Indexing for Data Models with Classes and Constraints

by

Sridhar Ramaswamy

B. Tech., Indian Institute of Technology, Madras, July 1989

Sc. M., Brown University, May 1991

Thesis

Submitted in partial fulfillment of the requirements for the

Degree of Doctor of Philosophy in the Department of Computer Science

at Brown University.

May 1995

This dissertation by Sridhar Ramaswamy
is accepted in its present form by the Department of
Computer Science as satisfying the
dissertation requirement for the degree of Doctor of Philosophy.

Date 9/13/94

Paris C. Kanellakis

Recommended to the Graduate Council

Date 9/13/94

Stanley B. Zdonik

Date 9/13/94

Thomas W. Doeppner Jr.

Approved by the Graduate Council

Date 10/6/94

Vita

I was born on February 26, 1967 in Tiruchirapally, India. After attending high school in the National College, Bangalore, India, I studied computer science and engineering at the Indian Institute of Technology, Madras, India, from which I graduated in 1989 with the Bachelor of Technology degree. I then joined the Ph.D. program in computer science at Brown University, where I received an Sc.M. degree in 1991.

Abstract

We examine I/O-efficient data structures that provide indexing support for new data models. The database languages of these models include concepts from constraint programming (e.g., relational tuples are generalized to conjunctions of constraints) and from object-oriented programming (e.g., objects are organized in class hierarchies).

Indexing by one attribute in many constraint data models is equivalent to external dynamic interval management, which is a special case of external dynamic two-dimensional range searching. We present the first optimal worst-case solution for the static version of this problem.

Indexing by one attribute and class name in an object-oriented database (OODB) in which objects are organized as a forest hierarchy of classes is also a special case of external dynamic two-dimensional range searching. On the basis of this observation, we first identify a simple algorithm, called the *class-divisions* algorithm, with provably good worst-case performance for this problem. Using the forest structure of the class hierarchy and techniques used in solving the constraint indexing problem, we improve its query I/O time.

We then introduce a technique called *path caching* that can be used to convert many main memory data structures for the problems we study to efficient secondary storage data structures. By doing so, we examine important time/space tradeoffs in secondary storage.

Finally, we present the experimental evaluation of the class-divisions algorithm for

indexing in OODBs by comparing it with the algorithm commonly adopted in OODBs. We prove that by paying a small overhead in storage (typically less than a factor of 3), impressive performance gains in querying (typically a factor of over 8) are made possible. By doing this, we demonstrate that algorithms can be designed with good guaranteed bounds for difficult indexing problems in databases.

Credits

Part of the research described in this thesis was done in collaboration with Paris Kanellakis, Sairam Subramanian, Darren Vengroff, and Jeffrey Vitter. In particular, the material in Chapters 1–5 was co-authored with Paris Kanellakis, Darren Vengroff and Jeffrey Vitter and appeared in [21]. The material in Chapter 6 was co-authored with Sairam Subramanian and appeared in [31].

While I was at Brown, I was supported by ONR grant N00014-91-J-4052, ARPA Order 8225 and by ONR grant N00014-91-J-1603. I gratefully acknowledge this support.

Acknowledgements

I would like to thank my advisor, Paris Kanellakis, for all that he has done for me at Brown. He has been much more than just a thesis advisor for me. He has been a warm friend, an astute career counselor and a colleague with a far-reaching vision. I am very grateful to him. In particular, he introduced the field of indexing to me, and it was in discussions about indexing that the central ideas of this thesis germinated.

My thanks go to my Master's thesis advisor, Stan Zdonik, for introducing me to research in computer science and for being very considerate with a very raw, and often very frustrated, researcher. My thanks go to Tom Doeppner for being very helpful in the two courses for which I was his teaching assistant. In addition, my thanks go to Stan and Tom for being on my thesis committee.

My co-authors, Prof. Jeff Vitter, Dr. S. Sairam, and Darren Vengroff, get my thanks for many stimulating discussions. Sai's unceasing energy and enthusiasm for research have been a great inspiration to me.

I would like to thank the departmental technical staff for providing the best work environment that I have seen in my life. (Thanks here should also go to Bob Cohen, Ashim Garg and Lloyd Greenwald whose sporadic presence in the office greatly helped my monopolization of a workstation.) The administrative staff, in particular Mary Andrade, Katrina Avery, Tina Cantor, Patti Courtemanche, Jennet Kirschenbaum, and Dawn Nicholaus, get my thanks for their help in dealing with the university bureaucracy and the many day-to-day

matters.

I would like to thank my friend and roommate Matthias Wloka for being a wonderful roommate. Our apartment has always been a happy and welcome refuge for me away from school and he deserves many thanks for that.

