §4.5 is not known to be necessary. This is typical in the study of high-dimensional range search, where known lower bounds generally do not reflect the "curse of dimensionality".

5.3 Planar Orthogonal Range Queries

We now turn our attention to the general problem of two-dimensional range queries, where the set of points on the plane is the worst possible. The analysis of two-dimensional arrays of the previous section resulted in a $\Omega(\log_A B)$ lower bound on the redundancy, which is weaker than the upper bound of the indexing scheme of §4.4, which is $O(\log_A(n/B))$. Unfortunately, we show that the upper bound of §4.4 is tight. That is, there are planar point sets which are much harder to index than two-dimensional arrays. One such set was used by Chazelle [Cha90a] in his proof of a lower bound for this same problem, in the context of the pointer machine, which we also adopt in this section.

We construct a hard instance $\mathcal{I}$ of n points on the plane, for given n,

block size B, and desired access overhead A. Set $c = \lceil 2(\eta A)^2 \rceil$, and $m = \log_c n$. For every m-ary tuple

$$(v_1, \ldots, v_m) \in \{0, 1, \ldots, c-1\}^m$$

define coordinates

$$x = \sum_{i=1}^{m} v_i c^{i-1} \tag{5.6}$$

$$y = \sum_{i=1}^{m} v_{m-i} c^{i-1} \tag{5.7}$$

and let (x, y) belong in $\mathcal{I}$. In other words, if v_i are though of as c-ary digits, let the set $\mathcal{I}$ contain all $c^m = n$ points of the form

$$(\langle v_1 v_2 \ldots v_{m-1} v_m \rangle, \langle v_m v_{m-1} \ldots v_2 v_1 \rangle)$$

where $\langle s \rangle$ is the integral value of a string s of c-ary digits.

For this planar set of points, we show the following:

Proposition 5.3.1. *A $c^k \times c^l$ rectangle defined by*

$$c^k p \le x < c^k(p+1) \qquad \text{for some } 0 \le p < c^{m-k}$$

$$c^l q \le y < c^l(q+1) \qquad \text{for some } 0 \le q < c^{m-l}$$

contains exactly c^{k+l-m} points.

Proof. Let point (x, y) correspond to m-ary vector $(v_1, \ldots, v_m)$, as defined by Eqs. (5.6) and (5.7). The x-restriction of the rectangle requires that $\langle v_1 v_2 \ldots v_{m-k} \rangle = p$. Similarly, the y-restriction requires that $\langle v_m v_{m-1} \ldots v_{l+1} \rangle = q$. Thus, the remaining $k+l-m$ variables, from v_{m-k+1} to v_l, are free to assume any value in $\{0, 1, \ldots, c-1\}$, and to each assignment to them corresponds a point that belongs to the rectangle. We conclude that the rectangle contains c^{k+l-m} points. □

We now use the above proposition to compute a large number of B-size queries. For a query to have size B, $c^{k+l-m} = B$, or equivalently

$$k + l = m + \log_c B$$

Since $k, l \leq m$, there are exactly $m - \log_c B + 1 = \log_c(n/B) + 1$ legal combinations of k and l, each defining a particular aspect ratio. For each aspect ratio, we partition our dataset with rectangles, producing $c^{2m-k-l} = n/B$ queries of the same aspect ratio. Thus, we have a total of

$$\frac{n}{B}\big(\log_c(n/B) + 1\big) \tag{5.8}$$

queries of size B each.

We verify that these queries intersect in at most B/c places. Indeed, for any two queries, if their dimensions correspond to the same values of k and l, then they are disjoint. Otherwise, let the first query have dimensions $c^{k_1} \times c^{l_1}$ and the second have dimensions $c^{k_2} \times c^{l_2}$. Without loss of generality, assume $k_1 > k_2$ and thus $l_1 < l_2$. Their intersection is a rectangle of dimensions $c^{k_2} \times c^{l_1}$. But $k_2 + l_1 \leq k_2 + l_2 - 1$, and thus by Prop 5.3.1 it contains at most B/c points.

Having checked all the conditions of the Redundancy Theorem, we can now state the following theorem:

Theorem 5.6. *For some set of n planar points, every indexing scheme of access overhead A will have redundancy at least*

$$r \geq \frac{\eta - 2}{2\eta}\left(\frac{\log(n/B)}{\log(2(\eta A)^2 + 1)} + 1\right) = \Omega\left(\frac{\log(n/B)}{\log A}\right)$$

Proof. Straightforward application of the redundancy theorem, using Eq.(5.8), and $c = \lceil 2(\eta A)^2 \rceil$. □

This lower bound matches within a small constant factor the upper bound of Theorem 4.6, and shows that the indexing scheme of §4.4 has optimal redundancy (within a constant factor) for the desired access overhead.

5.4 Point Enclosure Queries

We now turn our attention to the point enclosure problem, defined in §4.6, and study its indexability trade-off. The analysis of this section is somewhat weaker than those of previous sections, in the sense that we restrict the range of n and B over which our analysis will hold. In particular, we will construct a family of hard datasets for this problem, only for the case where the size n of the dataset is $n \geq B2^B$.

Let $N = n/B$. Consider the two-dimensional $N \times N$ grid

$$\{(x, y) \in \mathbb{N}^2 \mid 0 \leq x, y < N\}$$

To keep the formulas short, we define the following constants:

$$\begin{aligned} \alpha &= N^{1/(B-1)} \\ \gamma &= 1 - \frac{1}{2(\eta A)^2} \\ \lambda &= (\alpha N)^\gamma \end{aligned}$$

By a rectangle of size $u \times v$ we mean a rectangle whose horizontal side spans exactly u columns of the grid, and whose vertical side spans exactly v rows. However, the corners of the rectangle are not points on the grid, i.e., with integer coordinates, but lie on arbitrary points off the grid. Fig. 5.1 shows two such rectangles.

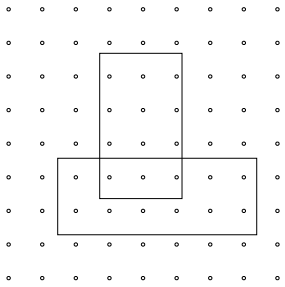

Figure 5.1: A 6×2 and a 3×4 rectangle on the 9×9grid.

The dataset consists of rectangles of sizes $\alpha^i \times \frac{N}{\alpha^i}$, for $i = 0, 1, \ldots, B-1$. Note that since $N \geq 2^B$, it is $\alpha \geq 2$. This last requirement is the reason for our restriction to very large values of n. For each i, a rectangle intersects N grid points, and thus we can tile the $N \times N$ grid with N rectangles of size $\alpha^i \times \frac{N}{\alpha^i}$. Thus, the total number of rectangles is indeed n.

We now select a set of hard queries, in order to apply the Redundancy Theorem. Queries will be identified with points of the $N \times N$ grid, thus each query will have size B (it will contain exactly one rectangle of size $\alpha_i \times (N/\alpha^i)$ for each $i = 0, \ldots, B-1$). Thus, we are free to select any M points on the grid, as long as they satisfy the condition that every two queries must intersect by at most $\frac{B}{2(\eta A)^2} = (1-\gamma)B$. For M such points, the Redundancy Theorem immediately gives a lower bound on redundancy of

$$\frac{\eta - 2}{2\eta} \frac{M}{N} \tag{5.9}$$

The following proposition bounds the number of rectangles stabbed by two grid points.

Proposition 5.4.1. *For any two points (x_1, y_1) and (x_2, y_2) on the $N \times N$*

grid, if,

$$(|x_1 - x_2| + 1)(|y_1 - y_2| + 1) \geq \lambda$$

then the points belong to at most $(1-\gamma)B = \dfrac{B}{2(\eta A)^2}$ *rectangles.*

Proof. Let i_1 be the minimum number such that a rectangle of size $\alpha^{i_1} \times (N/\alpha^{i_1})$ intersects both points. Since the horizontal side of the rectangle must extend at least as much as $|x_1 - x_2|$, we conclude that

$$i_1 = \lceil \log_\alpha(|x_1 - x_2| + 1) \rceil \tag{5.10}$$

Similarly, let i_2 be the maximum integer such that a rectangle of size $\alpha^{i_2} \times (N/\alpha^{i_2})$ intersects both points. It must be

$$\log_a N - i_2 = B - 1 - i_2 = \lceil \log_\alpha(|y_1 - y_2| + 1) \rceil \tag{5.11}$$

For each i, with $i_1 \leq i \leq i_2$, the two points will be enclosed by at most one rectangle of size $\alpha^i \times (N/\alpha^i)$. It follows that we must have $i_2 - i_1 + 1 \leq (1-\gamma)B$. By substituting Eqs.(5.10) and (5.11) we get

$$\lceil \log_\alpha(|x_1 - x_2| + 1) \rceil + \lceil \log_\alpha(|y_1 - y_2| + 1) \rceil \geq \gamma B$$

which follows from the assumption

$$(|x_1 - x_2| + 1)(|y_1 - y_2| + 1) \geq \lambda$$

□

Thus, the problem is reduced in selecting a set of points on the $N \times N$ grid, of maximum size, satisfying the condition of Prop. 5.4.1.

A suitable set of points is the so-called Fibonacci lattice. A lattice defined by vectors $\vec{a}, \vec{b} \in \mathbb{R}^2$ is the set of points

$$\left\{ (u\vec{a} + v\vec{b} \mid u, v \in \mathbb{Z} \right\}$$

The Fibonacci numbers are defined by the well-known recurrence $F_{k+2} = F_{k+1} + F_k$, where $F_0 = 0$ and $F_1 = 1$. A closed formula for Fibonacci numbers is

$$F_k = \frac{1}{\sqrt{5}}\left(\phi^k - (1-\phi)^k\right)$$

where $\phi = (1+\sqrt{5})/2$ is the *golden ratio*. The Fibonacci lattice $\mathbf{F}_k$ of order $k \geq 2$ is the lattice defined by the vectors $(0, F_k)$ and $(1, F_{k-1})$. This lattice was studied by Chor *et al.* [CLRS86] who showed the following theorem:

Theorem 5.7 (Chor *et al.*). *For any two points* $(x_1, y_1), (x_2, y_2) \in \mathbf{F}_{2k+1}$,

$$(|x_1 - x_2| + 1)(|y_1 - y_2| + 1) \geq (F_k + 1)(F_{k+1} + 1)$$

Also, Fiat and Shamir [FS89] have shown that

Theorem 5.8 (Fiat and Shamir). *Every rectilinear oriented rectangle of area* E *intersects the lattice* $\mathbf{F}_k$ *in* $E/F_k \pm \log_\phi(F_k)/3$ *points.*

For the set of queries, we choose the set of points of Fibonacci lattice $\mathbf{F}_{2k+1}$ which fall within the $N \times N$ grid. From Theorem 5.7, it is clear that k must satisfy

$$(F_k + 1)(F_{k+1} + 1) \geq \lambda$$

We use the following well-known Fibonacci identities (e.g. see Vajda's book [Vaj89]):

$$F_{i+j} = F_{i+1}F_j + F_iF_{j-1} \qquad F_n = \phi^i F_{n-i} + (1-\phi)^{n-i}F_i$$

We have

$$\begin{aligned} F_{2k-1} &= F_{2k} - F_{2k-2} \\ &= F_k F_{k+1} + F_{k-1} F_k - F_{k-1} F_k - F_{k-2} F_{k-1} \\ &= F_k F_{k+1} - F_{k-2} F_{k-1} \\ &< (F_k + 1)(F_{k+1} + 1) \end{aligned}$$

Thus, we select k so that

$$F_{2k-3} \leq \lambda \leq F_{2k-1}$$

and we conclude that

$$F_{2k+1} = \phi^4 F_{2k-3} + (1-\phi)^{2k-3} F_4 \leq \phi^4 \lambda$$

(since $(1-\phi)^{2k-3}$ is negative). By Theorem 5.8, the number of points of lattice $\mathbf{F}_{2k+1}$ that intersect the $N \times N$ grid, is approximately N^2/F_{2k+1} and thus at least $N^2/(\phi^4\lambda)$. Thus, from Eq.(5.9), we have the following theorem.

Theorem 5.9. *For some set of $n \geq B2^{B-1}$ planar rectangles with sides parallel to the axes, every indexing scheme for point enclosure queries, of access overhead A, will have redundancy*

$$r \geq \frac{\eta-2}{2\eta\phi^4} \left(\frac{n}{B} \right)^{\frac{1}{2(\eta A)^2}}$$

This trade-off can be written as

$$A^2 = \Omega\left(\frac{\log(n/B)}{\log r}\right)$$

which should be contrasted with the $A = O(\log_r(n/B))$ trade-off of §4.6. It can be seen that the two results differ by a factor of $O(\sqrt{\log_r(n/B)})$, which is

quite significant. However, it must be noted that this factor affects the access overhead, and not the redundancy, and thus is not as severe.

We do not know whether this lower bound, or the upper bound of §4.6 is tight. The issue is perplexed even further by results for this problem in main memory. It is known that this problem can be solved by a (main-memory) data structure of linear size $O(n)$, with a search time of $O(\log n + t)$ (for a query retrieving t points). These bounds are best possible for *any* range search problem, thus the problem is known not to be hard in main memory. Note that neither of our bounds contradict these results, because if one were to set $A = \log(n/B)$ in the trade-off formulas, it would be $r = \Omega(1)$. However, our analysis shows that indeed this problem's storage linearity is only apparent; when the obligatory $O(\log n)$ search factor of the cost is removed, the cost does *not* reduce to $O(t)$, as it does for other types of two-dimensional range search, unless slight polynomial space redundancy is allowed. For practical values of n and A however, this slight non-linearity will not be of any importance.

As a final observation, let us remark on the duality of the lower bound expression between this problem and its dual, the four-sided range search over points. Note that the duality of the two lower bounds is perfect, within a constant factor, if the redundancy r and the expression $2(\eta A)^2$ are exchanged. It begs the following hard question: is this duality a coincidence, specific to these problems and perhaps their extensions in higher dimensions, or is this a fundamental indexability property, for broad classes of problems?

5.5 Set queries

The problem of indexing sets is defined as follows: given an arbitrary set of *sets of keys*, construct an index such that each set can be retrieved efficiently. The sets of keys may be though to correspond to a set-valued attribute of some relation. Today, such queries are answered by a sequence of point queries, i.e., retrieving each key in the set individually. This solution has of course the highest possible access overhead. We study the effect of using redundancy in decreasing the access overhead.

From an indexability point of view, the problem stated above is simply the general indexability problem, and thus under-specified. However, practical considerations can be used to derive a more concrete specification. In particular, we can assume that the queries have small size, where by "small" we mean smaller than the block size B. This assumption has the unfortunate consequence that it disallows use of the Redundancy Theorem. A second consideration is to assume that the set of queries is very large, that is, the set of queries includes all "small" subsets.

Many different workloads can be constructed from the above considerations. An interesting workload is the λ-set workload $\mathcal{K}_{n,\lambda}$, whose instance is the set $\{1, \ldots, n\}$ and whose query set is the set of all λ-subsets of the instance, where $\lambda \leq B$. We show that these workloads are far worse than 2-dimensional queries.

To analyze set workloads, we prove a corollary of the following famous theorem by Turán [Tur41, JW96]:

Theorem 5.10 (Turán's Theorem). *If a simple graph of n vertices has*

more than

$$\frac{(p-2)n^2}{2(p-1)} - \frac{r(p-1-r)}{2(p-1)} \qquad (\textit{ where } r = n \bmod p)$$

edges, then it contains a complete graph of p vertices (a p-clique).

For a given graph, an *independent set* is a subset of its vertices such that there is no edge between any pair of these vertices.

Corollary 5.5.1. *If a simple graph of n vertices has fewer than*

$$\frac{n^2 - n(p-1)}{2(p-1)}$$

edges, then it has an independent set of size p.

Proof. Let $G(V, E)$ be the graph, and let $\tilde{G}(V, \tilde{E})$ with

$$\tilde{E} = \left\{ (v_1, v_2) \in \binom{V}{2} \;\middle|\; (v_1, v_2) \notin E \right\}$$

Then,

$$|\tilde{E}| = \binom{n}{2} - |E| > \frac{(p-2)n^2}{2(p-1)}$$

and thus by Turán's Theorem $\tilde{G}$ has a p-clique. The vertices of the clique form an independent set in G. □

We now show a lower bound for set workloads.

Theorem 5.11. *For workload $\mathcal{K}_{n,\lambda}(I, \mathcal{Q})$, $B \geq \lambda$, any indexing scheme with redundancy*

$$r < \frac{n - \lambda + 1}{(\lambda - 1)(B - 1)}$$

has the worst possible access overhead $A = \lambda$.

Proof. Construct graph $G(I, E)$ where $(x_1, x_2) \in E$ iff there exists a block containing both x_1 and x_2. This graph will have at most

$$r\frac{n}{B}\binom{B}{2} < \frac{n^2 - n(\lambda - 1)}{2(\lambda - 1)}$$

edges. By Corollary 5.5.1, it has an independent set of size λ. This set, taken as a query, will require exactly λ distinct blocks to be covered (by the construction of G). □

The last theorem states that $\mathcal{K}_{n,\lambda}$ requires space at least *quadratic* in n/B to avoid the worst possible access overhead. We show that within a factor of 2, the bound of the theorem is tight.

Theorem 5.12. *For workload $\mathcal{K}_{n,\lambda}$ and $B \geq \lambda$ there exists an indexing scheme of access overhead $A = \lambda - 1$ and redundancy*

$$r = \frac{2n}{(\lambda - 1)B} - 1$$

Proof. We fix an arbitrary partition of the instance into $\lambda - 1$ sets, $S_1, \ldots, S_{\lambda-1}$, of roughly equal size. For each set S_i, we will construct suitable blocks so that for any $x, x' \in S_i$ there is a single block containing both. Then, for every query Q, some elements x_1 and x_2 will belong to the same set S_i, and thus will be covered by a single block, and so $A(Q) \leq \lambda - 1$.

To construct blocks for set S_i, we arbitrarily partition the set S_i into $k = \frac{2n}{(\lambda-1)B}$ sets $t_j, j = 1, \ldots, k$ of size $B/2$ each. For each pair of these sets we construct a block containing their union. Thus, for any pair of elements of S_i, there exists a block containing both.

For each of the $\lambda - 1$ sets S_i we constructed $\binom{k}{2}$ blocks. The total number of blocks constructed thus is

$$(\lambda - 1)\binom{\frac{2n}{(\lambda-1)B}}{2} = \frac{n}{B}\left(\frac{2n}{(\lambda - 1)B} - 1\right)$$

which yields the required redundancy. □

It is suggested from the above analysis, that set workloads are not amenable to indexing with worst-case guarantees.

5.6 Discussion

In the context of indexability, the analysis of indexing schemes focuses not just on the complexity of the indexing problem, but on the space/access-cost trade-off as well. It has been typical for lower bounds analysis, to make some original assumption, e.g. of linear space, or of linear access cost, and then proceed to derive lower bounds on the other parameter. An investigation of the trade-off has been considered more challenging.

It is thus the good fortune of indexability, that its minimalist simplicity allows for the statement of the Redundancy Theorem. This has been demonstrated to be a powerful tool, decoupling almost completely any combinatorial argument of indexing hardness from the underlying geometry of the problem. In all the lower bounds results in this chapter, the main argument has been driven almost completely by geometric considerations, mainly by discrepancy properties [Mat99] of geometric datasets. In all cases, the Redundancy Theorem yielded optimal or close to optimal *trade-off* bounds. This is, we believe, a strong argument of its utility.