My many friends and acquaintances at Brown have contributed immensely to and have enriched my life at Brown. I would like to thank Swarup Acharya, Ajit Agrawal, the Anjalis, Rudrani Banik, Ravi Bellamkonda, Randy Calistri, Bob Cohen, Anne Fitzpatrick, Ashim Garg, Herb, Gerd Hillebrand, P. Krishnan, David Langworthy, Jenny Lowe, Uma Madapur, Alberto Marquez, T. M. Murali, Chip Natarajan, Andrea Pietracaprina, Amy Prashkar, Kavita Ramanan, Prabha, Susanne Raab, Shobhana Raghupathy, R. Rajesh, Viswanath Ramachandran, Ravi Ramamurthy, Ruth Rusch, Craige Ruffin, Fred Shoucair, Sai and Bharathi Subramanian, Sairam Sundaram, Ram Upadrastha, Emily Weymar and the many others who made life interesting and fun.

And finally, I would like to thank my parents, N. P. Ramaswamy and P. R. Namagiri, my sister R. Vijaya, and my girlfriend Seema Bansal for their love and support.

Contents

Chapter 1

Introduction

1.1 Motivation

The successful realization of any data model requires supporting its language features with efficient secondary storage manipulation. For example, the relational data model [10] includes declarative programming in the form of relational calculus and algebra and expresses queries of low data complexity because every fixed relational calculus query can be evaluated in LOGSPACE and PTIME in the size of the input database. More importantly, these language features can be supported by data structures for searching and updating that make optimal use of secondary storage. B-trees and their variants B^+-trees [2,11] are examples of such data structures and have been an unqualified success in supporting *external dynamic one-dimensional range searching* in relational database systems.

The general data structure problem underlying efficient secondary storage manipulation for many data models is *external dynamic k-dimensional range searching.* The problem of k-dimensional range searching in both main memory and secondary memory has been the subject of much research. To date, solutions approaching the worst-case performance of

B-trees for one-dimensional searching have not been found for even the simplest cases of external k-dimensional range searching. In this thesis, we examine new I/O-efficient data structures for special cases of the general problem of k-dimensional range searching. These special cases are important for supporting new language features, such as constraint query languages [20] and class hierarchies in object-oriented databases [23,41].

We make the standard assumption that each secondary memory access transmits one page or B units of data, and we count this as one I/O. (We use the terms *page* and *disk block* interchangeably, as also the terms *in-core* and *main memory*.) We also assume that at least $O(B^2)$ words of main memory are available. This is not an assumption that is normally made, but is entirely reasonable given that B is typically on the order of 10^2 to 10^3 and today's machines have main memories of many megabytes.

Let R be a relation with n tuples and let the output to a query on R have t tuples. We also use n for the number of objects in a class hierarchy with c classes and t for the output size of a query on this hierarchy. The performance of our algorithms is measured in terms of the number of I/Os they need for querying and updating and the number of disk blocks they require for storage. The I/O bounds are expressed in terms of n, c, t and B, i.e., all constants are independent of these four parameters. (For a survey of the state of the art in I/O complexity, see [40].) We first review B$^+$-tree performance since we use that as our point of reference.

A B$^+$-tree on attribute x of the n-tuple relation R uses $O(n/B)$ pages of secondary storage. The following operations define the problem of *external dynamic one-dimensional range searching* on relational database attribute x, with the corresponding I/O time performance bounds using the B$^+$-tree on x: (1) Find all tuples such that for their x attribute $a_1 \leqslant x \leqslant a_2$. If the output size is t tuples, then this range searching takes $O(\log_B n + t/B)$ secondary memory accesses, worst-case. If $a_1 = a_2$ and x is a key, i.e., it uniquely identifies the tuple, then this is key-based searching. (2) Inserting or deleting a given tuple takes $O(\log_B n)$ secondary memory accesses, worst-case. The problem of *external dynamic k-dimensional range searching* on relational database attributes $x_1, \ldots, x_k$ generalizes one-

dimensional range searching to k attributes, with range searching on k-dimensional intervals. If there are no deletes we say that the problem is *semi-dynamic.* If there are no inserts or deletes we say that the problem is *static.*

In this thesis we are concerned with external two-dimensional range searching: dynamic (Chapters 3, 6 and 7), and semi-dynamic and static (Chapters 4 and 5). We are concerned with algorithms that have provably good worst-case I/O bounds.

To put our contributions into perspective we point out (from the literature by using standard mappings of in-core data structures to external ones) that external dynamic two-dimensional range searching, and thus the problems examined here, can be solved in static query I/O time $O(\log_2 n + t/B)$ worst-case using fractional cascading, and in dynamic query I/O time $O(\log_2 n \log_B n + t/B)$ worst-case using $O((n/B)\log_2 n)$ pages. The amortized update I/O time is $O(\log_2 n \log_B n)$. (Note that $\log_2 n = (\log_2 B)(\log_B n)$ is asymptotically much larger than $\log_B n$.) Using the special structure of the indexing problems of interest, we improve the above bounds.

1.2 Indexing Constraints

Constraint programming paradigms are inherently "declarative", since they describe computations by specifying how these computations are constrained. A general constraint programming framework for database query languages called constraint query languages or CQLs was presented in [20]. This framework adapts ideas from constraint logic programming, e.g., from [19], to databases, provides a calculus and algebra, guarantees low data complexity, and is applicable to managing spatial data.

It is, of course, important to index constraints and thus support these new language features with efficient secondary storage manipulation (see Section 3.1 for a detailed exposition of the problem). Fortunately, it is possible to do this by combining CQLs with existing two-dimensional range searching data structures [20]. The basis of this observation

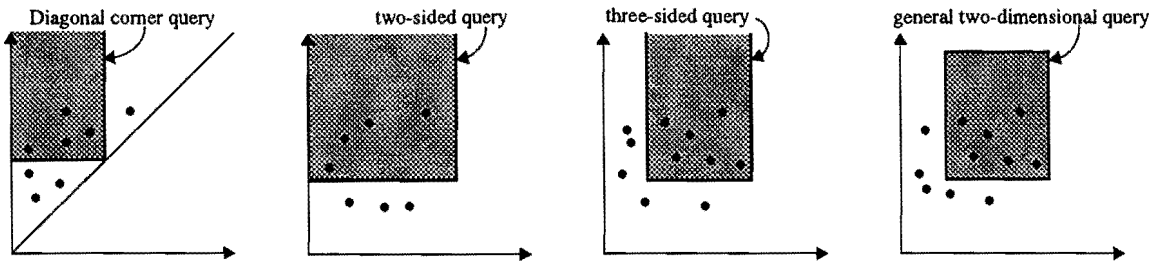


Figure 1.1: Diagonal corner queries, two-sided, three-sided and general two-dimensional range queries.

is a reduction of indexing constraints, for a fairly general class of constraints, to dynamic interval management on secondary storage. Given a set of input intervals, dynamic interval management involves being able to perform the following operations efficiently: (1) Answer interval intersection queries; that is, report all the input intervals that intersect a query interval. (2) Delete or insert intervals from the interval collection. Dynamic interval management can be shown to be a special case of external dynamic two-dimensional range searching.

Dynamic interval management is interesting because it can be solved optimally in-core using the Priority Search Tree of McCreight [27] in query time $O(\log_2 n + t)$, update time $O(\log_2 n)$, and space $O(n)$, which are all optimal. Achieving analogous I/O bounds is much harder. In Section 3.1, we reduce indexing constraints to a special case of external dynamic two-dimensional range searching that involves diagonal corner queries and updates. A *diagonal corner query* is a two-sided range query whose corner must lie on the line $x = y$ and whose query region is the quarter plane above and to the left of the corner (see Figure 1.1). In Chapter 4, we propose a new data structure, which we call the *metablock tree*, for this problem. Our data structure has optimal worst-case space $O(n/B)$ pages, optimal query I/O time $O(\log_B n + t/B)$ and has $O(\log_B n + (\log_B n)^2/B)$ amortized insert I/O time. This performance is optimal for the static case and nearly optimal for the semi-dynamic case where only insertions are allowed (modulo amortization).

1.3 Indexing Classes

Indexing by one attribute and by class name in an object-oriented model, where objects are organized as a static forest hierarchy of classes, is also a special case of external dynamic two-dimensional range searching. Together with the different problem of indexing nested objects, as in [26], it constitutes the basis for indexing in object-oriented databases. Indexing classes has been examined in [22] and more recently in [25], but the solutions offered there are largely heuristic with poor worst-case performance.

In Section 3.2, we reduce indexing classes to a special case of external dynamic two-dimensional range searching called three-sided searching. Three-sided range queries are a special case of two-dimensional range queries. In three-sided range queries, one of the four sides defining the query rectangle is always one of the coordinate axes or at infinity.[1] We also assume that the class-subclass relationship is static, although objects can be inserted or deleted from classes. Under this reasonable assumption, for a class hierarchy with c classes having a total of n objects, we identify a simple algorithm called the *class-divisions* algorithm with worst-case space $O((n/B)\log_2 c)$ pages, query I/O time $O(\log_2 c \log_B n + t/B)$, and update I/O time $O(\log_2 c \log_B n)$. Even with the additional assumption of a static class-subclass relationship, the problem is a nontrivial case of two-dimensional range searching. We show in Section 3.2 that it is impossible to achieve optimal query time for this problem ($O(\log_B n + t/B)$ disk I/Os) with only one copy of each object in secondary storage. (For lower bounds on range searching in main memory, see [14] and [7].) In Chapter 5, analyzing the hierarchy using the hierarchical decomposition of [37] and using techniques from the constraint indexing problem, we improve query I/O time to $O(\log_B n + t/B + \log_2 B)$ using space $O((n/B)\log_2 c)$ pages. Amortized update I/O time for the semi-dynamic problem with inserts is $O(\log_2 c(\log_B n + (\log_B n)^2/B))$.