5.6.1 Refinements of Redundancy Theorem

It is possible that, in some circumstances, the requirements of the Redundancy Theorem will introduce unwanted restrictions on the parameters of the prob-

lem. In such cases, better results may be achieved by a refinement of the two components of the Redundancy Theorem, namely the Flaking Lemma (5.1.1) and the Packing Lemma (5.1.3). Also, it may be beneficial to alter slightly the size of flakes considered. Briefly, the refined argument could be as follows:

1. By a refined Flaking Lemma, each query corresponds to some minimum number of flakes.

2. By a refined Packing Lemma, each block corresponds to a maximum number of flakes (under a restriction on the pairwise intersections of the flakes).

3. A selected set of queries is used to construct *many* flakes, *few* of which correspond to each block. Thus, a least required number of blocks can be asserted.

In the general case, such modifications will only give very small improvements, of a constant factor at most 2.

Chapter 6

Indexing for 3-Sided Queries

So far, our study of indexability concentrated to variants of planar range search. In this context, we were able to construct indexing schemes for many planar range search problems, and we showed corresponding lower bounds. However, indexing schemes are not index structures, because they do not include a search facility. In addition, indexability analysis does not include any considerations of updates. It might then be—prematurely—concluded that indexability is useful only as a theory for lower bounds, with indexing scheme construction being interesting insofar as it proves those lower bounds to be tight.

In this chapter, we demonstrate such conclusions to be wrong. This is done in two ways; first, we demonstrate that indexing scheme constructions can be used hierarchically to develop search structures, and thus full indexes. Such hierarchical constructs will, however, typically yield only *static* solutions. Database indexes are strongly desired to be *dynamic*, i.e, to allow insertion and deletion of keys. It might then be concluded—again prematurely— that indexability will not be useful in such situations. This conclusion is also shown to

be false. In this chapter, we use indexing scheme construction techniques from Ch. 4 to develop dynamic index structures for planar range search problems.

6.1 Hierarchical Search

In this section we consider the segment intersection problem, and show that there is a natural hierarchical extension that can be used to build a search structure upon an indexing scheme.

Let $\mathcal{I}$ be a set of n horizontal segments. In §4.3 an indexing scheme for segment intersection is constructed, with $r \leq 2 + 6/(A-3)$. The construction follows the idea of persistence; a vertical line sweeps the plane, from $x = -\infty$ to $x = +\infty$. During this sweep operation, a linear sequence E of blocks (§4.1) is maintained, updated with insertions and deletions of horizontal segments, as the sweep line crosses their endpoints. The maintenance of E, by the steps of the algorithm of page 55, is done by the following block-modifying operations:

Add: a new key is inserted into some block of E, at step 1a.

Flush: a block of E is replaced by another block, at step 1b.

Split: a block of E is split into two blocks, at step 1c.

Drop: a block of E is removed, at step 2a.

Merge: two blocks of E are merged into a new block, at step 2b.

A query corresponding to a vertical segment (a, b_1, b_2), is answered by the blocks that are in E at the step which corresponds to a, the x-position of the query segment, and which correspond (since E is a linear sequence) to the

interval (b_1, b_2). Thus, our task is to locate the appropriate blocks, for any query segment (a, b_1, b_2).

Every block-modifying operation listed above, except the first, will actually modify the set of blocks in E, by adding one or two new blocks, which correspond to one or two removed blocks. We can draw a directed graph from the removed blocks to the inserted ones, to denote the dependency between all the blocks in the indexing scheme, as shown in Fig. 6.1. However, this diagram

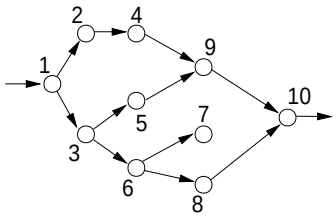

Figure 6.1: The graph shows the set of blocks of an indexing scheme, and the dependencies between them. Blocks 1, 3 and 6 split, blocks 9 and 10 are created by merge, block 2 is flushed, and block 7 is dropped.

does not show the relative order of the various operations. A diagram that shows the relative order of operations is the diagram of Fig. 6.2, which shows a partitioning of the plane into regions, each region corresponding to some block. The region is formed at the x position where the block is created, and extends to the x position where the block is removed from E. The upper and lower sides of the region denote the upper and lower limits of the key range in the block. A block may correspond to more than one region, as for example block 8 does. This happens exactly when some block—in this case block 7—is dropped. Intuitively, this case corresponds to a "dummy" merge between blocks 7 and 8. Thus, the regions can be split into rectangles, whose overall

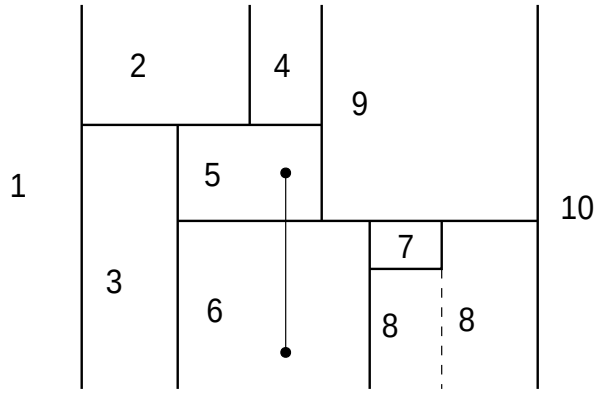

Figure 6.2: This partitioning of the plane corresponds to the dependency graph of Fig. 6.1. Numbers indicate the blocks corresponding to each rectangle. The shown vertical segment corresponds to a query that will be answered by blocks 5 and 6.

number is $O(n/B)$.

Given a vertical segment defining a query, the blocks to be retrieved are exactly the blocks whose regions it intersects, as in the example of Fig. 6.2. We create a new set of horizontal segments, by taking the upper and lower side of each rectangle of each region, totally $O(n/B)$ such segments. This set of horizontal segments can be processed to form a new indexing scheme, of $O(n/B^2)$ blocks. We proceed in this manner, building a hierarchy of $O(\log_B(n/B))$ layers, each describing the blocks of the layer below it. It is not hard to see that given any query defined by a vertical segment, we can recursively descend this hierarchy to search for the blocks that answer the query. The total cost of the search operation will be $O(\log_B(n/B))$. The total space used will be $O(n/B)$ blocks, since each layer has only approximately a fraction of $1/B$ blocks of the layer below it.

Having constructed a search structure for the segment intersection prob-

lem, it is quite straightforward to construct search structures for all planar range search problems in Ch. 4. These search structures will perform *optimal search*, that is to say, will only add a factor of $O(\log_B(n/B))$ to the access cost of the query. In many cases, the resulting index structure will be optimal. However, notice that the structures derived by this technique will, in general, not allow efficient updates.

6.2 Priority Search Trees

We now turn our attention to the construction of an efficient index structure for answering three-sided queries. Our index structure is the external-memory analog of the priority search tree of McCreight [McC85]. For completeness, we briefly describe the main-memory data structure, and then discuss the difficulties of "externalizing" this, and many other similar data structures.

6.2.1 The Priority Search Tree

The priority search tree is a combination of a one-dimensional search tree and a priority queue (or heap). It is an elegant dynamic data structure for main memory, optimally answering three-sided queries (cf §4.3.1) over a set of planar points. Let $\mathcal{I}$ be a set of planar points, and T a regular search tree (e.g., a red-black tree [GS78]) over the x-coordinates of the points. Each node in T is augmented with an additional point, the y-key. The augmentation is done recursively. At every node u of T, the y-key is the key with the least y-coordinate of all keys in the subtree of u, except those that are y-keys in u's ancestors. If there is no such key, the node has no y-key. Thus, the priority search tree requires linear space $O(n)$. Fig. 6.3 shows such a tree over 7 points.

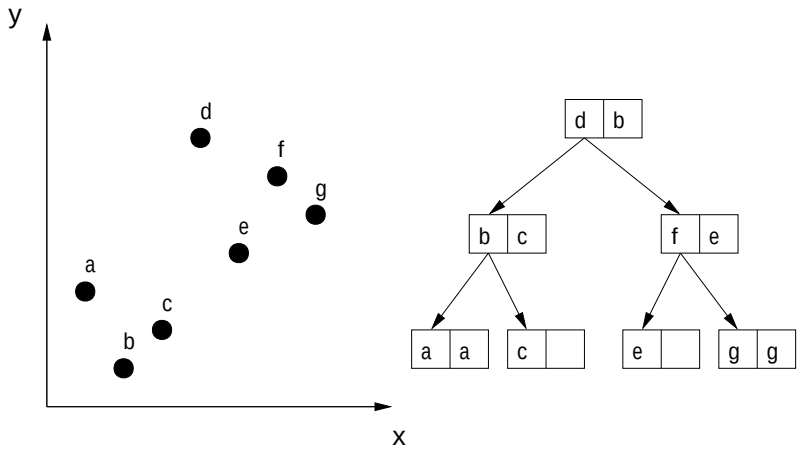

Figure 6.3: The set of 7 points is organized into a Priority Search Tree. Each node of the tree is possibly augmented with the y-key.

Given a tree-sided query (a, b, c), the goal is to retrieve those points (x, y) such that $a < x < b$ and $y < c$. Search proceeds recursively, starting at the root. At each visited node, both keys are reported if they satisfy the query condition. [1] The search stops at any node whose y-key has a y-coordinate higher than c, since all its descendants are guaranteed not to be in the query, or to have been reported already. Of course, the search is also pruned for those nodes that do not satisfy the x-restriction of the query. The asymptotic cost of a search is $O(\log n + t)$, where t is the number of points retrieved by the query. To see this, consider the nodes visited by the search. $O(\log n)$ nodes will be "fringe" nodes, whose subtree only partially satisfies the x-restriction of the query. For the rest of visited nodes, the cost of visiting a node that does not contribute a key in the query is amortized by the fact that its parent node

[1] To avoid reporting a key twice, the x-key is reported only if it is higher than the y-key.

did contribute a point (its y-key).

Updates of the priority search tree are also efficient. Their efficiency is derived by the underlying efficiency of the search tree. Given an insertion or deletion, the tree is updated by a number of rotations of its nodes. For each rotation, a cost of $O(\log n)$ must be paid in order to update the y-keys. However, many search trees, such as the red-black tree, are known to require only a constant number of rotations per update (for the red-black tree, at most 3). Thus, the overall update cost is $O(\log n)$.

6.2.2 Externalization

The asymptotic lower bounds for any range searching index structure are as follows: the total space is $\Omega(n/B)$ blocks, a search costs $\Omega(\log_B n + t/B)$ I/O operations (where t stands for the number of keys returned by the search), and updates require $\Omega(\log_B n)$ operations. These bounds correspond to main-memory bounds of $\Omega(n)$, $\Omega(\log n + t)$, and $\Omega(\log n)$ respectively.

A straightforward approach (e.g. see Diwan *et al.*[DRSS96]) to externalizing any tree-like data structure from main memory, is to partition the nodes of the tree into blocks, so that each block contains a subtree of height roughly $\log B$. Thus, the height of the corresponding external tree is $O((\log n)/(\log B))$, or, as it is usually written, $O(\log_B n)$ (in this context, n is the number of nodes of the main-memory structure, and B is the number of such nodes that fit into a block). Also, the disk space required is $O(n/B)$ blocks. Unfortunately, this simple-minded construction will not, in general, maintain the efficiency of search in main memory. This is true for the priority search tree, but also for other main-memory data structures, such as the

interval tree, the segment tree, etc.

The cause of the inefficiency lies in the high arity of index structures. In a binary tree, the cost of search is kept comparable to the number of reported points, $O(t)$, by amortizing the cost of visiting nodes that do not report points. Since each node has at most two children, the total overhead is within a factor of 2 of t. The nodes of index structures however have $O(B)$ children, and the overhead would be within a factor of $O(B)$. Thus, pruning the search requires much more involved techniques.

One technique addressing this problem is *path caching*, proposed by Ramaswamy and Subramanian [RS94]. Path caching augments each internal index node with additional data blocks, which cluster efficiently paths of the subtree of the node. This is a generic approach to the problem, but has a few drawbacks:

- The disk space is not $O(n/B)$ (linear) anymore.
- Updates are not optimal, because the additional blocks must be updated as well, which is not easy to do.

The techniques introduced in the following sections, overcome these drawbacks.

6.3 External Priority Search Trees

We now turn our attention to the construction of External Priority Search Trees (EPS-trees for short). Through this construction, we introduce a number of new techniques for index structure implementation. To keep the discussion simple, and to better explore the possible design trade-offs, we develop the EPS-tree in stages. In this section, we concentrate on developing a static

EPS-tree, and concentrate on the search and bulk-loading operations. The update techniques are developed incrementally in upcoming sections.

The purpose of this, and upcoming sections, is to describe the construction of a practical index structure. To achieve the highest possible preciseness, a major part of our description will be through the use of C-like pseudo-code.

6.3.1 Modeling the problem

The keys of an EPS-tree are tuples (x, y) from $\mathcal{X} \times \mathcal{Y}$, where $\mathcal{X}$ and $\mathcal{Y}$ are arbitrary, totally ordered types, such as numbers or strings. The index shall only use the ordering predicate from each type, $<_x$ and $<_y$ respectively.

For notational convenience, we assume that each type has special constants $+\infty$ and $-\infty$, representing greatest and least elements respectively. These constants may exist natively in the types, or they may be easily implemented by simple encoding techniques.

We introduce two alternative ordering predicates between keys, $\prec_x$and $\prec_y$, defined as

$$(x_1, y_1) \prec_x (x_2, y_2) \equiv (x_1 <_x x_2) \vee (x_1 = x_2 \wedge y_1 <_y y_2) \tag{6.1}$$

$$(x_1, y_1) \prec_y (x_2, y_2) \equiv (y_1 <_y y_2) \vee (y_1 = y_2 \wedge x_1 <_x x_2) \tag{6.2}$$

This notation has two advantages. First, it introduces the *general position assumption*, i.e, for any pair of distinct keys, p and p', exactly one of $p \prec_x p'$ and $p' \prec_x p$ holds (similarly for $\prec_y$). Second, it allows a single notation to express open and closed bounds of a query constraint. For example, a constraint on key p of the form $a <_x p.x$ is written as $(a, +\infty) \prec_x p$, whereas a constraint of the form $a \leq_x p.x$ is written as $(a, -\infty) \prec_x p$. Using this notation, we modify the statement of the 3-sided query problem as follows:

given **keys** $a = (a_1, a_2)$, $b = (b_1, b_2)$ and $c = (c_1, c_2)$, with $a_2, b_2, c_1 \in \{-\infty, +\infty\}$, retrieve all keys $p = (x, y)$ of the dataset such that

$$a \prec_x p \prec_x b \quad \text{and} \quad p \prec_y c.$$

With this notation, all comparisons in the EPS-tree algorithms are between points, never between point coordinates. We assume that the keys in an EPS-tree are unique. Duplicate keys can be treated by well-known tagging techniques.

6.3.2 General Structure

The EPS-tree is embedded into a B+-tree built on the $\prec_x$-order of the keys. In the B+-tree, all keys are stored in the leaves of the tree, and the internal (non-leaf) nodes of the tree contain discriminator keys, i.e., the keys in the internal nodes need not be actual keys in the dataset. This allows for compression on the internal nodes and can significantly increase the tree's degree. B+-trees are the index structure of choice in most database implementations.

We introduce some notation for B+-tree nodes. Let u be any node in the B+-tree.

- For node u, not the root, $u\uparrow$ denotes the parent of u.
- $\mathcal{I}$ denotes the set of keys stored in the B+-tree.
- For node u, $[u]$ denotes the range of $\mathcal{X}$ which corresponds to u. This range is recursively. For u the root, $[u] = \mathcal{X}$. For u not the root, the discriminating keys stored in $u\uparrow$ (an internal node), determine the range in the obvious way. By a slight abuse of notation, we write $p \in [u]$ for a key p, if its x-coordinate lies in $[u]$.

- For a node u, $\{u\}$ denotes the set of keys stored in the leaves which are descendants of u. Thus, for u a leaf, $\{u\}$ denotes the contents of block u.

- $|u|$ denotes the size of node u, i.e., the number of children for internal nodes, and the number of keys for leaf nodes.

- For internal node u, u^i stands for the i-th child of u.

- For internal node u, u^p (p a key) stands for the (unique) child v of u such that $p \in [v]$.

There are two trivial modifications to the structure of B+-tree nodes. One only applies to leaf nodes and will be described later. The other only applies to internal nodes. Each internal node is associated with its *child cache* (CC). A CC is a small subindex, used to reduce the cost of search. Thus, each internal node of the B+-tree has an additional pointer to its own CC.

In order to describe the search and update procedures of the EPS-tree, we must specify the abstract interface (API) of the CC structures. The actual implementation of the CCs will be postponed for later. To specify the interface to the CCs, we introduce the concept of Y-sets. Y-sets correspond to the y-keys of the (main-memory) priority search tree, as discussed in §6.2.1. Note that, in the context of Y-sets, the term "set" has the meaning of "container of keys", not the usual mathematical meaning. In other words, Y-sets have dynamic state. On the other hand, Y-sets are not explicitly stored as separate containers. Instead, groups of Y-sets are collectively materialized within each CC.

Each node u in the B+-tree, except the root, has its own Y-set, denoted

by $Y(u)$. The Y-set of node u is a subset of $\{u\}$. In addition, the following invariants apply to Y-sets:

1. If node u is a *proper* ancestor of node $v \neq u$, for all $p \in Y(u)$ and $p' \in Y(v)$,
$$p \prec_y p'$$

2. If u is an ancestor of leaf v (possibly $v = u$), and $p \in Y(u)$, and $p' \in v$ does not belong in any Y-set, then
$$p \prec_y p'$$

3. For any node u, $|Y(u)| \leq Y_{\max}$, where $Y_{\max}$ is a parameter of the tree, representing the maximum allowable Y-set size.

Parameter $Y_{\max}$, together with the B-tree's block size B (maximum number of keys per disk page) are the only parameters of the EPS-tree. In practice, $Y_{\max}$ should be comparable to B. Theoretically speaking, the EPS-tree is optimal iff $Y_{\max} = \Theta(B)$. We shall defer practical issues to the experimental section, where we describe the trade-offs associated with $Y_{\max}$. Note however, that for $Y_{\max} = 0$, the EPS-tree is identical to a B+-tree.

With respect to the Y-sets of the ancestors of a node, we say that node u is a *bottom*, and we write $\lfloor u \rfloor$, iff the union of the Y-sets of all ancestors of u (including u), exhausts $\{u\}$. For a bottom node u, all proper descendants of u are also bottom, and their Y-sets are empty.

The final modification to the structure of the underlying B+-tree can now be stated. Each leaf node u maintains efficiently the set $U(u) \subseteq \{u\}$ of keys that belong to no Y-set (including the leaf's own Y-set). Because of the

second invariant for Y-sets, the set $U(u)$ can be maintained by marking the $\prec_y$-least key that does not belong to a Y-set. We denote this key by **LNB(v)**. If all the keys of a leaf belong to some Y-set, **LNB(v)** is set to $(+\infty, +\infty)$ (not a key). This is equivalent to $\lfloor v \rfloor$.

LNB is an acronym for "Least Not Bubbled", where a key is "bubbled" if it belongs to some some Y-set. The choice of term will become apparent later.