[1] Note that diagonal corner queries are a special case of two-sided queries and two-sided queries are a special case of three-sided queries (see Figure 1.1).

1.4 Path Caching

Many efficient algorithms exist for two-dimensional range searching and its special cases (see [9] for a detailed survey). Most of these algorithms cannot easily be mapped efficiently to secondary storage. We present a general technique called *path caching* that can be used to map many main memory data structures like the priority search tree [27], segment tree [3], and interval tree [12,13] efficiently to secondary storage. These data structures are relevant and important to the indexing problems we are considering because of the following reasons: (1) Dynamic interval management can be solved efficiently in main memory by all the three data structures mentioned above; and (2) three-sided queries can be solved efficiently in main memory by the priority search tree. The data structures obtained by applying path caching to these main memory data structures will have very efficient query times at the expense of slight overheads in storage. This technique is also simple enough to allow for inserts and deletes in small amortized time.

Using path caching, we present a data structure that implements priority search trees on disk to answer two-sided queries, a generalization of diagonal corner queries (see Figure 1.1). We use path caching and obtain bounds of $O(\log_B n + t/B)$ I/Os for query time and $O(\log_B n)$ for amortized updates. The data structure occupies $O(\frac{n}{B} \log \log B)$ disk blocks of storage. We also present a data structure that implements priority search trees on disk to answer three-sided queries. For this data structure, we get bounds of $O(\log_B n + t/B)$ for query time and $O(\log_B n \log^2 B)$ for amortized updates. The data structure occupies $O(\frac{n}{B} \log B \log \log B)$ disk blocks of storage.

In addition to these data structures, path caching can also be applied to other main memory data structures like the segment tree and interval tree to obtain optimal query times at the expense of small space overheads. By doing this, we improve on the bounds of [6] for implementing segment trees in secondary memory.

1.5 Experimental Work

The actual use of a data structure in a database depends on many factors. Most of the data structures considered here have good worst-case bounds, but asymptotic bounds by themselves are not enough proof that a data structure will work well in practice. Many other factors like general efficiency, ease of implementation, interaction with the rest of the database system, etc. also play an important role. In this thesis, however, we restrict ourselves to experimentally verifying two of the important factors, namely general efficiency and ease of implementation. This can be thought of as a necessary prerequisite for an index structure to be successful.

We implemented the *class-divisions* algorithm presented in Section 3.2 for indexing classes in an OODB. This algorithm is especially suitable for implementation because it is simple and uses a data structure that is readily available in all database systems, namely B^+-trees. We performed extensive evaluation of this algorithm by comparing it with the method used commonly in OODBs today.

The experiments offered conclusive proof that we can successfully trade storage for good, guaranteed query times. For storage overheads that were generally under a factor of 3, the class-divisions algorithm offered an overall performance improvement of a factor of 8 or more. For sub-cases of the problem, this factor was often more than 50. These results were strong proof of the effectiveness of the algorithm, especially since the tests were conducted with "well behaved" data.

1.6 Summary of Contributions and Overview

The main contribution of this thesis can be summarized as follows: (1) We present I/O-efficient data structures, with provably good worst-case bounds, providing indexing support for new data models that are becoming more and more important. (2) We perform a rigorous

experimental evaluation of some of the new data structures to prove that they can perform extremely well in a realistic setting.

The rest of this thesis is organized as follows. Chapter 2 examines related research in the area of indexing. Section 3.1 explains the constraint data model in detail and shows that indexing constraints can be reduced to dynamic interval management in secondary storage, which in turn reduces to answering diagonal corner queries. In Chapter 4, we propose a new data structure, optimal for the static case, for this problem. Section 3.2 discusses the problem of indexing classes in more detail and presents a simple algorithm for this problem with good worst-case bounds. We improve on these bounds in Chapter 5 using the techniques developed for indexing constraints. Chapter 6 presents the idea of path caching and the results that we obtain using that idea. Chapter 7 presents the experimental evaluation of the work done on indexing in OODBs. We finish in Chapter 8 with the conclusions and open problems.

Chapter 2

Related Research

A large literature exists for in-core algorithms for two-dimensional range searching. The range tree [4] can be used to solve the problem in $O(n \log_2 n)$ space and static worst-case query time $O(\log_2 n + t)$. By using fractional cascading, we can achieve a worst-case dynamic query time $O(\log_2 n \log_2 \log_2 n + t)$ and update time $O(\log_2 n \log_2 \log_2 n)$ using the same space. We refer the reader to [9] for a detailed survey of the topic.

The ideal worst-case I/O bounds would involve making all the above logarithms have base B and compacting the output term to t/B; any other improvements would of course imply improvements to the in-core bounds. Unfortunately, the various in-core algorithms do not map to secondary storage in as smooth a fashion as balanced binary trees map to B^+-trees. For example, [30,38] examine mappings which maintain the logarithmic overheads and make the logarithms base B; however, their algorithms do not compact the t-sized output on t/B pages.

The practical need for general I/O support has led to the development of a large number of data structures for external k-dimensional searching. These data structures do not have good theoretical worst-case bounds, but have good average-case behavior for common

spatial database problems. Examples are the grid-file, various quad-trees, z-orders and other space filling curves, k-d-B-trees, hB-trees, cell-trees, and various R-trees. For these external data structures there has been a lot of experimentation but relatively little algorithmic analysis. Their average-case performance (e.g., some achieve the desirable static query I/O time of $O(\log_B n + t/B)$ on average inputs) is heuristic and usually validated through experimentation. Moreover, their worst-case performance is much worse than the optimal bounds achievable for dynamic external one-dimensional range searching using B^+-trees. We present here a brief survey of general purpose data structures to solve external two-dimensional range searching.

General purpose external k-dimensional range searching techniques can be broadly divided into two categories: those that *organize the embedding space* from which the input data is drawn and those that *organize the input data.*

We will first consider data structures that organize the embedding space with reference to our problem. Quad trees [33,34] were designed to organize two-dimensional data. They work by recursively subdividing each region into four equal pieces until the number of points in each region fits into a disk block. Because they do not adapt to the input data, they can have very bad worst-case times.

The grid file [28] was proposed as a data structure that treats all dimensions symmetrically, unlike many other data structures like the inverted file which distinguish between primary and secondary keys. The grid file works by dividing each dimension into ranges and maintaining a grid directory that provides a mapping between regions in the search space and disk blocks. The paper [28] does not provide analysis for worst-case query times. They do mention that range queries become very efficient when queries return "many" records.

If we assume a uniform two-dimensional grid of points (with a point on each location with integral x and y coordinates) as input, the grid file would produce regions which are of size $O(\sqrt{B}) \times O(\sqrt{B})$. The time to report answers to a two-sided range query would then be (in the worst-case) $O(t/\sqrt{B})$ where t is the number of points in the query result. This is higher than the optimal time of $O(t/B)$. In fact, it turns out that most data structures

in the literature fail to give optimal performance for this very simple example.

The second class of general purpose external k-dimensional range searching data structures that have been proposed for multi-attribute indexing are based on the principle of building a search structure based on the recursive decomposition of input data. Many of them are based on a B-tree-like organization. We will consider several of them with reference to our problems.

Two related data structures that have been proposed for multi-attribute indexing are the k-d-B-tree [32] and the hB-tree [24].

k-d-B-trees combine properties of balanced k-d-trees in a B-tree-like organization. In the two-dimensional case (these ideas generalize readily to higher dimensions), the k-d-B-tree works by subdividing space into rectangular regions. Such subdivisions are stored in the interior nodes of the k-d-B-tree. If a subdivision has more points than can be stored in a disk block, further subdivision occurs until each region in the lowest level of the tree contains no more points than can be stored in a disk block. Insertion and deletion algorithms for the k-d-B-tree are also outlined in [32]. This work does not offer any worst-case analysis for range search. As mentioned before, the k-d-B-tree works by subdividing space into rectangular regions. With a uniform grid of points as input, we would read $O(t/\sqrt{B})$ disk blocks to report t points on a straight line.

The hB-tree is based on the k-d-B-tree. Instead of organizing space as rectangles, they organize space into rectangles from which (possibly) other rectangles have been removed. This helps bring down the cost of managing insertions and deletions. This paper also does not provide a formal analysis of the cost of range searching. Although range searching in hB-trees is similar to range searching in B-trees, the crucial difference is that with B-trees, it is possible to totally order data. This is because B-trees index data along only one attribute. In fact, almost all implementations of B-trees recognize this and keep data only in their leaves and chain the leaves from left to right. (Such B-trees are called B^+-trees.) This makes range searching extremely simple and efficient. In order to find elements in a range $[a, b]$, we locate a in the B^+-tree and follows pointers to the right until the value b is

crossed. Such a total ordering is not possible for k-dimensional data. The problem that we had with k-d-B-trees with respect to the question at hand remains. Given a uniform grid, hB-trees still produce rectangles of size $O(\sqrt{B}) \times O(\sqrt{B})$ and range searching is inefficient. That is, we would read $O(t/\sqrt{B})$ disk blocks to report t points on a straight line.

Several data structures have been proposed in the literature to handle region data. These include the R-tree [17], the R^+-tree [35], the cell tree [16] and many others. These data structures are not directly applicable to point data. However, they can deal with one-dimensional range data and hence are relevant to our problem. All of them are based on the recursive decomposition of space using heuristics and cannot offer the worst-case guarantees in space and time that we seek.