We now describe the abstract interface of the CC. The CC of internal

int Ysize(int i**)**	Return the *size* of $Y[i]$, at most $Y_{\max}$.
List Find(Query $Q(a,b,c)$**)**	Return a list of pointers to disk blocks, of minimum size, containing all points p from all Y-sets that belong to query (a,b,c).
boolean bottom(int i**)**	returns `true` iff child i is bottom.
void Insert(Key p**, int** i**)**	Insert key p into $Y[i]$.
void Delete(Key p**)**	Delete p from its associated Y-set.
void SplitSet(int i**, Key** p**)**	Split $Y[i]$ into two Y-sets, containing the points $\prec_x$-left and $\prec_x$-right of p respectively.
void Merge(int i**)**	Merge Y-sets $Y[i]$ and $Y[i+1]$.
Key LeastKey()	Return the $\prec_y$-least key p among all Y-sets
Key GreatestKey(int i**)**	Return $\prec_y$-greatest element of $Y[i]$.
void SplitCache(int i**)**	Returns a new CC containing all Y-sets $Y[j]$ for $j \geq i$. These sets are deleted from the CC.
void MergeCache(CCache C**)**	Append all the Y-sets of CC C to the array of Y-sets.

Table 6.1: The abstract interface (API) of Child Caches, in pseudo-code notation.

node u, denoted by $\mathbf{CC}(u)$, stores a sequence of $|u|$ Y-sets, one for each child of u. The sequence order reflects the x-order of the children of u. Let us denote by $Y[i]$ the i-th Y-set of a CC. Note that, by definition, the Y-sets of a CC are disjoint. The API of CCs is shown in Table 6.1, in pseudo-C notation.

The CC interface definition is straightforward. In this section, we only use the calls **Ysize**, **bottom** and **Find**. The other calls of the CC API are used by the updating techniques, developed in later sections.

6.3.3 Querying the EPS-tree

Because the EPS-tree is structurally a B+-tree (with two trivial modifications), it can answer point and x-range queries exactly like the B+-tree. There is no reduction in efficiency over the B+-tree. Thus, we only concern ourselves with pure 3-sided queries. For query $Q(a, b, c)$, $Q(p)$ denotes a predicate which is true iff key p belongs to the 3-sided range (a, b, c).

Alg. 6.1 defines the procedure **Search**, which is used to report all the points of a 3-sided query. The search process starts at the root and descends recursively down the tree. At an internal node, it will pose the query to the node's CC, and then descend only to those children whose full Y-set was reported, or to the (at most 2) children whose x-range contains the x-restriction points, a and b, of query (a, b, c). At a leaf node, the **lnb** is used to determine which keys have been reported already (since they belong in the Y-set of some parent) and report the rest.

Algorithm 6.1 Recursive tree search for 3-sided queries.

```
1.  Search(Query Q(a,b,c), TreeNode u)
2.  {
3.     if(u is a leaf)
4.        foreach key p ∈ u where (p ⊀_y LNB(v) and Q(p))
5.           Report(p);
6.     else {        // report from CC and recurse
7.        int rep[|u|] = {0,...,0};        // count keys reported per Y-set
8.        foreach block b ∈ CC(u).Find(Q)
9.           foreach key p ∈ b where (Q(p)) {
10.             Report(p);
11.             rep[u^p] ++ ;
12.          }
13.       foreach int i ∈ {1,...,|v|}
14.             where (rep[i] = CC(u).Ysize(i) or u^i = u^a or u^i = u^b)
15.          if(! CC(u).bottom(i)) Search(Q(a,b,c), u^i);
16.    }
17. }
```

6.4 Implementation of Child Caches

The basic idea behind the construction of the Child Cache is the following: the data in the CC is organized as an indexing scheme, using the techniques of §4.3.1. Each CC will hold a dataset consisting of the union of all Y-sets it stores. The maximum number of keys in the CC, is $BY_{\max}$. By Thm. 4.5, we can construct an indexing scheme on these keys, for any access overhead $A \geq 4$ and $r = 1 + 2/(A-2)$. Let us fix $A = 4$. The total number of blocks of the indexing scheme, will be $2Y_{\max}$. We will show that, in order to query this indexing scheme, we only need to store $\Theta(Y_{\max})$ bytes of information. For a suitable choice of $Y_{\max} = \Theta(B)$, this information can fit into a 1-block "catalog". We now present this basic idea in more detail.

6.4.1 Basic structure

A CC consists of a catalog block C, and a number of *data blocks*, which contain key/value pairs, like the leafs of the B+-tree. The data blocks are split into *base* and *internal*. Each data block can fit B keys (the same number as B+-tree leaves).

The base blocks contain all keys of the cache, sorted by $\prec_x$ and packed. Thus, if the cache has n keys, there are $D_b = \lceil n/B \rceil$ base blocks. The number of internal data blocks will be $D_i = D_b - 1$. Since the cache contains at most B Y-sets, and each Y-set has size at most $Y_{\max}$, the maximum number of base blocks is $Y_{\max}$. The purpose of the catalog block C is to store information that describes the data blocks (base and internal). This information is organized as shown in Table 6.2

N	the number of keys in all base blocks.
N_Y	the number of Y-sets in the cache.
$YSize[B]$	array storing the size of each Y-set.
$bot[B]$	a bit vector demarking whether a child is bottom.
N_B	the number of base data blocks.
$b_B[Y_{\max}]$	array storing pointers to the base data blocks.
$b_I[Y_{\max} - 1]$	array storing pointers to the internal data blocks.
R	pointer to a special data block called the root.
$K[Y_{\max} - 1]$	this array stores keys that describe the contents of the base and internal blocks.

Table 6.2: The attributes stored in the catalog block of a child cache.

6.4.2 Construction of a Child Cache

To build the cache on the set of keys in the cache (for all Y-sets), first sort the keys in x-order, and store them into fully packed base data blocks. Record the pointers to these blocks in the b_B array of the catalog. Also, for each base block $b_B[i]$ except $b_B[0]$ ($i > 0$), store into $K[i-1].x$ the x-coordinate of the $\prec_x$-lowest key of the block.

The internal blocks are built by the iteration of Alg. 6.2, which corresponds to the technique of §4.3.1. To build the internal data blocks, af-

Algorithm 6.2 Construct internal blocks of Child Cache.

```
1.  BuildInternal(Block b_B[N])
2.  {
3.     Block S[N] := b_B;
4.     int pos[N] := {0, 1, 2, ..., N − 1};
5.     Key H[N − 1];        // holds merging heights
6.     for(int i = 0 ; i < N − 2 ; i ++)
7.        H[i] := MergingHeight(S[i], S[i + 1]);
8.
9.     while(S.size > 1) {
10.       Key p := Maximum(H, ≺_y);
11.       int i := oneof {j : H[j] = p};
12.       Block M := Merge(S[i],S[i + 1],p);
13.       K[pos[i]].y := p.y;        // update catalog block
14.       b_I[pos[i]] := M;          // update catalog block
15.       Replace S[i], S[i + 1] by M and update H;
16.       Replace pos[i], pos[i + 1] by pos[i + 1];
17.    }
18.    R := S[0];          // the root block
19. }
```

ter having built the base blocks, use a sequence S of blocks (line 3). Initially, the sequence contains all the base blocks, arranged in the sequence

based on their x-position. For each pair of adjacent blocks $S[i]$ and $S[i+1]$, **MergingHeight**($S[i]$,$S[i+i]$) is the key p from one of these blocks such that there are totally exactly $B-1$ keys below p in those blocks. **Merge**($S[i]$,$S[i+1]$,p) returns a new (internal) data block containing all keys of the original blocks not $\prec_y$-greater than p. Fig. 6.4 depicts the result of this process graphically.

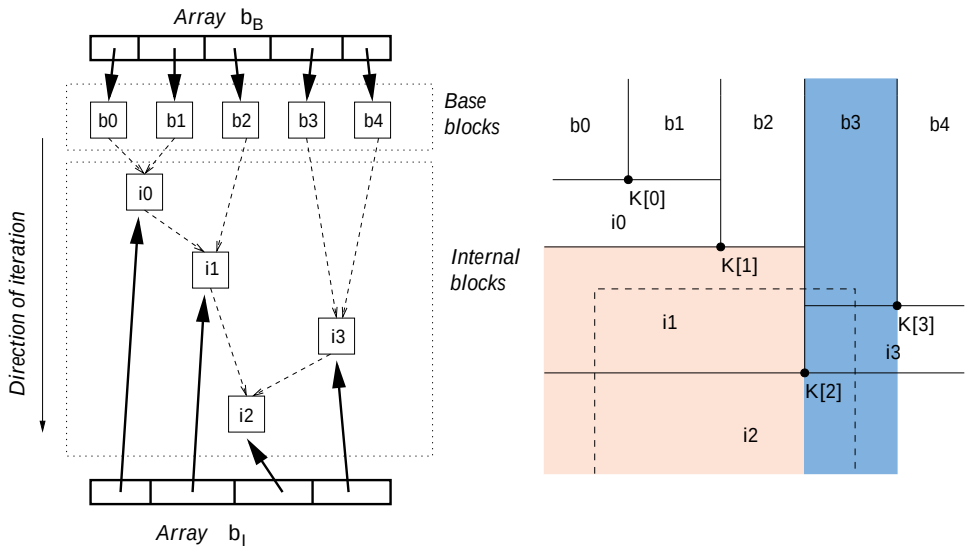

Figure 6.4: Structure of a CC. On the left, the data blocks of the CC, dashed arrows designate merging during construction. On the right, plane partitioning recorded in array K. Dashed line shows a 3-sided query, coverable by blocks i_1 and b_3.

6.4.3 Querying the Child Cache

When a call **Find**($Q(a,b,c)$ is made to the cache, a list of data blocks is computed and returned. These blocks must contain all data relevant to the query. Array $K[i]$ in the catalog block is used to select the fewest possible blocks. The

contents of the $K[i]$ array can be visualized as depicted in Fig.6.4. Alg. 6.3 computes a minimum-size list of data blocks that relate to a given query. The

Algorithm 6.3 Cover query with data blocks.

```
1.  List Find(Query Q(a, b, c)) {
2.  {
3.      List L := ();
4.      if(N = 0) return L; else if(N_B = 1) { L.append(R); return L; }
5.
6.      int low := max{i | j < i ⟹ K[j] ≺_x a};
7.      int high := max{i | j < i ⟹ b ⊀_x K[j]};
8.      Block β := b_B[low];
9.      for(int p := low; p < high; ++ p)
10.         if(K[p] ≺_y c)
11.             { L.append(β);     β := b_B[p + 1]; }
12.         else if (K[p] ≺_y Γ(β))
13.             β := b_I[p];
14.     L.append(β);
15.     return L;
16. }
```

algorithm works by iterating through a slice of array K, determined by the query's x-restriction (a, b). A block pointer β points to the next candidate block to insert into the result. The function $\Gamma(\beta)$, that appears on line 12 of Alg. 6.3, is defined a follows:

$$\Gamma(\beta) = \begin{cases} K[j] & \text{if } \beta = b_I[j] \text{ for some } j; \\ (+\infty, +\infty) & \text{otherwise.} \end{cases} \tag{6.3}$$

The process of Alg. 6.3 has the advantage that it does not explicitly refer to the order of construction of the internal blocks, i.e, the "tree" of Fig. 6.4 (left), but instead infers enough, using only the data in array K. This

is advantageous, because it removes any need of storing any such information for internal blocks, into the catalog block. These space savings allows greater values of $Y_{\max}$, whose range is limited by the capacity of the catalog block. However, it is not obvious that Alg. 6.3 is correct, or that indeed the returned list is of minimum size. Thus, additional explanation is warranted.

Some definitions are necessary. Let β be a block. This block corresponds to some planar region $\rho(\beta)$, of the plane partitioning of Fig. 6.4, which is determined by array K. The *x-span of β*, $\chi(\beta)$, is simply the interval of the x-axis corresponding to the horizontal side(s) of $\rho(\beta)$. Also, the *height of β* is the y-coordinate of the top side of $\rho(\beta)$, or $+\infty$ if β is a base block whose region $\rho(\beta)$ is unbounded on top. Note that the height of β is equal to the y-coordinate of $\Gamma(\beta)$, defined in Eq. 6.3. Finally, the *extent of β*, $E(\beta)$, is the (open) rectangular region $\chi(\beta) \times (-\infty, \Gamma(\beta))$. In Fig. 6.4, the two shaded regions correspond to $E(i_1)$ and $E(b_3)$.

We first discuss correctness, i.e, that the list of blocks returned indeed covers the query. Let $Q(a, b, c)$ be some query, and $L = \beta_1\beta_2 \ldots \beta_m$ be the list returned by Alg. 6.3. Using the above definitions, we can see that correctness can be assured by the following sufficient properties:

1. The x-region of the query, (a, b), must be covered everywhere, i.e,

$$\bigcup_{\beta \in L} \chi(\beta) \supseteq (a, b)$$

2. The height of each block is higher than the query height, i.e.,

$$\forall i, \quad 1 \leq i \leq m, \quad c \prec_y \Gamma(\beta_i)$$

The second property is easy to show; it follows from the loop invariant

$$c \prec_y \Gamma(\beta) \tag{6.4}$$

Trivially, this holds when β is a base block. For β an internal block, this is assured by the combined effect of the **if** conditions, when execution reaches line 13. Since L contains only blocks that β pointed to at some instant, the property is assured.

We now prove the first property. An internal block β is a *derivative* of a (base or internal) block β', if $\beta = \beta'$, or if β was constructed (by Alg. 6.2) by merging on some derivative of β'. This is denoted by $\beta' \rightarrow \beta$. Also, define

$$\beta \longleftrightarrow \beta' \equiv \beta' \rightarrow \beta \vee \beta \rightarrow \beta'.$$

The following properties are easy to see:

$$\beta' \rightarrow \beta \iff \chi(\beta') \subseteq \chi(\beta) \tag{6.5}$$

$$\beta' \not\leftrightarrow \beta \iff \chi(\beta) \cap \chi(\beta') = \emptyset \tag{6.6}$$

Now, the main observation is that each internal block is "surrounded" in array b_I by the blocks that derive it. Formally, for any $0 \leq i < j < k < N_B - 1$,

$$b_I[i] \rightarrow b_I[k] \implies b_I[j] \rightarrow b_I[k], \text{ and} \tag{6.7}$$

$$b_I[k] \rightarrow b_I[i] \implies b_I[j] \rightarrow b_I[i]. \tag{6.8}$$

Similarly, the base blocks that derive an internal block β, are compactly stored in array b_B; for any $0 \leq i < j < k < N_B$,

$$b_B[i] \rightarrow \beta \wedge b_B[k] \rightarrow \beta \implies b_B[j] \rightarrow \beta. \tag{6.9}$$

The above properties are easily shown inductively from Alg. 6.2, line 16.

Now, consider the values of loop variable p, when execution enters line 11. Using Eqs. (6.5) to (6.9), it can be seen that

$$\bigcup_{\beta \in L} \chi(\beta) \supseteq \bigcup_{p=\text{low}}^{\text{high}} \chi(b_B[p]) \supseteq (a, b)$$

Let us now examine whether the list returned by Alg. 6.3 is of minimum length. Note that a minimum-length list is not unique in general. For example, for the space partitioning shown in Fig. 6.5, there is no list of 1 block covering

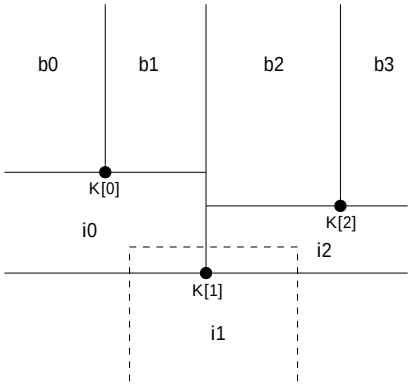

Figure 6.5: An example of a query with multiple minimum lists that cover it.

the shown query, but there are 4 lists of 2 blocks each, namely, b_1b_2, i_0b_2, b_1i_2 and i_0i_2. Alg. 6.3 will, for this case, return b_1b_2. We should clarify that, by a minimum-size list we do not mean minimum-size among all lists that cover the *keys* of the query, but minimum-size among all lists such that

$$\bigcup_{\beta \in L} E(\beta) \supseteq Q,$$

in other words, it is required that the extents of blocks must cover the query's *region*. As an example, consider Fig. 6.5. Assume that the small areas $\rho(i_0) \cap Q$ and $\rho(i_2) \cap Q$ do not contain any points. Then, the shown query could be covered by i_1 alone. However, we do not consider i_1 a legal cover.

We now sketch the proof that L (returned by Alg. 6.3) is of minimum size (in the above sense). Suppose that a strictly shorter cover exists, and

take an minimum-size such cover, say L_0. It is not hard to see that optimality implies that for $\beta, \beta' \in L_0, \beta \neq \beta'$, it must be $\beta \nleftrightarrow \beta'$.

Some reflection shows that, for some block of L_0, say β, region $\chi(\beta)$ would be covered by more than one block in L, say β_1, β_2. But then, from Eq. 6.5, $\beta_1 \rightarrow \beta$ and $\beta_2 \rightarrow \beta$. However, in this case, it must be $\Gamma(\beta) \prec_y c$, (follows from the **if** conditions). Thus, block β is extraneous in L_0, since its span $\chi(\beta)$ must be covered by other block $\beta' \in L_0$, and thus $\beta' \rightarrow \beta$. This contradicts the optimality assumption for L_0.

The final conclusion follows from Thm. 4.5. The size of L is at most $4\lceil t/B \rceil$, where t is the number of keys in the CC, contained in Q.

6.5 Search cost of the EPS-tree

Having completed the analysis of CC query performance, we return to the EPS-tree, and study the I/O cost of queries, by computing the total cost of procedure **Search**, shown in Alg. 6.1.

Let $Q(a, b, c)$ be some query, and let t be the number of points reported for Q. Let procedure **Search** visit k internal nodes, $u_1, \ldots, u_k$. For each visited internal node u_i, let f_i be the number of blocks returned by the call to **Find** (on line 8 of Alg. 6.1). Also, let t_i be the number of points reported from these blocks. Finally, let c_i be the number of children of u_i that are visited.

Let us consider the total number of I/Os. The total number of B+-tree nodes visited is $1 + \sum_{i=1}^{k} c_i$. To see this, observe that $\sum_{i=1}^{k} c_i$ counts every visited node except the root. Also, for each a visited internal node u_i, the call to the CC will require $f_i + 1$ I/Os (the number of pages returned by $\mathbf{CC}(u_i)\mathbf{.Find}(Q)$, plus the catalog block). Thus, the total number of accessed

pages is

$$C = k + \sum_{i=1}^{k} f_i + 1 + \sum_{i=1}^{k} c_i \tag{6.10}$$

Now, from the analysis of the CC access of §6.4.3, we have that $f_i \leq 4\lceil t_i/B \rceil$. Thus,

$$C \leq 5k + 4\frac{t}{B} + 1 + \sum_{i=1}^{k} c_i \tag{6.11}$$

Also, $k \leq 1 + \sum_{i=1}^{k} c_i$, thus,

$$C \leq 4\frac{t}{B} + 6 + 6\sum_{i=1}^{k} c_i \tag{6.12}$$

We now focus on the execution of procedure **Search**, for some internal node u_i, and in particular on estimating c_i, the number of children of u_i visited. The condition for visiting a child node of u_i appears on line 14 of Alg. 6.1. There are two cases:

- First, a child v is visited if $a \in [v]$ or $b \in [v]$. Let h_i denote the number of such children (trivially, $h_i \leq 2$).

- Second, for a child not in the above case, it is visited if its full Y-set is reported in query Q. Let e_i be the number of such children.

We thus have $c_i = h_i + e_i$. It is easy to see that, if $H \leq \lceil \log_{B/2}(n/B) \rceil \leq \log_B n + 1$ is the height of the B+-tree, it is

$$\sum_{i=1}^{k} h_i \leq 2H - 1 \tag{6.13}$$

and thus, Eq. 6.12 becomes

$$C \leq 4\frac{t}{B} + 2\log_B n + 7 + \sum_{i=1}^{k} e_i \tag{6.14}$$

Let $v_1, \ldots, v_{e_i}$ be the children of u_i visited because their full Y-set was reported, and let $y_{ij}, 1 \leq j \leq e_i$ be the number of keys from $Y(v_j)$ reported.

Assume that for some $\delta > 0$, and for all u_i,

$$\delta B e_i \leq \sum_{j=1}^{e_i} y_{ij} \tag{6.15}$$

By $\sum_{j=1}^{e_i} y_{ij} \leq t_i$ and the above condition, Eq. 6.14 yields

$$C \leq 2 \log_B n + 7 + (4 + 1/\delta)\frac{t}{B} \tag{6.16}$$

A static EPS-tree can be constructed so that $y_{ij} = Y_{\max}$. Then, $\delta = Y_{\max}/B$, and, if $Y_{\max} = \Theta(B)$, we have $C = O\big(\log_B n + t/B\big)$.

Thus, optimality of search cost reduces to the condition of Eqs. 6.15–6.16. These two equations reveal the main idea behind our approach; amortize the cost of extending the search to a node's children, on the number of query outputs produced by the Y-sets of these nodes. The degree of amortization is expressed by δ, which affects the access overhead by a factor of $1/\delta$.

The preceding analysis is more detailed than required for a *static* EPS-tree, but it is useful in that it guides the construction of a *dynamic* EPS-tree. In the dynamic case, assuming $y_{ij} = Y_{\max}$ for all i and j is not feasible, because B+-tree node splits may cause Y-sets to split unevenly. One approach would be to re-balance eagerly these Y-sets, but then the cost of updates would be too high. Instead, all we must guarantee is an update scheme where the *total sum* $\sum_{j=1}^{e_i} y_{ij}$ of Eq. 6.15 remains high for every visited node u_i.

6.6 Construction of EPS-trees

We now describe a simple and efficient construction technique for a EPS-tree over a given dataset. This operation, also known as *bulk loading*, is of great

practical importance, thus it deserves a short discussion.

We assume that the dataset has size n, and resides in a disk file, sorted on the $\prec_x$ order. To construct an EPS-tree, we first construct the underlying B+-tree. This can be done by well-known techniques (see [Com79]), for any chosen *fill factor* f_B, i.e., the nodes of the resulting B+-tree will have $f_B B$ elements each (where $1/2 < f_B \leq 1$). A fill factor less than 1 may be desired, when insertions to the tree are expected. For a fill factor of 1, each insertion would split the (originally) packed block, which results both in expensive insertions, and in reduced efficiency for later range searches. Practical fill factor are typically $f_B \approx 0.8$. The total number of I/Os required, is (approximately) $(1 + 1/f_B)\frac{n}{B}$.

Having constructed the underlying B+-tree, it is now possible to construct a CC for each internal node. Unfortunately, we do not know how to perform this operation in a linear number of I/Os. In fact, we conjecture that this is not possible. We show how to construct the child caches in $O\big((n/B)\log_M(n/B)\big)$ I/Os, where M is the amount of main memory used, measured in disk blocks. We conjecture that this cost is asymptotically tight.

Each Child Cache will hold a number of Y-sets. Parameter $Y_{\max}$ of the tree determines the maximum allowable Y-set size. Like for B+-tree blocks however, a *fill factor* $0 < f_Y \leq 1$ for the constructed Y-sets may be chosen. Note that in the case of Y-sets, smaller values of f_Y reduce space requirement, whereas smaller f_B will increase the space requirement for the B+-tree. On the other hand, smaller values of f_Y will increase the worst-case query cost (cf. Eq. 6.16). For the chosen value of f_Y, the construction of CCs will create Y-sets of size at most $f_Y Y_{\max}$. Note that it is possible for some Y-sets to have smaller size. This can happen for some node u, if $\lfloor u \rfloor$, i.e., the node is bottom.

For the chosen values of f_B and f_Y, the total space for all CCs will be approximately

$$\frac{f_Y Y_{\max}}{f_B B}(n/B)$$

blocks, and thus the total space of the constructed EPS-tree will be approximately

$$\big(f_Y \frac{Y_{\max}}{B} + \frac{1}{f_B}\big)\frac{n}{B}$$

blocks.

To construct the Child Caches, we work with the *levels* of the B+-tree. Level l of the B+-tree consists of the list of B+-tree nodes at distance l from the leaves, i.e., all leaf nodes are on level 0, and level $l > 0$ consists of all nodes whose children are at level $l-1$. The root is at level $L-1$, where L is number of levels of the tree. The height of the B+-tree is $H = L - 1$. In the B+-tree, all the nodes of each level form a linked list, sorted on the their x-ordering.

The construction of CCs begins at the root's level and proceeds downwards. When processing level l, the process constructs the Y-sets of all nodes at levels $l+1, \ldots, l+k$, for some $k \geq 1$, and thus creates the CCs for levels $l, \ldots, l+k-1$. The parameter k is determined by the available memory M. The memory must be large enough to fit $(f_B B)^i$ Y-sets for level $l+i+1$, $0 \leq i < k$. Thus, the following inequality must hold:

$$f_Y Y_{\max} \sum_{i=0}^{k-1} (f_B B)^i \approx f_Y Y_{\max}(f_B B)^k - 1 \leq MB$$

from which we obtain

$$k \leq 1 + \frac{\log M + \log B - \log(f_Y Y_{\max})}{\log(f_B B)}$$

The processing of each level proceeds by reading the input, and maintaining a tree of heaps (priority queues) in main memory, corresponding to the

Y-sets of a k-level subtree of some node in layer $l+1$. As soon as the Y-sets for all children of node u are computed, the CC of u is built on disk. Thus, the total I/O during CC construction is that of $H/k = O(\log_M(n/B))$ scans of the dataset. A final scan of the leaves of the B+-tree is needed, to update their **LNB**. Since these I/Os are roughly sequential, the overall process is efficient.

6.7 Making the EPS-tree dynamic

We now turn our attention to techniques for efficient updates to the EPS-tree. Central to these techniques, is the ability to perform efficient updates to CCs, i.e., an efficient implementation for the CC API of Table 6.1. Thus, we first concentrate on this issue. Then, we discuss updates to the EPS-tree.

As we will show in this section, it is possible to achieve $O(\log_B n)$ worst-case update cost, by using highly complicated update techniques for the CCs. Because implementation complexity may render these techniques impractical, we also show how to achieve $O(\log_B n)$ *amortized* update cost, with simpler implementation.

6.7.1 Updates to a Child Cache

The main update operations on CCs are insertion/deletion of keys to Y-sets, and split/merge of CCs.

Split of a CC can be done by building two new CCs out of the old one, in $O(Y_{\max})$ I/Os. Merging can be done also by building, with the same cost. An important observation is that the building operation need not be a continuous operation, but can be split up into $O(Y_{\max})$ steps of $O(1)$ I/Os each. Between two steps, all information of the rebuilding may be removed

from main memory. More precisely, the building has two stages: building the base blocks, and building the internal blocks, by Alg. 6.2. Observe that the state variables for both procedures can fit into $O(1)$ blocks. For example, the state for procedure **BuildInternal** (Alg. 6.2) consists of the variables S, *pos* and H, defined in lines 3–5 of Alg. 6.2. Thus, each step can cost $O(1)$ I/Os for loading the procedure's state, and $O(1)$ I/Os that produce $O(1)$ data blocks of the new cache.

Insertion and deletion for Y-sets must also be done by rebuilding. However, rebuilding for each insertion/deletion of a point would be prohibitively expensive. Instead, a special block, called the *update block*, can be added to the CC. Insertions and deletions of elements can be recorded to the update block. When the block becomes full, the CC can be rebuilt. Thus, the *amortized cost* of a Y-set update is $O(1)$ I/Os. With a slight modification, we can obtain $O(1)$ *worst-case* cost. We either use two update blocks, or we split a single update block into two parts. Thus, we get two update areas. We then use one area—the *active* area—to record Y-set updates, while the contents of the other area—the *frozen* area—are incorporated into a new CC by a rebuilding process. Thus, each Y-set update involves the following: record the update into the active update area (1 I/O), and perform $O(1)$ work in building a new CC, on the contents of the existing base blocks, modified by the updates recorded in the frozen area. The amount of work of each rebuilding step must be large enough to guarantee that the rebuilding will be finished before the active update area becomes full.

The operations of Y-set updates and split/merge of CCs can be combined, albeit with significant implementation complexity. This is outlined in the following statement:

Proposition 6.7.1. *Using multiple Child Caches with $Y_{\max} = O(B)$, we can amortize the work of a sequence of Y-set updates and split/merge operations, in $O(1)$ I/Os per step, provided that for some fixed constant k, there are at most k split/merge operations in every subsequence of operations of size $Y_{\max}$. Procedure* **Find** *(Alg. 6.3) on these CCs will return $O(t/B)$ blocks for a query retrieving t elements from the CC. The total number of records stored in the CCs must be $O(BY_{\max})$. (The hidden constants in these complexities depend on k).*

Proof. We only sketch the argument. The basic idea is to maintain an array of pointers to at most k *subsumed* CCs. A call to *Find* can be computed by posing the query to each subsumed CC, and concatenating the returned lists. The condition that split/merge operations do not happen too often, guarantees that there exists a suitable rebuilding schedule of $O(1)$ I/Os per step, so that the number of subsumed CCs does not grow above k. □

We now briefly discuss the implementation of the rest of the CC API, namely operations **SplitSet**, **LeastKey** and **GreatestKey** (see Table 6.2). The **LeastKey** operation can be implemented by examining the root block of the CC, incorporating any changes from the update block. This will require $O(1)$ I/Os. Also, **GreatestKey** can be implemented in $O(1)$ I/Os, by retrieving all $O(Y_{\max})$ keys of the desired Y-set, and selecting the $\prec_y$-highest (again, the update block must be scanned). It is easy to retrieve all points in a Y-set, from the base blocks of the CC. Finally, **SplitSet** and **Merge** are simply operations on array **YSize**, which resides in the catalog block, and cost 1 I/O.

Our CC implementation involved a number of design choices, that re-

quire some justification. In designing the CC, we consistently chose to favor efficient search over simplified updates. A number of alternative implementations are possible. For example, each Y-set may be maintained in its own block, replacing our implementation's base blocks. This choice would simplify somewhat the updates of CCs, but would require more complicated search, and would likely incur a small constant access overhead for queries. Choosing to favor search over update is a well-established engineering rule in database indexing.

The implementation complexity of some of the operations outlined in this section can be somewhat daunting in a real setting. This could qualify the EPS-tree as impractical. However, our experimental results, presented in Ch. 7, show that the requirement for worst-case EPS-tree updates is not necessary for an efficient solution. An optimal *amortized* update cost should be sufficient in practice. For such an EPS-tree implementation, it suffices to assure $O(1)$ *amortized* Y-set updates for the CCs of the EPS-tree. In particular, it suffices for the CC to have a single update block. When that block is full, the CC is rebuilt in one step of $O(Y_{\max})$ I/Os. CC splits and merges also happen immediately, with a cost of $O(Y_{\max})$ I/Os. We shall refer to the CC implementation outlined here as the ACC (Amortized Child Cache) implementation, and to the full implementation described by Prop. 6.7.1, as the WCC (Worst-case CC) implementation.

We summarize the update complexities of CC operations, in Table 6.3.

	ACC (amortized)	ACC (worst-case)	WCC
YSize	1	1	$O(1)$
Find	1	1	$O(1)$
bottom	1	1	$O(1)$
Insert	2	$O(Y_{\max})$	$O(1)$
Delete	2	$O(Y_{\max})$	$O(1)$
SplitSet	1	1	$O(1)$
Merge	1	1	$O(1)$
LeastKey	2	2	$O(1)$
GreatestKey	3	3	O(1)
SplitCache	$2Y_{\max}+1$	$2Y_{\max}+1$	$O(1)$†
MergeCache	$2Y_{\max}+1$	$2Y_{\max}+1$	$O(1)$†

Table 6.3: The I/O cost of Child Cache API operations.
†with the caveats of Prop. 6.7.1

6.7.2 Two operations on Y-sets

The Y-sets of an EPS-tree are re-organized using two EPS-tree operations: bubble-up and trickle-down. These operations are dual. Each operation originates at a particular node, and is propagated downward in the tree. Alg. 6.4 shows the pseudo-code for these two operations. Note that a bubble-up or trickle-down to node u will affect $Y(u)$, and thus must update the CC of u's parent.

The bubble-up operation on node u increases the size of $Y(u)$ by one. This is done by moving a key "upwards", i.e., retrieving the **LNB** of u, if u is leaf, or retrieving a key from the Y-set $Y(v)$ of some child v of u, if u is not leaf. In the latter case, the bubble-up recursively extends to v, so that $Y(v)$ does not decrease in size.

The trickle-down operation on node u decreases the size of $Y(u)$, by

Algorithm 6.4 Trickle-down and bubble-up.

```
1.  TrickleDown(Node u)
2.  {    3Key p := CC(u↑).GreatestKey(u);
4.     CC(u↑).Delete(p);
5.     if( u is leaf )
6.        adjust LNB(u);
7.     else {
8.        if(CC(u).YSetSize(u^p)=Y_max)
9.           TrickleDown(u^p);        // make space
10.       CC(u).Insert(p, u^p);
11.    }
12. }
13.
14. BubbleUp(Node u)
15. {
16.    if( u is leaf ) {
17.       CC(u↑).Insert(LNB(u), u);
18.       adjust LNB(u);
19.    } else {
20.       Key p = CC(u).LeastKey();
21.       if(CC(u).YSetSize(u^p) ≤ Y_max/2)
22.          BubbleUp(w, u);
23.       CC(u).Delete(p);
24.       CC(u↑).Insert(p, u);
25.    }
26. }
```

moving a key "downwards", in a dual manner to bubble-up.

6.7.3 Insertion in EPS-trees

We will now describe the insertion operation. The description is stated in terms of well-known B+-tree operations, and CC API operations already discussed. To make the description cleaner, we introduce a concept that relates to B+-tree updates.

Consider an insertion into a B+-tree (a similar description applies to deletions). We will define a B+-tree node to be the *designated node* for the insertion. Let u be the leaf where the inserted key will reside. If u does not split as a result of the insertion, the u is the designated node. If u splits, the split is propagated towards the root, splitting ancestors until some ancestor v that does not split. In this case, v is the designated node. Fig. 6.6 depicts this

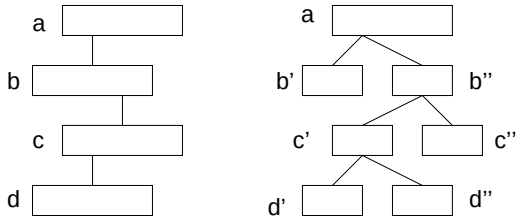

Figure 6.6: The root-to-leaf path to the left is from a tree, before an insertion into leaf d. To the right, some ancestors of d have split. The designated node is a.

case. Finally, it may happen that the splits propagate to the root, which splits too, increasing the height of the B+-tree. In this case, there is no designated node.

We now describe the insertion procedure into an EPS-tree. Let p be

the inserted key. Insertion will follow these steps:

1. Insert p into the B+-tree. Let u be the leaf containing p (after the insertion, i.e., after all splits), and let d_0 be the designated node of the insertion (or be null if the root split).

2. Perform any **SplitCache** operations required (one for each internal node that split).

3. If $p \prec_y$ **LNB(u)**, examine the path from root to leaf u, to locate the node v for which p must be inserted into $Y(v)$, in order to maintain the Y-set invariant. This can be done by calling **LeastKey** for the CCs of the nodes on the path. If $|Y(v)| = Y_{\max}$, call **TrickleDown(v)**. Then, insert p into $Y(v)$.

4. If d_0 is undefined, or d_0 is the root, we are done.

5. Else, let d_{-1} and d_1 be the left and right siblings of d (if they exist). For each $Y(d_i)$, such that $|Y(d_i)| < Y_{\max}/2$, call **BubbleUp(d_i)** $\lceil \frac{Y_{\max}}{B} \rceil$ times. Note: in practice, $\lceil \frac{Y_{\max}}{B} \rceil = 1$.

All of the above steps, except the last, are easy to justify. The purpose of the last step, which is called the *rebalance step*, is not immediately obvious. It will be justified by the forthcoming discussion.

6.7.4 Analysis of insertion cost

The cost of an insertion operation is determined by the choice of implementation for Child Caches. From Table 6.3, it can be seen that the worst-case cost for the WCC implementation is $O(\log_B n)$. This can be seen easily, by

inspecting the steps of the insertion process. The only point of interest is to assure that the caveat of Prop. 6.7.1 is obeyed. Indeed, consider the interval between two successive calls to **SplitCache** on the CC of some internal node u. Right after the first call, u had exactly $B/2$ children (because it had just split). Right before the second call, it had B children (because then it split). Thus, during this time, it had become the designated node $B/2$ times. Therefore, at least $B/2$ bubble-ups were called on its children, which translate to at least $B/2$ insertions into u's CC. Thus, there are at least $B/2$ Y-set updates between two calls to **SplitCache**.

For the ACC implementation, the *amortized* cost of insertion is $O(\log_B n)$. In the worst case, an insertion can cause $\log_B n$ splits, and thus a cost of $O(Y_{\max} \log_B n)$ I/Os. However, the frequency of such splits is very low. Over a large number of insertions, only one in every (approximately) $B/2$ insertions will cause a split of some leaf. Thus, the amortized cost is $O((2Y_{\max}/B) \log_B n) = O(\log_B n)$.

6.7.5 Search in Dynamic EPS-trees

The **Search** procedure of Alg. 6.1 will return a correct result for dynamic EPS-trees, but in some rare circumstances, its cost may not be optimal. We explain the problem, and describe a slight modification that assures optimal cost.

The source of the problem is that, when a node u of the B+-tree splits into nodes u_1 and u_2, as a result of some insertion, $Y(u)$ does not, in general, split into equal parts. Without loss of generality, let $|Y(u_1)| \leq |Y(u_2)|$. It is possible that $Y(u_1)$ can be empty, or contain very few elements. Such an

extreme case is shown in Fig. 6.7. In the following, a node whose Y-set contains fewer than $Y_{\max}/4$ elements is called a *light* node, otherwise it is a *heavy* node.

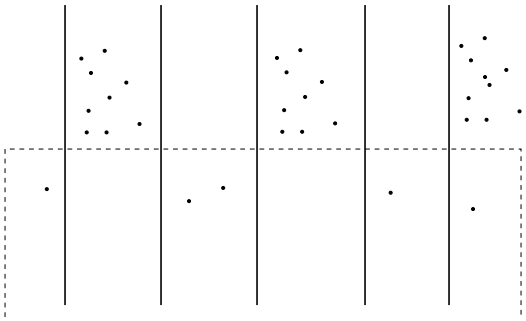

Figure 6.7: The Y-sets of sibling nodes, after splitting. The shown query (dashed line) will report all keys from the light nodes' Y-sets.