An interesting idea based on the use of space-filling curves is proposed in [29]. This paper identifies a space-filling curve to order points in k-dimensional space. This curve has the desirable property that points that are close by in the input will, with high probability, be close by in the resulting ordering. This helps in making range searching efficient, because it is desirable to keep nearby points in the same disk block. This paper also does not offer any worst-case analysis for range searching. Specifically, this method will not have optimal reporting time with our standard case, i.e., the uniform grid of points.

Dynamic interval management has been examined extensively in the literature (see [9]). As mentioned before, the best in-core bounds have been achieved using the priority search tree of [27], yielding $O(n)$ space, dynamic query time $O(\log_2 n + t)$ and update time $O(\log_2 n)$, which are all optimal. Other data structures like the Interval Tree [12,13], and Segment Tree [3] can also solve the interval management problem optimally in-core, with respect to the query time. Among these, the priority search tree does the best because it solves the interval management problem in optimal time and space, and provides an optimal worst-case update time as well.

There have been several pieces of work done by researchers to implement these data structures in secondary memory. These works include [18], [5], [6]. [18] contains a claimed optimal solution for implementing static priority search trees in secondary memory. Unfor-

tunately, the [18] static solution has static query time $O(\log_2 n + t/B)$ instead of $O(\log_B n + t/B)$ and the claimed optimal solution is incorrect. None of the other approaches solve the problem of interval management in secondary memory in the optimal time of $O(\log_B n + t/B)$ either.

Recently, some progress has been made in the case where our goal is to handle a very large number of queries in a batch [15]. In this case, two-dimensional queries can be answered in $O((n/B + k/B)(\log_{M/B}(n/B)) + t/B)$ I/Os where k is the number of queries being processed and M is the amount of main memory available.

Chapter 3

The Problems and Initial Approaches

3.1 Indexing Constraints

To illustrate indexing constraints in CQLs consider the domain of rational numbers and a language whose syntax consists of the theory of rational order with constants + the relational calculus. (See [20] for details.)

In this context, a generalized k-tuple is a quantifier-free conjunction of constraints on k variables, which range over the domain (rational numbers). For example, in the relational database model $R(3,4)$ is a tuple of arity 2. It can be thought of as a single point in two-dimensional space and also as $R(x,y)$ with $x = 3$ and $y = 4$, where x, y range over some finite domain. In our framework, $R(x,y)$ with $(x = y \wedge x < 2)$ is a generalized tuple of arity 2, where x, y range over the rational numbers. Hence, a generalized tuple of arity k is a finite representation of a possibly infinite set of tuples of arity k.

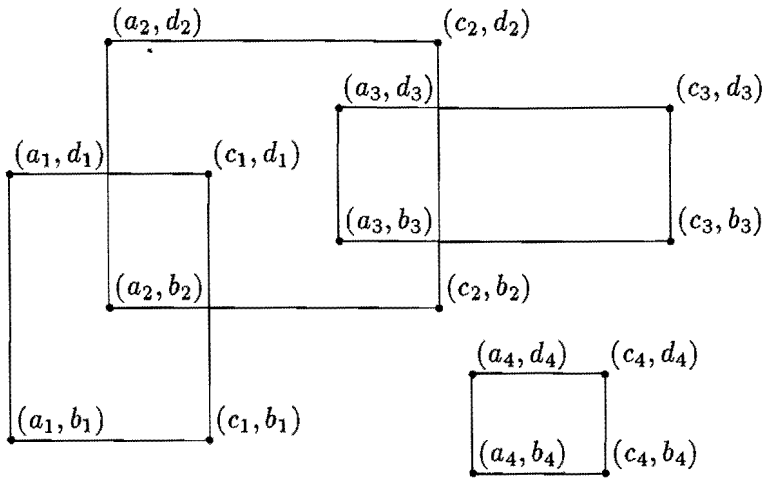


Figure 3.1: Rectangle intersection

A generalized relation of arity k is a finite set of generalized k-tuples, with each k-tuple over the same variables. It is a disjunction of conjunctions (i.e., in disjunctive normal form DNF) of constraints, which uses at most k variables ranging over domain D. A generalized database is a finite set of generalized relations. Each generalized relation of arity k is a quantifier-free DNF formula of the logical theory of constraints used. It contains at most k distinct variables and describes a possibly infinite set of arity k tuples (or points in k-dimensional space D^k).

The syntax of a CQL is the union of an existing database query language and a decidable logical theory (theory of rational order + the relational calculus here). The semantics of the CQL is based on that of the decidable logical theory, by interpreting database atoms as shorthands for formulas of the theory. For each input generalized database, the queries can be evaluated in closed form, bottom-up, and efficiently in the input size. Let us motivate this theory with an example.

Example 3.1.1 The database consists of a set of rectangles in the plane, and we want to compute all pairs of distinct intersecting rectangles. (See Figure 3.1.)