If there are no light nodes in the EPS-tree, the optimality condition of Eq. 6.15,

$$\delta B e_i \leq \sum_{j=1}^{e_i} y_{ij},$$

is satisfied for $\delta = \frac{Y_{\max}}{4B}$. In the presence of light nodes however, $1/\delta = O(1)$ cannot be achieved, without slight modifications of procedures **Search** and **BubbleUp**. We now explain this issue.

The rebalance step of the insertion procedure (step 5), assures that every node that splits has at least $Y_{\max}/2$ elements in its Y-set. This is because, between the time a node was created and the time the node splits, it has become the designated node $B/2$ times, and thus receives at least $(B/2)\lceil Y_{\max}/B \rceil \geq Y_{\max}/2$ bubble-ups. In what follows, let u be a node that split into a light node u_1 and a heavy node u_2 (if $|Y(u)| \geq Y_{\max}/2$, then at least one of u_1 and u_2 is heavy).

The pitfall of procedure **Search** is that it will extend the search to the light node u_1, if $Y(u_1)$ is fully reported. The cost of extending the search cannot, in this case, be amortized on the number of reported points with optimal efficiency.

The remedy is based on the following important observation: immediately after the split, the search need not descend to u_1, *unless the whole contents of* $Y(u_2)$ *are also fully reported*. If search is modified in this way, amortization of its cost is again efficient. To make our discussion precise, we introduce two definitions.

Definition 6.1. *A key* $p \in \mathcal{I}$ *is below node* u *if, and only if, for some ancestor* v *of* u *(including* u *itself),* $p \in Y(v)$*. Otherwise,* p *is above* u*.*

Definition 6.2. *Let* u_1 *and* u_2 *be adjacent sibling (children of the same node) EPS-tree nodes, and assume* u_1 *is light and* u_2 *is heavy. We say that* u_1 *and* u_2 *are joined if, and only if, for all* $p \in [u_1]$*,* p *is above* u_1 *implies that* p *is above* u_2*.*

We can now restate our important observation as follows: immediately after the split, nodes u_1 and u_2 are joined. When two nodes are joined, search need not descend to the light node, unless both their Y-sets are fully reported.

Thus, our task is reduced to maintaining joinedness over a number of subsequent operations, so that u_1 can receive enough bubble-ups by the rebalance step (step 5) of the EPS-tree insertion procedure. Once it holds, joinedness can be broken if u_2 (the heavy node) receives bubble-ups. Trickle-downs to u_2, or bubble-ups to u_1, do not break joinedness. Also, trickle-downs are never applied to u_1, while u_1 is light (a light node's Y-set does not reduce in size).

Thus, to maintain joinedness, we can simply modify procedure **BubbleUp**, and in particular lines 21–22 of Alg. 6.4. These are the lines that recursively extend the bubble-up to a child node that had an element of its Y-set bubbled-up. The modified version will confirm that u_2 is the heavy side of a joined pair of nodes, and will assign the bubble-up to the light node of the joined pair. We omit pseudo-code for this process, as we omit pseudo-code for the modifications of Alg. 6.1.

6.7.6 Deletion in EPS-trees

Removing a key from the EPS-tree is straightforward, unless the deletion causes some nodes to merge. In this case, EPS-tree deletion suffers from the problems of deletion in B+-trees, such as thrashing [Com79]. As discussed in [ASV99], a *global rebuilding* technique (e.g. see Overmars [Ove83]) can be used, where keys are not immediately removed from leaves, but instead the structure is rebuilt after $\Theta(n)$ deletions. This technique can achieve $O(\log_B n)$ amortized deletion cost, and is in fact quite popular in database practice (it is often called *reorganization*).

If it is acceptable for deletion costs to be high (in the case of B+-tree node merges), deletion can be performed relatively simply, along the same lines as insertion: perform B+-tree deletion, and maintain the Y-set invariants. Maintaining Y-set invariants can be expensive when two nodes merge, because the union of their Y-sets is not, in general, a legal Y-set for the new merged node. Thus, up to $\Omega(Y_{\max})$ trickle-downs and bubble-ups may need to be performed, to maintain correctness. Yet, assuming that merges only happen rarely, the expected cost of deletion is $O(\log_B n)$.

The above discussion seems to be of academic interest with the given state of affairs in the database industry. In many commercial implementations of B+-trees, merging of B+-tree nodes never happens [GG98]; instead, nodes are allowed to underflow, and are only removed when they become empty. Under this state of operations, deletion in EPS-trees consists of simply removing the key from its leaf node, and possibly from the Y-set it resides in (if the key is bubbled). Naturally, worst-case search cost can be adversely affected if many nodes underflow (as is the case for B+-tree search costs). However, deletions seem to be less important in practical settings.

6.8 Conclusions

This chapter examined techniques for implementing index structures for two-dimensional range queries. The focus was on the EPS-tree, a new access method for three-sided queries, with asymptotically optimal worst-case performance. The detail of the presentation was high, in order to demonstrate the practical decisions involved in designing such data structures. We now review some of the main conclusions from the material presented.

6.8.1 Practical aspects of indexability

Although the indexability model does not include a search component, we have shown that indexing schemes can be useful in index construction. The Child Cache subindex is derived directly from the indexing scheme of 4.3.1. The approach can be generalized to other problems, beyond 3-sided queries. In the survey of Vitter [Vit99] this technique is called *bootstrapping*.

The general idea behind bootstrapping is to externalize a main-memory

data structure, by using small subindices (designed as indexing schemes) to perform *filtering search* [Cha86]. In various forms, this technique was employed by Ramaswamy and Subramanian in the P-range tree [SR95], by Arge and Vitter in the External Interval Tree [AV96], and by Vengroff and Vitter in their 3-dimensional index structures [VV96a]. Since our work, it has been employed by Agarwal *et al.* [AAE+98] to indexing for half-space queries, and moving points on the plane [AAE00].

The main merit of bootstrapping as a general approach, is that it does not (potentially) suffer from the weaknesses of the Path Caching technique of Ramaswamy and Subramanian, discussed in Sec. 6.2.2. The main drawback is that it assumes a solution to the underlying indexability problem. Such solutions seem to be hard for many problems of interest.

6.8.2 Dynamizing external data structures

The update procedures of our dynamic EPS-tree are undoubtedly more complicated than typical in indexing techniques. It is desirable to develop simpler update techniques, that do not sacrifice performance. Such techniques would not only benefit the EPS-tree but other dynamic index structures as well, like the External Interval Tree, or the p-range tree, whose update operations are (at least) equally complicated.

6.8.3 EPS*-trees

An alternative to our EPS-tree design could be to merge B+-tree internal nodes with the catalog blocks of their corresponding CCs. However, our choice to maintain these as separate blocks has a desirable consequence; it allows for

multiple choices of the y-dimension to be served by the same underlying B+-tree. For example, it may be desired that both open-above (i.e., $\sqcup$-like) and open-below (i.e., $\sqcap$-like) 3-sided queries be supported over the same dataset, as in the case of 4-sided indexing scheme of §4.4. With our design, it suffices to augment each B+-tree internal node with multiple CCs, and each B+-tree leaf node with multiple **LNB**s. Updates will work well, because the B+-tree update procedure is independent of any CC state. In fact, with our design, 3-sided query capability can be added or removed from the underlying B+-tree at will, simply by creating and destroying the CCs of internal nodes. We designate such multi-CC trees as EPS*-trees.

In addition, our decision not to compromise the underlying B+-tree, assures that 1-dimensional range queries over the B+-tree attribute will have excellent performance. One application where this may prove important is interval management. An interval intersection query can be seen as a 3-sided query. However, as shown in Sec. 4.2, an interval intersection query can also be split into two disjoint, smaller queries, one of which is an interval stabbing query, and the other a one-dimensional range query. In our design, we can use the underlying B+-tree to answer the one-dimensional query, and three-sided search to answer the stabbing query. For large queries, where the cost of the one-dimensional query dominates by far the cost of the stabbing query, our design will probably achieve superior performance, since the one-dimensional query is answered with the efficiency of a B+-tree.

Chapter 7

Empirical Evaluation of EPS-trees

The performance analysis of the EPS-tree has, so far, concentrated on worst-case search and update cost. From a practical perspective, an analysis of expected cost is also important. In this chapter, we carry out a thorough experimental study of EPS-trees. Our goals are two-fold; first, to measure the average performance of EPS-trees, and second, to estimate the implementation complexity of the various EPS-tree operations, and contrast it against the performance gains.

The worst-case I/O of Eq. 6.16 is already within a constant factor of optimal. Thus, the only question about the average-case I/O is, what is the average-case constant factor. The question becomes more interesting when one considers the role of parameter $Y_{\max}$ in the worst-case cost. In general, larger values of $Y_{\max}$ decrease the I/O cost of search, and increase disk space consumption and update cost. One goal is to study the effect of the choice of

$Y_{\max}$ on average access cost.

There are additional experimental goals, relating to search performance. Some derive from the approximate nature of I/O modeling, in the analysis of the previous chapter. The access cost for real disks varies significantly when the I/Os are sequential, vs. random. Another related issue is the effect of caching some disk blocks in a main memory buffer. This is a standard feature of modern systems, and is well-known to reduce significantly the actual access cost of queries.

A final goal concerns the implementation complexity of EPS-trees, compared to practical index structures, and in particular, the trade-off between implementation complexity and performance.

7.1 Experiment design

Our first task is to describe and justify the adopted experimental procedures. An important choice is the evaluation workload, i.e., the dataset stored in the EPS-tree, and set of queries whose performance is to be measured. The keys returned by an EPS-tree query are retrieved from two sources: data blocks of Child Caches, and leaves of the B+-tree. The contribution of these two sources of data can vary, depending on the shape of a query, and the data distribution of the underlying dataset. We chose a synthetic, uniformly distributed dataset. For this dataset, we constructed a set of queries, by varying the query size, as well as the aspect ratio. By this choice, we were able to exercise both types of sources where data resided, with varying contributions from each type of source.

In order to compare our results to other, well-known index structures,

we faced a dilemma, because there are no well-known data structures for three-sided queries. Comparing against some more general index structure, such as the R-tree, would not provide a meaningful comparison,since the EPS-tree can out-perform R-trees—and other similar structures of more general scope—by orders of magnitude in some cases. Thus, we chose to compare EPS-trees against B+-trees. Of course, B+-trees are not two-dimensional structures. In order to have a meaningful comparison, we compare the access cost of the EPS-tree answering a query of size t, to the cost of a B+-tree answering a one-dimensional query of the same size t. Clearly, the result of such a comparison is not to choose the "best" between these two structures, since we compared them on different tasks. The rationale is the following: the access cost of a B+-tree query of size t, is close to the absolute optimum cost that any type of access method, retrieving t keys, would have to incur. Thus, by contrasting the EPS-tree cost to such an optimum, we relate the performance of EPS-trees to the physical parameters imposed by the hardware and operating system.

Our experiments are guided by a number of hypotheses that we seek to validate. All of these hypotheses are founded in the theoretical analysis of the previous chapters, where we concentrated on the number of I/Os as the only performance metric.

- We expect the constant factor of average-case query cost to be smaller than that of the worst-case query cost. Intuitively, we expect it to be about half of the worst-case constant, for our uniformly distributed dataset. By query cost, we understand not only the number of blocks accessed, but also the time per query.

- We expect that for some "hard" queries, the actual cost will be very

close to the worst-case cost. Those will be the queries where search cost is just barely amortized better than what worst-case analysis predicts.

- We expect that smaller values of $Y_{\max}$ will yield inferior performance over larger values, with the effect being more pronounced for certain "hard" queries. Thus, we validate the notion that I/O cost can be improved by disciplined use of redundancy.

- We expect EPS-tree performance to be within a small constant factor of B+-tree performance on average, for same-size query searches, even when the B+-tree is laid sequentially on disk. Our expectation is based on the fact that CC data blocks were also laid sequentially (this is possible because CCs are static indexes). Thus, we did not expect the EPS-tree to be penalized too much from random disk seeks.

- We expect that the effect of the main-memory buffer will be very beneficial to access cost, as it is for most other tree-like index structures. In particular, we expect that the cost of search will be, to a large extent, hidden by the effects of the buffer. Thus, the average number of real I/Os should approximate the *indexability* access overhead. From §4.3.1, the worst-case indexability overhead is $A = 4$. On average however, we expect the I/Os per query to be (roughly) only $2 \log_B n + 2\lceil t/B \rceil$.

- We expected the real I/O cost per update to be comparable to that of the B+-tree. Our reason is that the additional work of EPS-tree updates involves mainly operations on the update blocks of CCs. The number of these update blocks is equal to the number of internal tree nodes, and most of them will fit in the main-memory buffer. Thus, updates to the

CCs will not cost as much in real I/O operations.

Our experimental results confirmed these hypotheses.

7.2 Experiment setup

Our workload simulated a secondary index over a database table. We constructed a dataset of 100,000,000 keys, where each key consisted of a tuple of 4-byte integers. Each key had an associated 4-byte datum, which was unused—presumed to represent a physical pointer in our hypothetical database table. Thus, each record was 12 bytes long, and the ideal space requirement for our dataset was approximately 1.2 Gbytes.

Using this dataset, we constructed a number of EPS-trees, for different values of $Y_{\max}$. The block size was set to 4 kbytes. The internal and leaf nodes of the EPS-tree had a fill factor of 0.8. Thus, each leaf node of the EPS-tree would hold roughly $4096 \times 0.8/12 \approx 270$ keys, and also that was the maximum branching factor of the internal nodes of the EPS-tree. Our main-memory buffer was set to 64 Mbytes, or almost 5% of the raw data size.

Our hardware was an Intel Pentium III 650MHz machine, with 128 Mbytes of RAM. Our disk was a 35 Gbyte Maxtor Diamond, with a 2 Mbyte on-disk cache. Its nominal access time was 9ms. The disk was accessed through a UltraDMA/66 interface, and had a maximum transfer rate of 16 Mbytes/sec.

In order to establish a baseline of the performance of our system, we constructed a B+-tree over our dataset, and ran a large workload of one-dimensional queries over it. For our block size of 4 kbytes, and load factor 0.8, the B+-tree had approximately 270 keys per leaf block. Although our B+-tree (and all our indexes) were stored as operating system files, our bulk

loading process and the file system's algorithms resulted in a mostly sequential allocation of the B+-tree leaf blocks. This had the effect that, for large B+-tree queries, the access cost was almost equal to the disk bandwidth (adjusted for the B+-tree's fill factor).

We ran a workload of uniformly distributed queries. The query size t ranged from 10 to 100,000 keys. For each query, we measured four quantities: the CPU time (**CPU**), the total wall-clock time (**TOT**), the number of block accesses (**ACC**), either from disk, or from the main-memory buffer, and finally the actual number of disk I/Os (**IOS**). Fig. 7.1 shows a scatter plot of ACC and IOS, vs. the normalized query size $\lceil t/B \rceil$.

As can be seen in Fig. 7.1, the ACC results are very tightly correlated to the query size. However, the IOS results have many points well below the number of block accesses. This is the effect of the main-memory buffer.

In order to approximate analytically the displayed data, we computed least-square approximations. From these we got

$$\text{ACC} = 3.4 + \lceil \frac{t}{B} \rceil \tag{7.1}$$

$$\text{IOS} = 1.1 + 0.96 \lceil \frac{t}{B} \rceil \tag{7.2}$$

The formula for ACC is interpreted as follows: our B+-tree had height 3, thus, each query would access 3 internal nodes, and between $\lceil t/B \rceil$ and $\lceil (t-1)/B \rceil + 1$ leaves. Thus, about 40% of the queries needed $\lceil (t-1)/B \rceil + 1$ blocks.

The formula for IOS has a coefficient of 0.96 with respect to the query size. This is explained as the hit ratio of the buffer. The buffer had size 64 Mbytes, and the total B+-tree size was approximately 1.5 Gbytes. Thus, the hit ratio was (roughly) 4% for leaf blocks. Note also that the expected number of I/Os for accessing internal B+-tree nodes was 0.7 accesses per query

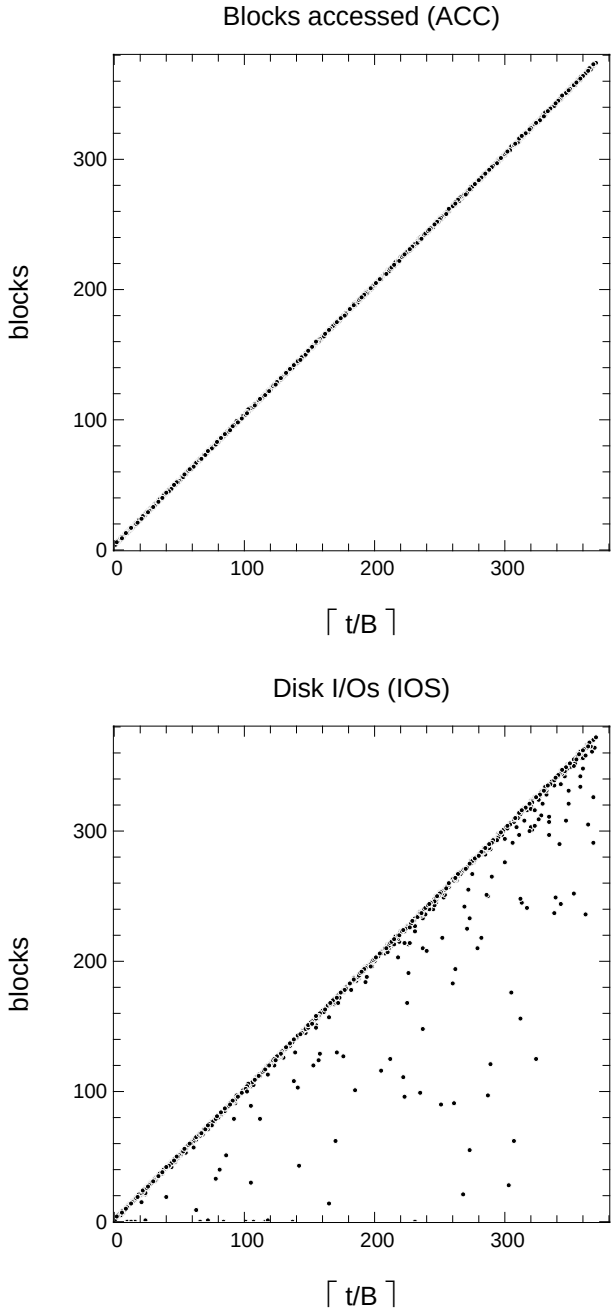

Figure 7.1: Scatter plot showing block accesses and disk I/Os, vs. the normalized query size $\lceil t/B \rceil$.

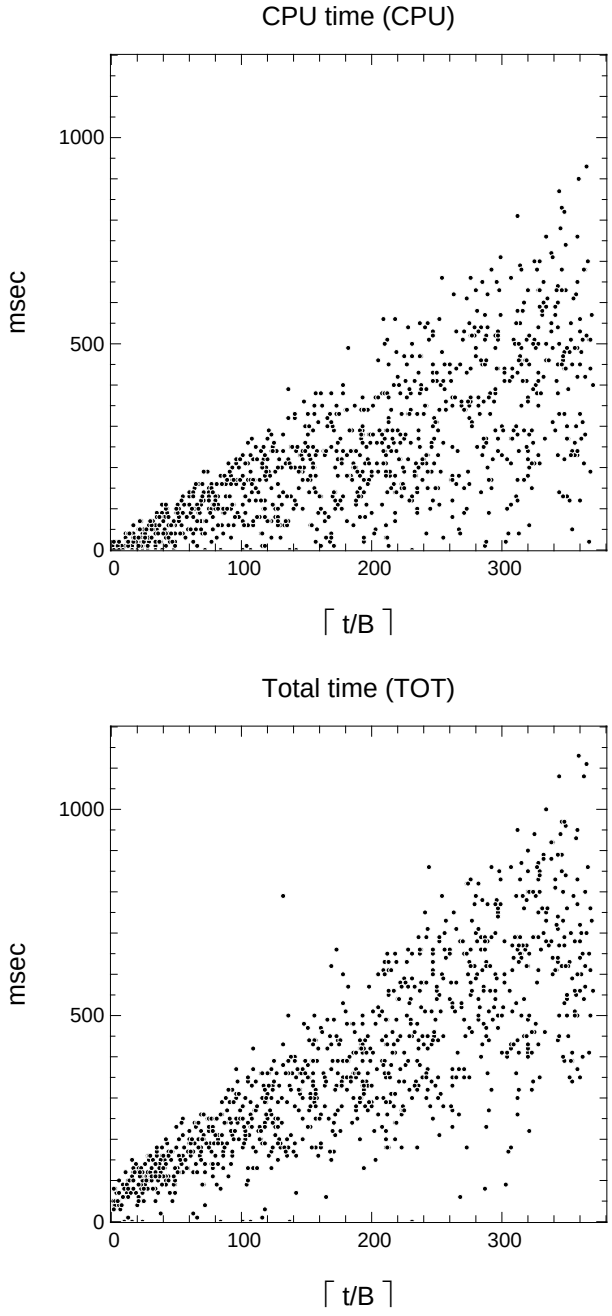

Figure 7.2: Scatter plot showing CPU and total time, vs. the normalized query size $\lceil t/B \rceil$.

(this is equal to 1.1-0.4, where 0.4 is subtracted because 40% of the queries would access $\lceil (t-1)/B \rceil + 1$ blocks).

Fig. 7.2 shows a scatter plot of the CPU and total time, against the normalized query size $\lceil t/B \rceil$. Although the variance of the data is higher, it can be seen that the times are linearly correlated to the query size, and almost all of the queries completed in under 1 second. A least-squares linear approximation gave the following formulas:

$$\text{CPU} = 1.95 + 1.26\lceil \frac{t}{B} \rceil \text{ (in msec)} \quad (7.3)$$

$$\text{TOT} = 54 + 1.74\lceil \frac{t}{B} \rceil \text{ (in msec)} \quad (7.4)$$

We should especially comment on the TOT formula, that the 54 msec overhead is caused by the 1 or 2 disk seeks required by the query, but that then, most of the leaf block accesses are very fast (1.74 msec on average), which implies that the I/O was mostly sequential.

Having presented the performance of B+-trees on our experiment setup, we are now ready to present the results for EPS-tree performance. The experiment presented in this section should be considered a control experiment, to help interpret the numbers to be presented subsequently.

7.3 EPS-tree query performance

In order to explore the effect of the $Y_{\max}$ parameter on the query performance of EPS-trees, we constructed a number of different trees, where $Y_{\max}$ varied from 15 to 170. We used a block size of 4 kbytes, and record size of 12 bytes. Thus, the Y-sets of the EPS-trees ranged in size from 180 bytes to 2040 bytes. For this range, the space overhead above the 1.5 Gbytes of the underlying B+-

tree, ranged from 152 Mbytes to 1.5 Gbyte. Totally, the space consumption was from 1.65 Gbytes to 3 Gbytes.

Our workload consisted of 10,000 three-sided queries, whose sizes ranged from 10 to 100,000 records. These queries had varying shapes, from "broad and short" to "thin and tall". In the following, we will designate the shape of a query by a parameter ρ. For a three-sided query $Q(a, b, c)$, $\rho(Q)$ is the ratio $c/y_{\max}$, where c is the y-restriction of the query, and $y_{\max}$ is the maximum y-coordinate over all points in the dataset. Since our workload consisted of queries with $c \leq y_{\max}$, ρ takes values between 0 and 1.

From the foregoing discussion, we should expect that the cost of a query will depend strongly on parameters t and ρ of the query, and parameter $Y_{\max}$ of the EPS-tree it was posed against. Our measurements fully support this conclusion.

We begin with query performance on an EPS-tree with $Y_{\max} = 170$. Fig. 7.3 shows the block accesses (ACC) and disk I/Os (IOS). Each measurement is shown as a colored point, depending on its ρ parameter. The color bar on the right of the graphs associates the colors with the corresponding values of ρ. As can be seen, the performance is linearly correlated to the query size. However, the correlation becomes much stronger when the value of ρ is taken into account. This agrees with our hypothesis, that the performance depends on which parts of the EPS-tree nodes contribute to the search. For queries of the same ρ, i.e., of the same height, the size of the query is a function of the breadth of the query, and the fractions of contribution of CC data blocks and B+-tree leaves remain unchanged. Thus, for fixed ρ, the performance is—as it should be—more strongly correlated to the query size t.

The results of Fig. 7.3 demonstrate that, with respect to the number

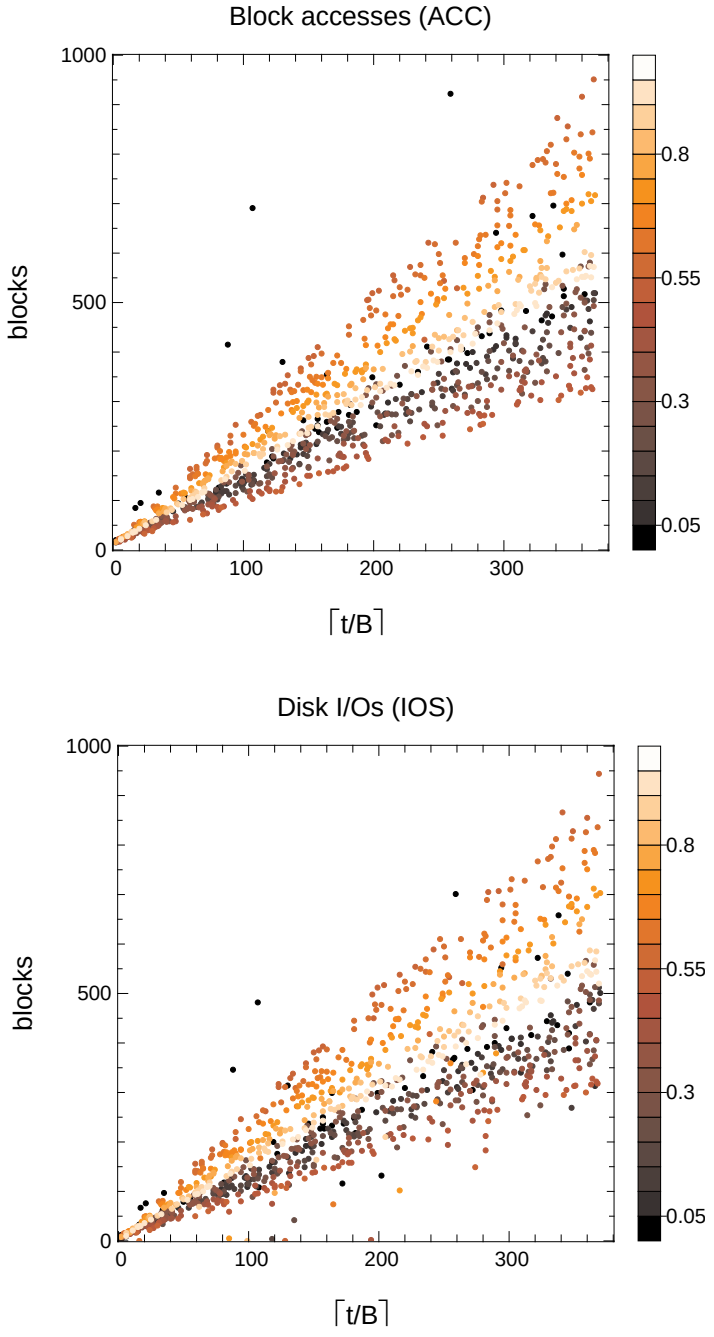

Figure 7.3: Scatter plot showing block accesses and disk I/Os, vs. $\lceil t/B \rceil$.

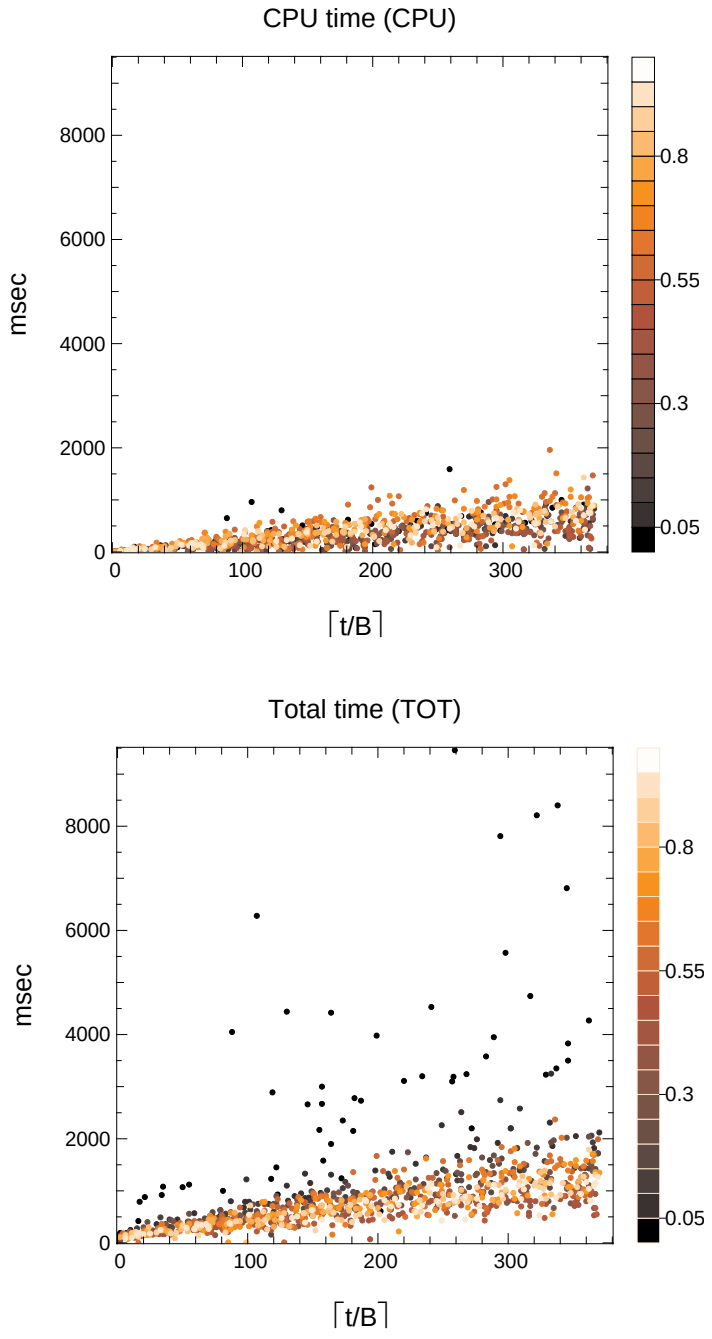

Figure 7.4: Scatter plot showing CPU and total times, vs. $\lceil t/B \rceil$.

of block accesses (either I/Os or buffer hits), the EPS-tree delivers excellent performance. Indeed, the number of block accesses is within a factor of 2.5 of the theoretical optimum $\lceil t/B \rceil$. The situation is also very good with respect to the CPU and total time. Fig. 7.4 shows the relevant data. Notice that the CPU time is relatively well-bounded, but the total time of some queries is as big as 8.5 sec. Yet, overall, the total times of even the larger queries are within 2 sec for the largest queries. Contrasting this number with the actual I/Os performed for these largest queries ($\approx$ 1000 I/Os from Fig. 7.3), we see that the average time per disk I/O is roughly 2 msec. Clearly, most of these I/Os are sequential. This confirms our hypothesis that the EPS-tree will not be penalized too heavily by random seeks.

7.3.1 The effect of $Y_{\max}$ on performance

Our analytical results imply that the choice of $Y_{\max}$ should have a significant effect on query performance. Indeed, our measurements indicate that this is correct. Unfortunately, it would be quite difficult to visualize directly the measurements over a broad range of $Y_{\max}$, ρ and t. We can approximate the performance measurements analytically, if we assume that query performance is linear in $\lceil t/B \rceil$. By the our discussion so far, it should be clear that this assumption is reasonable. Under this assumption, we can write the following

expressions for our performance measures.

$$C_{\mathrm{CPU}}(\rho, Y_{\max}, t) = K_{\mathrm{CPU}}(\rho, Y_{\max}) + L_{\mathrm{CPU}}(\rho, Y_{\max})\lceil \frac{t}{B} \rceil \quad (7.5)$$

$$C_{\mathrm{TOT}}(\rho, Y_{\max}, t) = K_{\mathrm{TOT}}(\rho, Y_{\max}) + L_{\mathrm{TOT}}(\rho, Y_{\max})\lceil \frac{t}{B} \rceil \quad (7.6)$$

$$C_{\mathrm{ACC}}(\rho, Y_{\max}, t) = K_{\mathrm{ACC}}(\rho, Y_{\max}) + L_{\mathrm{ACC}}(\rho, Y_{\max})\lceil \frac{t}{B} \rceil \quad (7.7)$$

$$C_{\mathrm{IOS}}(\rho, Y_{\max}, t) = K_{\mathrm{IOS}}(\rho, Y_{\max}) + L_{\mathrm{IOS}}(\rho, Y_{\max})\lceil \frac{t}{B} \rceil \quad (7.8)$$

We used piece-wise linear least-squares fitting to estimate the $K(\rho, Y_{\max})$ and $L(\rho, Y_{\max})$ coefficients from our measurements. From these, we then computed the averages over ρ. These are shown in Fig. 7.5. Two interesting observa-

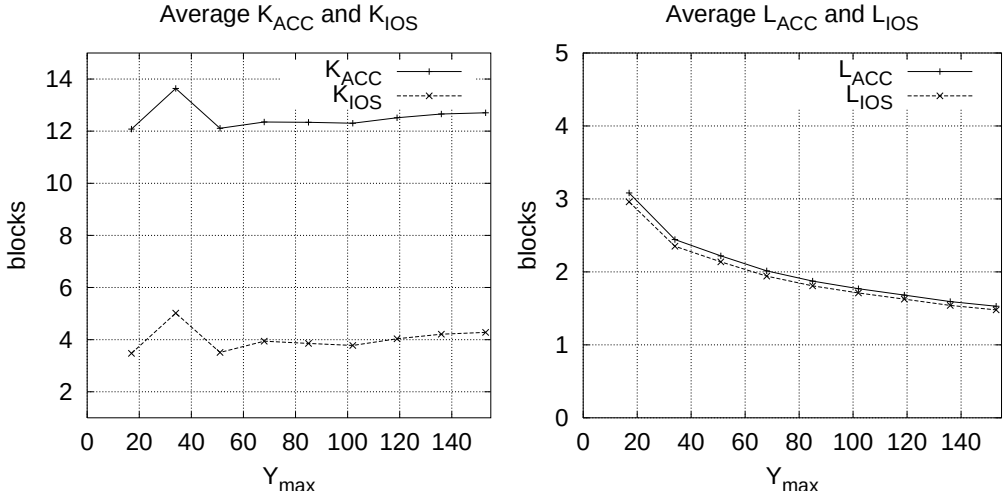

Figure 7.5: Average K and L coefficients for ACC and IOS.

tions follow. First, we expected that the L_{ACC} curve would be bounded by $O(B/Y_{\max})$. We observe that a similar relationship seems to exist for the average case. Note that the curve for L_{ACC} of Fig. 7.5 is essentially a space-I/O trade-off; larger values of $Y_{\max}$ improve performance. Also, notice the K_{ACC} is close to 12. This number is explained by our implementation. Each access

of an internal EPS-tree node incurs 3 block accesses; the node itself, and the catalog and update blocks of the corresponding Child Cache. Since the height of the EPS-tree was 3, each query would access between 3 and 5 internal nodes, thus would incur between 9 and 15 block accesses. However, as can be seen by the curve for K_{IOS}, most of these block accesses do not incur I/Os, but are fetched from the buffer.

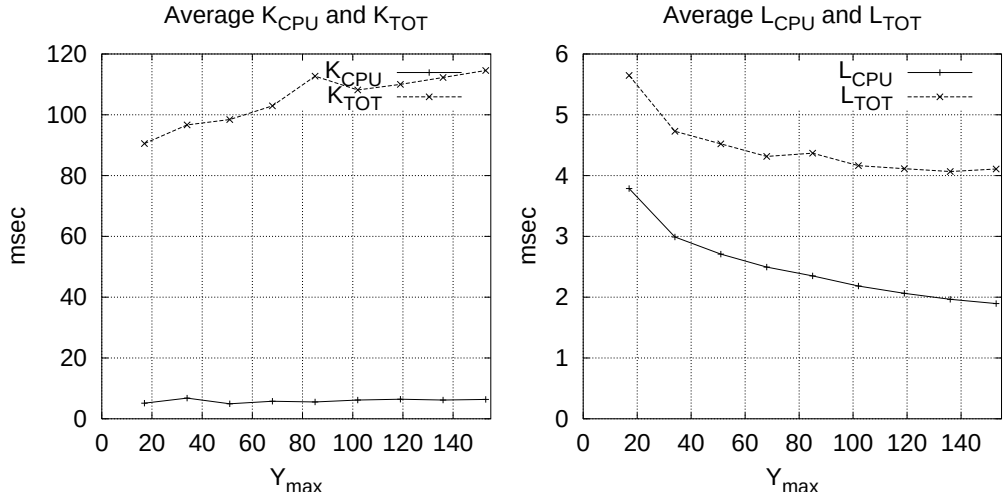

Figure 7.6: Average K and L coefficients for CPU and TOT.

The average coefficients for CPU and TOT times are shown in Fig. 7.6. It can be seen that the CPU time is significantly smaller than the wall-clock time.

The cost formulas derived for the EPS-tree with $Y_{\max} = 155$, averaged

over ρ, are:

$$C_{\mathrm{CPU}}(t) = 6.3 + 1.9\lceil\frac{t}{B}\rceil \text{ (msec)} \quad (7.9)$$

$$C_{\mathrm{TOT}}(t) = 114 + 4.1\lceil\frac{t}{B}\rceil \text{ (msec)} \quad (7.10)$$

$$C_{\mathrm{ACC}}(t) = 12.7 + 1.5\lceil\frac{t}{B}\rceil \text{ (blocks)} \quad (7.11)$$

$$C_{\mathrm{IOS}}(t) = 4.3 + 1.5\lceil\frac{t}{B}\rceil \text{ (blocks)} \quad (7.12)$$

By contrasting these equations with those of the B+-tree from §7.2, we see that EPS-tree performance is within a factor of 1.5 to 3 of the performance of B+-trees, compared with respect to query result size.

7.3.2 The effect of ρ on performance

The average-case results presented so far, are significantly better that the worst-case results derived analytically in the previous chapter. In particular, the fact that the coefficient L_{ACC} will only improve from 3 to 1.5, as $Y_{\max}$ ranges from a low value of 15 to a high value of 160, is intriguing. It could possibly be argued that low values of $Y_{\max}$ could be sufficient to guarantee acceptable query performance. A low $Y_{\max}$ would result not only in reduced space usage, but also in more efficient updates.