This query is expressible in a relational data model that has a $\leqslant$ interpreted predicate. One possibility is to store the data in a 5-ary relation named R. This relation will contain

tuples of the form (n, a, b, c, d), and such a tuple will mean that n is the name of the rectangle with corners at (a, b), (a, d), (c, b) and (c, d). We can express the intersection query as

$$\{(n_1, n_2) \mid n_1 \neq n_2 \wedge (\exists a_1, a_2, b_1, b_2, c_1, c_2, d_1, d_2)(R(n_1, a_1, b_1, c_1, d_1) \wedge R(n_2, a_2, b_2, c_2, d_2) \\ \wedge(\exists x, y \in \{a_1, a_2, b_1, b_2, c_1, c_2, d_1, d_2\}) \\ (a_1 \leqslant x \leqslant c_1 \wedge b_1 \leqslant y \leqslant d_1 \wedge a_2 \leqslant x \leqslant c_2 \wedge b_2 \leqslant y \leqslant d_2))\}$$

To see that this query expresses rectangle intersection note the following: the two rectangles n_1 and n_2 share a point if and only if they share a point whose coordinates belong to the set $\{a_1, a_2, b_1, b_2, c_1, c_2, d_1, d_2\}$. This can be shown by exhaustively examining all possible intersecting configurations. Thus, we can eliminate the $(\exists x, y)$ quantification altogether and replace it by a boolean combination of $\leqslant$ atomic formulas, involving the various cases of intersecting rectangles.

The above query program is particular to rectangles and does not work for triangles or for interiors of rectangles. Recall that, in the relational data model quantification is over constants that appear in the database. By contrast, if we use generalized relations the query can be expressed very simply (without case analysis) and applies to more general shapes.

Let $R(z, x, y)$ be a ternary relation. We interpret $R(z, x, y)$ to mean that (x, y) is a point in the rectangle with name z. The rectangle that was stored above by (n, a, b, c, d), would now be stored as the generalized tuple $(z = n) \wedge (a \leqslant x \leqslant c) \wedge (b \leqslant y \leqslant d)$. The set of all intersecting rectangles can now be expressed as

$$\{(n_1, n_2) | n_1 \neq n_2 \wedge (\exists x, y)(R(n_1, x, y) \wedge R(n_2, x, y)\}$$

The simplicity of this program is due to the ability in CQL to describe and name point-sets using constraints. The same program can be used for intersecting triangles. This simplicity of expression can be combined with efficient evaluation techniques, even if quantification is over the infinite domain of rationals. For more examples and details, please see [20]. □

The CQL model for rational order + relational calculus has low data complexity, be-

cause every fixed query is evaluable in LOGSPACE. That alone is not enough to make it a suitable model for implementation. This situation is similar to that in the relational model, where the language framework does have low data complexity, but does not account for searches that are logarithmic or faster in the sizes of input relations. Without the ability to perform such searches relational databases would have been impractical. Very efficient use of secondary storage is an additional requirement, beyond low data complexity, whose satisfaction greatly contributes to any database technology.

In the above example the domain of database attribute x is infinite. How can we index on it? For CQLs we can define *indexing constraints* as the problem of *external dynamic one-dimensional range searching on generalized database attribute* x using the following operations: (i) Find a generalized database that represents all tuples of the input generalized database such that their x attribute satisfies $a_1 \leqslant x \leqslant a_2$. (ii) Insert or delete a given generalized tuple.

If $(a_1 \leqslant x \leqslant a_2)$ is a constraint of our CQL then there is a trivial, but inefficient, solution to the problem of one-dimensional searching on generalized database attribute x. We can add the constraint $(a_1 \leqslant x \leqslant a_2)$ to every generalized tuple (i.e., conjunction of constraints) and naively insert or delete generalized tuples in a table. This involves a linear scan of the generalized relation and introduces a lot of redundancy in the representation. In many cases, the projection of any generalized tuple on x is one interval $(a \leqslant x \leqslant a')$. This is true for Example 3.1.1, for relational calculus with linear inequalities over the reals, and in general when a generalized tuple represents a convex set. (We call such CQLs convex CQLs.) Under such natural assumptions, there is a better solution for one-dimensional searching on generalized database attribute x.

- A *generalized one-dimensional index* is a set of intervals, where each interval is associated with a generalized tuple. Each interval $(a \leqslant x \leqslant a')$ in the index is the projection on x of its associated generalized tuple. The two endpoint a, a' representation of an interval is a fixed length *generalized key.*

- Finding a generalized database, that represents all tuples of the input generalized