Unfortunately, this is not the case. Our results indicate that there exists a systematically derivable set of queries, whose performance can be significantly worse than the average, for small values of $Y_{\max}$. Fig. 7.7 depicts K_{ACC} and L_{ACC} for different values of ρ. Each curve corresponds to a different value of $Y_{\max}$. From this figure, it becomes clear that, although for some ρ the EPS-tree remains efficient even for small values of $Y_{\max}$, there are values of ρ where the performance will deteriorate significantly, almost by an order

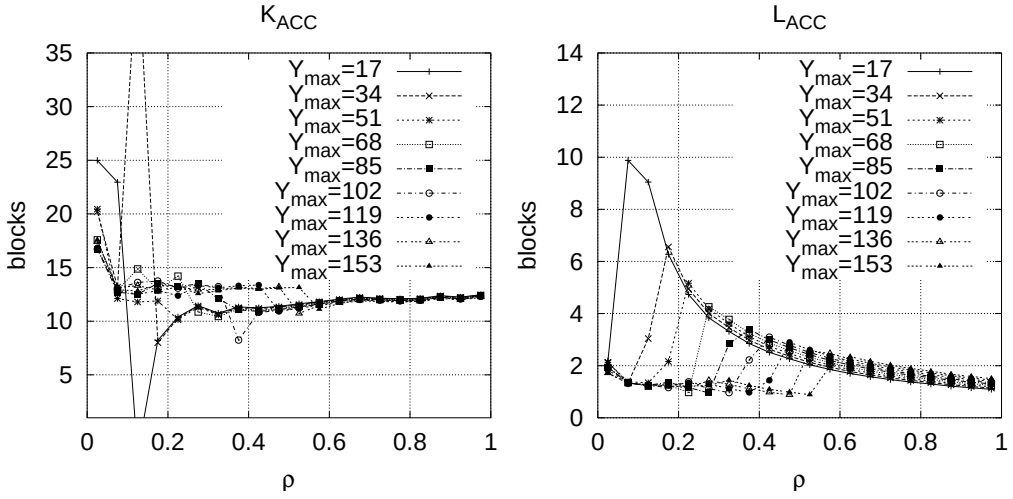

Figure 7.7: $K_{\mathrm{ACC}}(\rho, Y_{\max})$ and $L_{\mathrm{ACC}}(\rho, Y_{\max})$ vs. ρ.

of magnitude, unless $Y_{\max}$ is relatively large. The situation is similar with our other performance measures, as shown in Fig. 7.8.

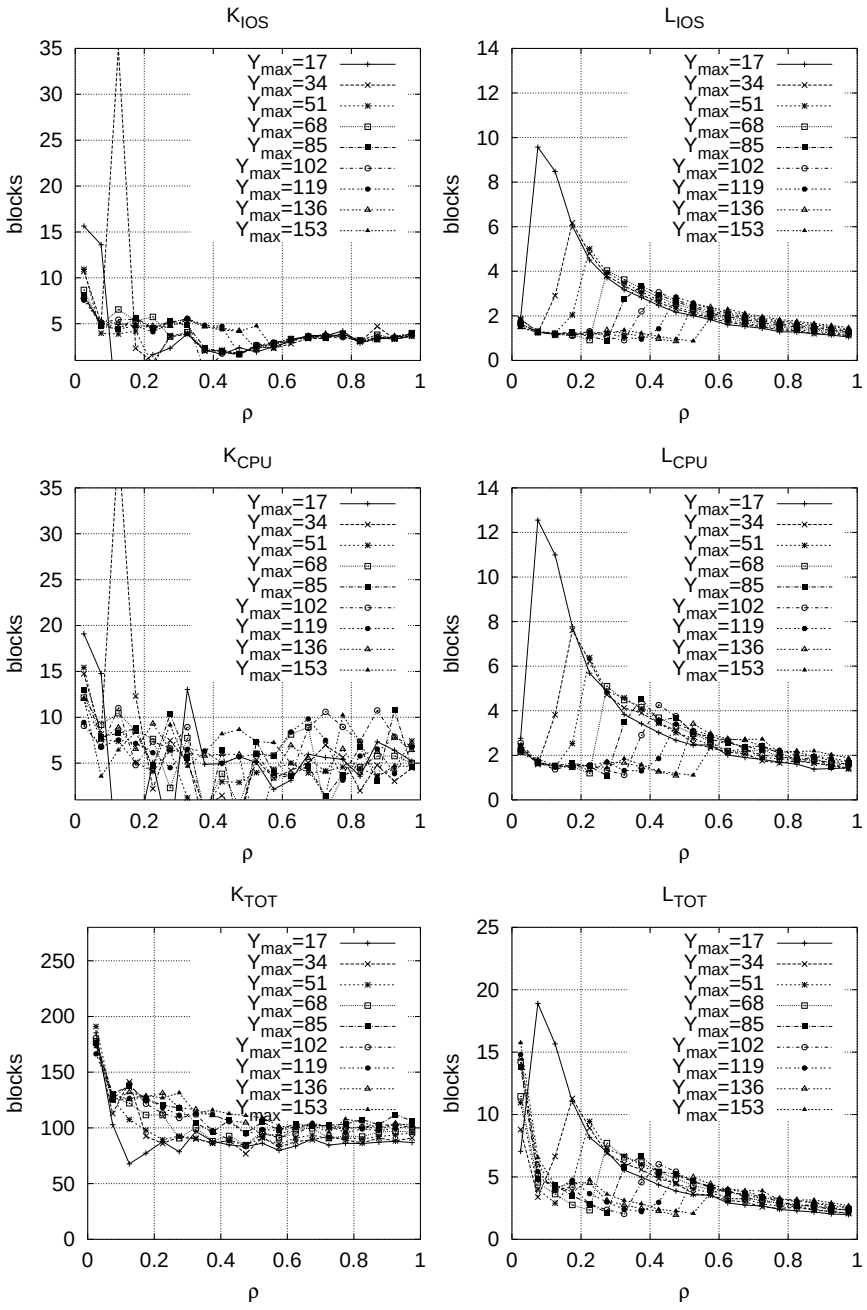

Figure 7.8: The effect of ρ on IOS, CPU and TOT.

7.4 Update performance

The update performance of EPS-trees is of significant practical importance. As discussed in the previous chapter, EPS-trees can have $O(\log_B n)$ update cost, either amortized or worst-case, depending on implementation choices. In this section, we study the average-case performance, and we contrast it with that of the B+-tree.

Although EPS-tree insertion is a relatively complicated process, the I/O cost of insertion is not expected to be significantly higher than that of the B+-tree. Certainly, it will not be less, since an EPS-tree insertion will include the B+-tree insertion step. This step will generally require only 1-2 I/Os, if the main-memory buffer is adequately large to fit most internal nodes. The second step of EPS-tree insertion consists of maintaining Y-set invariants, and the third step is rebalancing in the case of B+-tree node splits. Both of these last two steps may cause multiple Child Cache updates, and possibly leaf updates, as a result of trickle-down or bubble-up operations. However, Child Cache updates typically require only a write to the update block. The number of update blocks is equal to the number of internal nodes. Thus, it is likely that the buffer will fit most update blocks, and thus many of these I/Os can be avoided. For this reason, we expect EPS-tree insertions to be quite efficient.

In order to validate this hypothesis, we constructed an EPS-tree over a dataset of 100,000,000 keys, and inserted an additional 100,000 keys into it. We measured CPU and total time, block accesses, and real I/Os per insertion. We also constructed a B+-tree over the same dataset (ordered by the x-coordinates of the points) and inserted the same 100,000 elements into the B+-tree. The

results of these experiments are summarized in Fig. 7.9. As can be seen,

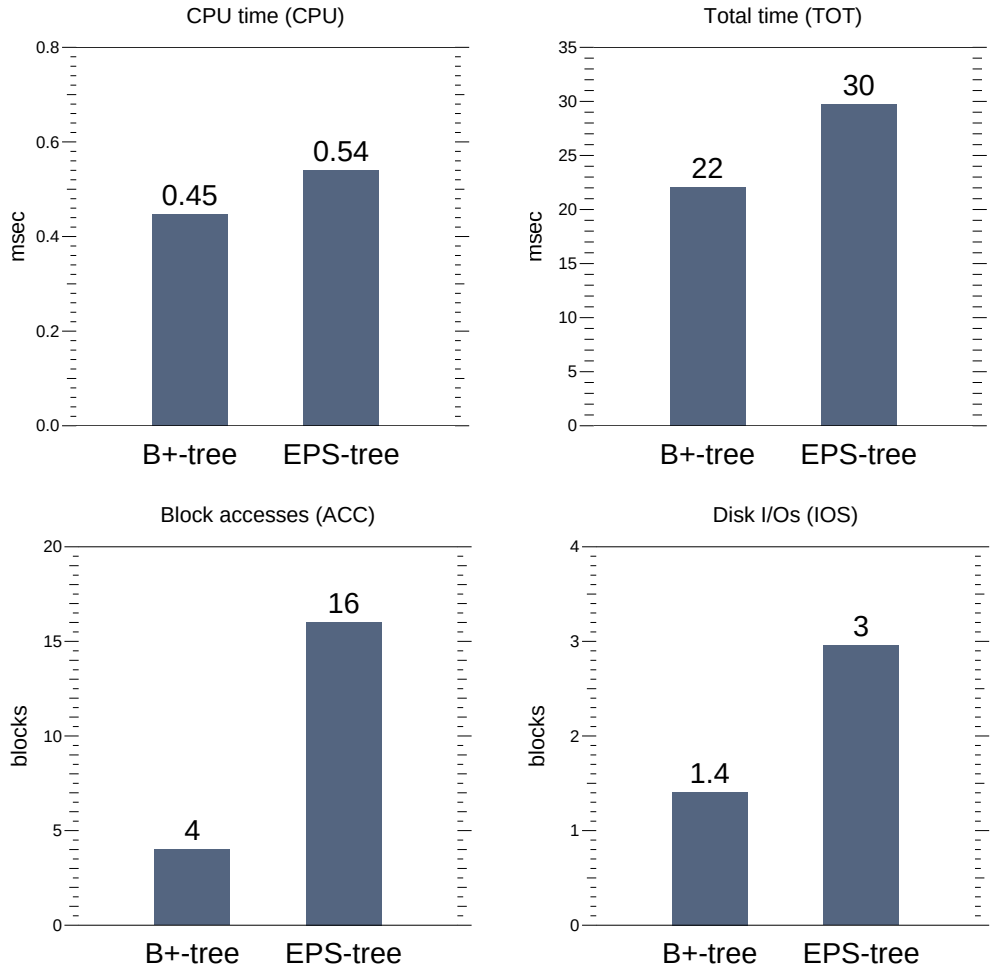

Figure 7.9: EPS-tree update performance.

although the number of block accesses (ACC) is much larger for the EPS-tree than for the B+-tree, the overhead for disk I/Os (IOS) and total time (TOT) are not much higher. In fact, the difference in total time is much smaller than expected, if the additional I/Os of the EPS-tree were random. This is explained as follows: our constructed EPS-tree had strong sequential locality. The insertions performed were not sufficient to destroy that locality

significantly. Thus, the disk head did not have to move far for the additional EPS-tree I/Os, and likely it would not have to move to a different track. Thus, only rotational latency was paid most of the time.

Another conclusion from our experiments is that the amortized-cost implementation of EPS-trees is preferable to the worst-case one. Child Cache rebuilding happens relatively infrequently, and because most of the I/Os are sequential, does not impact the total time significantly when it occurs. By contrast, the worst-case implementation requires twice the number of I/Os, and even worse, these I/Os will be mostly random. Thus, the worst-case implementation could have a big negative impact on performance.

7.5 Implementation Complexity

The EPS-tree is capable of achieving very good query performance. Traditionally though, performance has only been one of the concerns of the database industry, when selecting multidimensional index structures to implement in products. Most of the time, the industry has leaned towards the simplest index structures, even when alternatives were known to be significantly more efficient.

In this section, we discuss the implementation complexity of our EPS-tree implementation. Unfortunately, there is no concrete, objective measure of complexity. Thus, we will limit our discussion in describing our code at a high level, and will only provide some—admittedly crude—measures of code size.

Our implementation was based on EMIL, the External Memory Infrastructure Library. EMIL is a C++ library, which provides an object-oriented

API to the programmer of external data structures and algorithms. To the extent possible, EMIL strives to abstract all explicit I/O operations. However, the block-based nature of external memory is not hidden by EMIL. Instead, EMIL provides facilities for organizing individual blocks, as if they were C++ objects. Thus, an internal B+-tree node would be constructed as an object, with a few scalar and two array attributes (which would correspond to an array of keys, and an array of pointers to child nodes). By treating a block as an object, the coding of search algorithms is greatly simplified, while the programmer retains full control over those aspects of the processing that are critical to high I/O performance. EMIL exposes an interface built around two primitive operations:

1. A disk space allocation interface, similar to C++ `new` and `delete` operators, with which the user can allocate blocks.

2. A "smart-pointer" facility, which hides I/O operations behind pointer accesses.

In order to make a concrete evaluation of the implementation complexity of the EPS-tree, we will first describe an implementation of the B+-tree. Using EMIL, we implemented a templetized B+-tree structure, in 643 lines of C++ code. This is a full implementation, including B+-tree insertion and deletion, an iterator interface for range queries, similar to that of the C++ Standard Template Library, and a bulk-loading operation. A rough break-down of the code in terms of functionality is shown in Table 7.1.

To the extent that code size is related to complexity, the EPS-tree is significantly more complicated than the B+-tree. The relevant break-down is given in Table 7.2. The bulk of the implementation effort was spent in two

Code function	Size (lines of code)
Class declarations/initialization	90
Insertion/deletion	250
Query	115
Bulk loader	150
Miscellaneous (comments etc.)	38
Total	643

Table 7.1: Implementation size of the components of a B+-tree.

Code function	Size (lines of code)
Class declarations/initialization	870
Child cache Updates	530
Updates other than in Child Cache	590
Child cache query	70
EPS-tree query (except in Child Cache)	210
Bulk loader	361
Miscellaneous (comments etc.)	100
Total	2731

Table 7.2: Implementation size of the components of an EPS-tree.

areas; class declarations and initialization (870 lines), and updates (1120 lines). A few comments are in order. First, the EPS-tree required significantly more code for class declarations. Most of this code was used to specify the different disk block formats required in an EPS-tree. Indeed, where the B+-tree only has 2 kinds of blocks (internal and leaf blocks), the EPS-tree has an additional 3 kinds (CC update block, CC catalog and CC data blocks). Also, some of these blocks (like the CC catalog block for example) have a rich structure, and thus require more code to specify. However, the biggest amount of code by far, is related to update processing. This reflects the complexity of the update

operations described in the previous chapter.

Thus, we must conclude that the EPS-tree has non-trivial implementation complexity. However, this implementation complexity is not, in our opinion, prohibitive. We believe that the performance benefits to be reaped, outbalance the added cost of implementation, especially if the EPS-tree is used to implement fundamental language features of a new data model.

7.6 Discussion

The experiments performed in this section complement the analytical results of the previous chapter. They indicate that search and update performance of the EPS-tree is very good, and despite the increased implementation complexity, the EPS-tree is a very good candidate for three-sided range search indexing. At a higher level, the results of this chapter help to support the case for using redundancy for increasing index performance. The space-I/O trade-offs studied analytically in the context of indexability, come into effect and impact performance in the theoretically predicted way.

In particular with respect to indexability, our experiments validate our claim that the contribution of search in the actual I/O cost is small, in the presence of a main-memory buffer.

Chapter 8

Conclusions

In this dissertation, we showed that provably efficient access methods are a good candidate for serving the indexing needs of new data models. Despite their increased complexity, they have superior performance, scalability, and robustness, compared to access methods based on ad hoc assumptions about the workload.

We adopted the indexability model to the study of multidimensional range search in external memory. Indexability can be very useful as a simplifying tool in developing access methods for these problems. It can be seen as an intermediate step, from the problem statement to the actual access method, which focuses attention on the clustering aspects of the problem, ignoring the aspects of search. Its introduction leads to a simpler, more structured argument, and allows for the exploration of more implementation alternatives.

8.1 Main contributions

Our contributions are both to the theory of indexing, as well as to practical concerns. The theoretical part of our work included the development of indexability results for many types of range search, with emphasis on two-dimensional problems. We managed to derive flexible indexing schemes, able to trade space for I/O cost and vice versa. Notable contributions include the first optimal indexing scheme for two-dimensional range search, and solving the long-standing open problem of three-sided range search in external memory.

We also developed comprehensive techniques for lower bounds on these trade-offs, and proved many of our techniques to be optimal. Our main contribution is the discovery of a powerful theorem, which allows the study of lower bounds to focus on combinatorial aspects of the problem at hand, with little concern for issues related to external memory.

The practical part of our work focused on the application of indexability to the design of access methods, and mainly on the design and empirical evaluation of the EPS-tree, an asymptotically optimal access method for three-sided queries, with efficient dynamic behavior. The EPS-tree is more general, and asymptotically more efficient, than a number of previous access methods for similar problems. We also carried out the first empirical study of an access method with provably good access cost, and we demonstrated that our techniques exhibit superior performance, and thus offer a valuable alternative to existing ad hoc techniques. Our experimental results should encourage similar studies of other access methods with asymptotic efficiency guarantees.

8.2 Future work

A number of questions, both theoretical and practical, have emerged as a result of our work. We briefly mention some of the main ones.

Our results on the two-dimensional point enclosure problem have identified a remarkable duality of space-I/O trade-off between dual range search problems. We were unable to discover any general laws related to this phenomenon, but we conjecture that such laws are there to be discovered. Results in this direction could have applications in other scientific areas, such as in geometric discrepancy theory and computational geometry.

An important line of work should be a treatment of three and four-dimensional range search within the indexability context. Currently, only (probably) suboptimal solutions are known. Results in this direction would not only have significant practical impact, but would also generate new insights for generalizing external range search to arbitrary dimensions.

On the practical side, there are two major areas of future work. First, a concerted effort to simplify the techniques developed in this work, and particularly the update algorithms. Such developments are common in indexing, where an initial breakthrough on a hard problem is followed by a number of refinements that reduce the complexity of the original work, and make it more appealing for practical adoption. The second avenue of work is the empirical evaluation of access methods with non-linear space requirements, such as, an access method for two-dimensional range search.

Appendix A

Manipulations for §4.4

Let

$$r = \frac{A - 2c + 10}{A - 2c + 2} \cdot \frac{\log(n/AB)}{\log c} \tag{A.1}$$

We wish to solve equation $\dfrac{\partial r}{\partial c} = 0$. This is equivalent to

$$\frac{\partial}{\partial c}\Big(\frac{A - 2c + 10}{(A - 2c + 2)\ln c}\Big) = 0$$

We have

$$\begin{aligned}
\frac{\partial}{\partial c}\Big(\frac{A - 2c + 10}{A - 2c + 2} \cdot \frac{1}{\ln c}\Big) &= \frac{\partial}{\partial c}\Big(\frac{A - 2c + 10}{A - 2c + 2}\Big) \cdot \frac{1}{\ln c} + \frac{A - 2c + 10}{A - 2c + 2} \cdot \frac{\partial(1/\ln c)}{\partial c} \\
&= \frac{16}{(A - 2c + 2)^2 \ln c} - \frac{A - 2c + 10}{(A - 2c + 2)c \ln c} \\
&= 0
\end{aligned}$$

and thus

$$\begin{aligned} 16c\ln c &= (A+10-2c)(A+2-2c) \\ &= (A+6-2c+4)(A+6+2c-4) \\ &= (A+6-2c)^2-16 \end{aligned}$$

Thus,

$$16(c\ln c+1) = (A+6-2c)^2$$

By solving with respect to A, we obtain

$$A = 2(c-2)+4\sqrt{c\ln c+1}-2 \qquad \text{(A.2)}$$

Bibliography

[AAE+98] P. K. Agarwal, L. Arge, J. Erickson, P. Franciosa, and J. Vitter. Efficient searching with linear constraints. In *Proc. ACM Symp. Principles of Database Systems*, pages 169–178, 1998.

[AAE00] P.K. Agarwal, L. Arge, and J. Erickson. Indexing moving points. In *Proc. ACM Symp. Principles of Database Systems*, pages 175–186, 2000.

[ABR00] S. Alstrup, G. Brodal, and T. Rauhe. New data structures for orthogonal range searching. In *Proc. IEEE Symp. on Foundations of Comp. Sci.*, pages 198–207, 2000.

[AE97] P. K. Agarwal and J. Erickson. Geometric range searching and its relatives. In B. Chazelle, E. Goodman, and R. Pollack, editors, *Discrete and Computational Geometry: Ten Years Later*. Mathematical Society Press, 1997.

[Aok99] P. Aoki. How to avoid building datablades that know the value of everything and the cost of nothing. In *Proc. 11th IEEE Int'l Conf. on Scientific and Statistical Database Mgmt.*, pages 122–133, 1999.

[ASV99] L. Arge, V. Samoladas, and J.S. Vitter. On two-dimensional indexability and optimal range search indexing. In *Proc. ACM Symp. Principles of Database Systems*, 1999.

[AV96] L. Arge and J. S. Vitter. Optimal dynamic interval management in external memory. In *Proc. IEEE Symp. on Foundations of Comp. Sci.*, pages 560–569, 1996.

[Ben75] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18:509–517, 1975.

[Ben80] J. L. Bentley. Multidimensional divide and conquer. *Communications of the ACM*, 23(6):214–229, 1980.

[BG90] G. Blankenagel and R. H. Güting. XP-trees—External priority search trees. Technical report, FernUniversität Hagen, Informatik-Bericht Nr. 92, 1990.

[BGO+96] B. Becker, S. Gschwind, T. Ohler, B. Seeger, and P. Widmayer. An asymptotically optimal multiversion B-tree. *VLDB Journal*, 5(4):264–275, 1996.

[BKSS90] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: An efficient and robust access method for points and rectangles. In *Proc. SIGMOD Intl. Conf. on Management of Data*, pages 322–331, 1990.

[BM70] Rudolf Bayer and Edward M. McCreight. Organization and maintenance of large ordered indexes. In *Record of the 1970 ACM SIGFIDET Workshop on Data Description and Access, November 15-*

16, 1970, Rice University, Houston, Texas, USA (Second Edition with an Appendix), pages 107–141. ACM, 1970.

[BM72] R. Bayer and E. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1:173–189, 1972.

[CE85] B. Chazelle and H. Edelsbrunner. Optimal solutions for a class of point retrieval problems. *Journal of Symbolic Computing*, 1(1):47–56, 1985.

[CE87] B. Chazelle and H. Edelsbrunner. Linear space data structures for two types of range search. *Discrete Comput. Geom.*, 2:113–126, 1987.

[CG86a] B. Chazelle and L. J. Guibas. Fractional cascading: I. A data structuring technique. *Algorithmica*, 1:133–162, 1986.

[CG86b] B. Chazelle and L. J. Guibas. Fractional cascading: II.Applications. *Algorithmica*, 1:163–191, 1986.

[CGL85] B. Chazelle, L.J. Guibas, and D.T. Lee. The power of geometric duality. *BIT*, 25(1):76–90, 1985.

[Cha86] B. Chazelle. Filtering search: a new approach to query answering. *SIAM Journal of Computing*, 15(3):703–724, 1986.

[Cha88] B. Chazelle. A functional approach to data structures and its use in multidimensional searching. *SIAM Journal of Computing*, 17(3):427–462, 1988.

[Cha90a] B. Chazelle. Lower bounds for orthogonal range searching, i: the reporting case. *Journal of the ACM*, 37:200–212, 1990.

[Cha90b] B. Chazelle. Lower bounds for orthogonal range searching, ii: the arithmetic model. *Journal of the ACM*, 37:439–463, 1990.

[Cha95] B. Chazelle. Lower bounds for off-line range searching. In *Proc. ACM Symp. on Theory of Computation*, pages 733–740, 1995.

[CLRS86] B. Chor, C.E. Leiserson, R.L. Rivest, and J.B. Shearer. An application of number theory to the organization of raster-graphics memory. *Journal of the ACM*, 33(1):86–104, 1986.

[Cod70] E. F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

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

[DRSS96] A. A. Diwan, Sanjeeva Rane, S. Seshadri, and S. Sudarshan. Clustering techniques for minimizing external path length. In *Proc. IEEE International Conf. on Very Large Databases*, pages 342–353, 1996.

[DSST89] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. Tarjan. Making data structures persistant. *Journal of Computer and System Sciences*, 38:86–124, 1989.

[FB74] R.A. Finkel and J.L. Bentley. Quad trees: a data structure for retrieval on composite keys. *Acta Inform.*, 4:1–9, 1974.

[FK94] C. Faloutsos and I. Kamel. Beyond uniformity and independence: Analysis of R- trees using the concept of fractal dimension. In *Proc. ACM Symp. Principles of Database Systems*, pages 4–13, 1994.

[FNPS79] R. Fagin, J. Nievergelt, N. Pippenger, and H.R. Strong. Extendible hashing – a fast access method for dynamic files. *ACM Transactions on Database Systems*, 4(3):315–344, 1979.

[FR91] C. Faloutsos and Y. Rong. DOT: A spatial access method using fractals. In *Proc. IEEE Intl. Conf. Data Engineering*, pages 152–159, 1991.

[Fre80] M.L. Fredman. The inherent complexity of dynamic data structures which accomodate range queries. In *Proc. IEEE Symp. on Foundations of Comp. Sci.*, pages 191–199, 1980.

[Fre81] M.L. Fredman. Lower bounds on the complexity of some optimal data structures. *SIAM Journal of Computing*, 10:1–10, 1981.

[Fre87] M. Freeston. The bang file: a new kind of grid file. In *Proc. SIGMOD Intl. Conf. on Management of Data*, pages 260–269, 1987.

[FS89] A. Fiat and A. Shamir. How to find a battleship. *Networks*, 19:361–371, 1989.

[Gae95] V. Gaede. Optimal redundancy in spatial database systems. In *Proc. 4th Int. Symposium on Spatial Databases*, pages 96–116, 1995.

[GG98] V. Gaede and O. Günther. Multidimensional access methods. *Computing Surveys*, 30(2):170–231, June 1998.

[GS78] L. J. Guibas and R. Sedgewick. A dichromatic framework for balanced trees. In *Proc. IEEE Symp. on Foundations of Comp. Sci.*, pages 8–21, 1978.

[Gut85] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proc. SIGMOD Intl. Conf. on Management of Data*, pages 47–57, 1985.

[Hin85] K. Hinrichs. Implementation of the grid file: design concepts and experience. *BIT*, 25:569–592, 1985.

[HKP97] J.M. Hellerstein, E. Koutsoupias, and C.H. Papadimitriou. On the analysis of indexing schemes. In *Proc. ACM Symp. Principles of Database Systems*, 1997.

[HNP95] J.M. Hellerstein, J.E. Naughton, and A. Pfeffer. Generalized search trees for database systems. In *Proceedings of the 21st VLDB Conference*, 1995.

[HSW89] A. Henrich, H.-W. Six, and P. Widmayer. The lsd tree: spatial access to multidimensional point and non-point objects. In *Proc. IEEE International Conf. on Very Large Databases*, pages 45–53, 1989.

[HW87] D. Haussler and E. Welzl. ε-nets and simplex range queries. *Discr. Comput. Geom.*, 2:127–151, 1987.

[IKO87] Ch. Icking, R. Klein, and Th. Ottmann. Priority search trees in secondary memory. In *Proc. Graph-Theoretic Concepts in Computer Science, LNCS 314*, pages 84–93, 1987.

[Jag90] H.V. Jagadish. Spatial search with polyhedra. In *Proc. IEEE Intl. Conf. Data Engineering*, pages 311–319, 1990.

[Joh62] S.M. Johnson. A new upper bound for error-correcting codes. *IEEE Trans. Information Theory*, 8:203–207, 1962.

[JW96] J.H. van Lint and R.M. Wilson. *A Course in Combinatorics*. Cambridge University Press, 1996.

[KF94] I. Kamel and C. Faloutsos. Hilbert R-tree: An improved R-tree using fractals. In *Proceedings 20th International Conference on Very Large Databases*, pages 500–509, 1994.

[KGT99] G. Kollios, D. Gunopoulos, and V.J. Tsotras. On indexing mobile objects. In *Proc. ACM Symp. Principles of Database Systems*, pages 261–272, 1999.

[KPS00] H. Kriegel, M. Potke, and T. Seidl. Managing intervals efficiently in object-relational databases. In *Proc. IEEE International Conf. on Very Large Databases*, pages 407–418, 2000.

[KRV+93] P. C. Kanellakis, S. Ramaswamy, D. Vengroff, E., and J. S. Vitter. Indexing for data models with constraints and classes. In *Proc. ACM Symp. Principles of Database Systems*, pages 233–243, 1993.

[KS89] K.V. Ravi Kanth and A.K. Singh. Optimal dynamic range searching in non-replicating index structures. In *Proc. Intl. Conf. Database Theory*, pages 257–276, 1989.

[KS91] C.P. Kolovson and M. Stonebraker. Segment indexes: Dynamic indexing techniques for multi-dimensional interval data. In *Proc.*

SIGMOD Intl. Conf. on Management of Data, pages 138–147, 1991.

[KT98] E. Koutsoupias and David S. Taylor. Tight bounds for 2-dimensional indexing schemes. In *Proc. ACM Symp. Principles of Database Systems*, pages 52–58, 1998.

[KT99] E. Koutsoupias and David S. Taylor. Indexing schemes for random points. In *Proc. ACM-SIAM Symp. on Discrete Algorithms*, pages 17–19, 1999.

[KTF98] A. Kumar, V.J. Tsotras, and C. Faloutsos. Designing access methods for bitemporal databases. *IEEE Transactions on Knowledge and Data Engineering*, 10(1):1–20, 1998.

[LM91] S. Lanka and E. Mays. Fully persistent B+trees. In *Proc. SIGMOD Intl. Conf. on Management of Data*, pages 426–435, 1991.

[LS90] D.B. Lomet and B. Salzberg. The hB-tree: A multiattribute indexing method with good guaranteed performance. *ACM Transactions on Database Systems*, 15(4):625–658, 1990.

[Lue78] G.S. Lueker. A data structure for orthogonal range queries. In *Proc. IEEE Symp. on Foundations of Comp. Sci.*, pages 28–34, 1978.

[LW80] D.T. Lee and C.K. Wong. Quintary trees: a file structure for multidimensional database systems. *ACM Transactions on Database Systems*, 5:339–353, 1980.

[Mat92] J. Matoušek. Reporting points in halfspaces. *Comput. Geom. Theory Appl.*, 2:169–186, 1992.

[Mat94] J. Matoušek. Geometric range searching. *ACM Computing Surveys*, 26(4):422–461, 1994.

[Mat99] J. Matoušek. *Geometric Discrepancy: an illustrated guide*, volume 18 of *Algorithms and Combinatorics*. Springer, 1999.

[McC85] E.M. McCreight. Priority search trees. *SIAM Journal of Computing*, 14(2):257–276, 1985.

[Meh84] K. Mehlhorn. *Data Structures and Algorithms 3: Multi-dimensional Searching and Computational Geometry*. Springer-Verlag, EATCS Monographs on Theoretical Computer Science, 1984.

[MT98] C. Makris and A.K. Tsakalidis. Algorithms for three-dimensional dominance searching in linear space. *Information Processing Letters*, 66(6):277–283, 1998.

[MTT99] Y. Manolopoulos, Y. Theodoridis, and V.J. Tsotras. *Advanced Database Indexing*. Kluwer Academic Publishers, 1999.

[NGV96] M. H. Nodine, M. T. Goodrich, and J. S. Vitter. Blocking for external graph searching. *Algorithmica*, 16(2):181–214, 1996.

[NHS84] J. Nievergelt, H. Hinterberger, and K.C. Sevcik. The grid file: An adaptable, symmetric multikey file structure. *ACM Transactions on Database Systems*, 9(1):257–276, 1984.

[OM84] J. A. Orenstein and T. H. Merrett. A class of data structures for associative searching. In *Proc. ACM Symp. Principles of Database Systems*, pages 181–190, 1984.

[Oos90] P. Oosterom. *Reactive data structures for geographic information systems*. PhD thesis, University of Leiden, The Netherlands, 1990.

[Ove83] Mark H. Overmars. *The Design of Dynamic Data Structures*. Springer-Verlag, LNCS 156, 1983.

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

[PST+93] B.-U. Pagel, H.-W. Six, H. Toben, , and P. Widmayer. Towards an analysis of range query performance in spatial data structures. In *Proc. ACM Symp. Principles of Database Systems*, pages 214–221, May 1993.

[Ram97] S. Ramaswamy. Efficient indexing for constraint and temporal databases. In *Proc. Intl. Conf. Database Theory*, pages 419–431, 1997.

[RL85] N. Roussopoulos and D. Leifker. Direct spatial search on pictorial databases using packed R-trees. In *Proc. SIGMOD Intl. Conf. on Management of Data*, pages 17–31, 1985.

[Rob84] J.T. Robinson. The K-D-B tree: A search structure for large multidimensional dynamic indexes. In *Proc. ACM SIGMOD Conf. on Management of Data*, pages 10–18, 1984.

[RS94] S. Ramaswamy and S. Subramanian. Path caching: A technique for optimal external searching. In *Proc. ACM Symp. Principles of Database Systems*, pages 25–35, 1994.

[Sam84] H. Samet. The quadtree and related hierarchical data structures. *ACM Computing Surveys*, 16(2):187–260, 1984.

[Sam89a] H. Samet. *Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS*. Addison Wesley, MA, 1989.

[Sam89b] H. Samet. *The Design and Analyses of Spatial Data Structures*. Addison Wesley, MA, 1989.

[SKH99] M.A. Shah, M. Kornacker, and J.M. Hellerstein. Amdb: A visual access method development tool. In *User Interfaces for Data Intensive Systems (UIDIS)*, 1999.

[SM98] V. Samoladas and D. Miranker. A lower bound bound theorem for indexing schemes and its application to multidimensional range queries. In *Proc. ACM Symp. Principles of Database Systems*, pages 44–51, 1998.

[SR95] S. Subramanian and S. Ramaswamy. The P-range tree: A new data structure for range searching in secondary memory. In *Proc. ACM-SIAM Symp. on Discrete Algorithms*, pages 378–387, 1995.

[SRF87] T. Sellis, N. Roussopoulos, and C. Faloutsos. The R^+-tree: A dynamic index for multi-dimensional objects. In *Proc. IEEE International Conf. on Very Large Databases*, 1987.

[ST86] N. Sarnak and R. E. Tarjan. Planar point location using persistent search trees. *Communications of the ACM*, 29:669–679, 1986.

[ST99] B. Salzberg and V.J. Tsotras. Comparison of access methods for time-evolving data. *ACM Computing Surveys*, 31(2):158–221, 1999.

[TSS00] Y. Theodoridis, E. Stefanakis, and T. Sellis. Efficient cost models for spatial queries using r-trees. *IEEE Transactions on Knowledge and Data Engineering*, 12(1):19–32, 2000.

[Tur41] P. Turán. An extremal problem in graph theory (in hungarian). *Mat. Fiz. Lapok.*, 48:435–452, 1941.

[TVM98] T. Tzouramanis, M. Vassilakopoulos, and Y. Manolopoulos. Overlapping linear quadtrees: A spatio-temporal access method. In *Proc. Advances in Geographic Information Systems*, pages 1–7, 1998.

[Vai89] P.M. Vaidya. Space-time trade-offs for orthogonal range queries. *SIAM Journal of Computing*, 18:748–758, 1989.

[Vaj89] S. Vajda. *Fibonacci and Lucas numbers, and the Golden Section.* Mathematics and its Applications. Ellis Horwood Ltd., 1989.

[Vit99] J. S. Vitter. External memory algorithms and data structures. In J. Abello and J. S. Vitter, editors, *External Memory Algorithms and Visualization.* American Mathematical Society Press, Providence, RI, 1999.

[VS94] J. S. Vitter and E. A. M. Shriver. Algorithms for parallel memory, I: Two-level memories. *Algorithmica*, 12(2–3):110–147, 1994.

[VV96a] D. E. Vengroff and J. S. Vitter. Efficient 3-d range searching in external memory. In *Proc. ACM Symp. on Theory of Computation*, pages 192–201, 1996.

[VV96b] Darren Erick Vengroff and Jeffrey Scott Vitter. Efficient 3-D range searching in external memory. In *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing*, pages 192–201, 22–24 May 1996.

[VV97] P. J. Varman and R. M. Verman. An efficient multiversion access structure. *IEEE Transactions on Knowledge and Data Engineering*, 9(3):391–409, 1997.

[Wil78] D. E. Willard. *Predicate-oriented database search algorithms*. PhD thesis, Harvard Univ., Cambridge, MA, 1978.

[Wil82] D.E. Willard. Polygon retrieval. *SIAM Journal of Computing*, 11(1):149–165, 1982.

[WL85] D.E. Willard and G.S. Lueker. Adding range restriction capability to dynamic data structures. *Journal of the ACM*, 32(3):597–617, 1985.

[Yao82] A.C. Yao. Space-time trade-off for answering range queries. In *Proc. ACM Symp. on Theory of Computation*, pages 128–136, 1982.

Vita

Vasilis Samoladas was born in Thessaloniki, Greece in 1969. He grew up in his home city. From 1987 to 1992, he studied electrical engineering at the Aristotle University of Thessaloniki. He graduated with an engineering diploma in 1992. His diploma thesis was on VLSI architectures for fuzzy logic controllers. He became a fully licensed engineer in the summer of 1992. For the next year, he worked as a research engineer for a number of projects funded by the European Union. In 1993, he joined the Masters program of the Department of Computer Sciences of the University of Texas at Austin. After he received his M.Sc. in 1995, he remained to pursue a Ph.D. in the same department, under the supervision of Dan Miranker. His initial interests focused on database integration. In 1998 he joined MCC in Austin, on a part-time basis, and worked with the Infosleuth group on agent-based information systems. His work at MCC lasted for 20 months, until the company was dissolved. His research interests focused on theoretical aspects of indexing. In 1998, his results on lower bounds received the Best Paper and Best Newcomer awards of the ACM SIGMOD Symposium on Principles of Database Systems. During his graduate studies, he worked as a Teaching Assistant and as a Research Assistant, until his graduation, in August 2001.

Permanent Address: Hatzikyriakou 4,Thessaloniki, 54643 Greece

This dissertation was typed by the author.