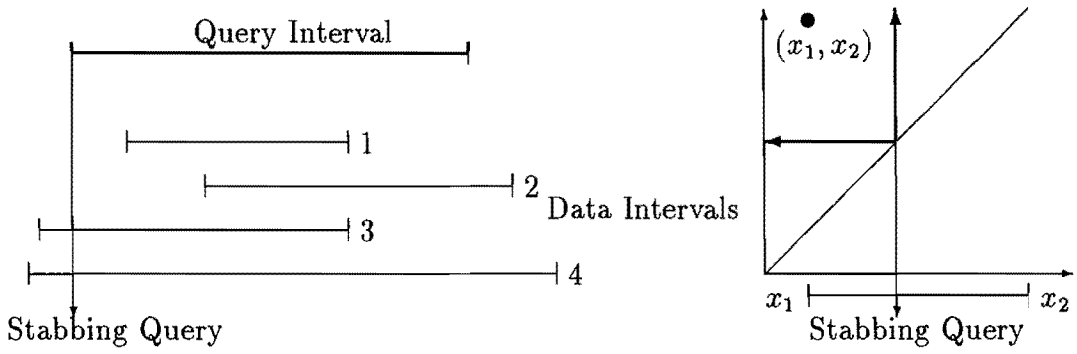


Figure 3.2: Reducing interval intersection to stabbing queries and stabbing queries to diagonal corner queries

database such that their x attribute satisfies $(a_1 \leqslant x \leqslant a_2)$, can be performed by adding constraint $(a_1 \leqslant x \leqslant a_2)$ to only those generalized tuples whose generalized keys have a non-empty intersection with it.

- Inserting or deleting a given generalized tuple is performed by computing its projection and inserting or deleting an interval from the set of intervals.

By the above discussion, the use of generalized one-dimensional indexes reduces redundancy of representation and transforms one-dimensional searching on generalized database attribute x into the problem of external dynamic interval management. The framework described in this section is from [20].

In this thesis, we examine solutions for the problem of external dynamic interval management that have good I/O performance. Keeping in mind that a diagonal corner query is a two-sided range query whose corner must lie on the line $x = y$ and whose query region is the quarter plane above and to the left of the corner (see Figure 1.1), we now can show the following proposition:

Proposition 3.1.2 Indexing constraints for convex CQLs reduces to external dynamic interval management which reduces to external dynamic two-dimensional range searching with diagonal corner queries and updates.

Proof: As remarked before, indexing constraints means solving the problem of interval intersection in secondary memory. Given a set of input intervals, we would like to find all intervals that intersect a query interval. Intervals that intersect a query interval $[x_1, x_2]$ can be divided into four categories as shown in Figure 3.2. Types 1 and 2 can be reported by sorting all the intervals on the basis of their first endpoint and reporting those intervals whose first endpoint lies between x_1 and x_2. This can be done efficiently using a B^+-tree. Types 3 and 4 can be reported by performing what is called a *stabbing query* at x_1. (A stabbing query at x_1 on a set of intervals returns those intervals that contain the point x_1. See Figure 3.2.) It is also clear that no interval gets reported twice by this process.

Therefore, we will be able to index constraints if we can answer stabbing queries efficiently. An interval contains a point q if and only if its first endpoint y_1 is less than or equal to q and its second endpoint y_2 is greater than or equal to q. Let us map an interval $[y_1, y_2]$ to the point (y_1, y_2) in two-dimensional space. Clearly, the stabbing query now translates into a two-sided query. That is, an interval $[y_1, y_2]$ belongs to a stabbing query at q if and only if the corresponding point (y_1, y_2) is inside the box generated by the lines $x = 0$, $x = q$, $y = q$, and $y = \infty$. Since the second endpoint of an interval is always greater than or equal to the first endpoint, all points that we generate are above the line $x = y$. One of the corners of any two-sided query (corresponding to a stabbing query) is anchored on this line as well. (See Figure 3.2.) The proposition follows. □

3.2 Indexing Classes

To illustrate the problem of indexing classes, consider an object-oriented database. The *objects* in the database are classified in a *forest class hierarchy.* Each object is in exactly one of the classes of this hierarchy. This partitions the set of objects and the block of the partition corresponding to a class C is called C's *extent.* The union of the extent of a class C with all the extents of all its descendants in this hierarchy is called the *full extent* of C. Let us illustrate these ideas with an example.

Example 3.2.1 Consider a database that contains information about people such as names and incomes. Let the people objects be organized in a class hierarchy which is a tree with root Person, two children of Person called Professor, Student, and a child of Professor called Assistant-Professor. (See Figure 3.4.) We can read this as follows: Assistant-Professor *isa* Professor, Professor *isa* Person, Student *isa* Person. People get partitioned in these classes. For example, the full extent of Person is the set of all people, whereas the extent of Person is the set of people who are not in the Professor, Assistant-Professor, and Student extents. □

Indexing classes means being able to perform *external dynamic one-dimensional range searching on some attribute of the objects, but for the full extent of each class in the hierarchy.*

Example 3.2.2 Consider the class hierarchy in Example 3.2.1. Indexing classes for this hierarchy means being able to find all people in (the full extent of) class Professor with income between \$50K and \$60K, or to find all people in (the full extent of) class Person with income between \$100K and \$200K, or to insert a new person with income \$10K in the Student class. □

Let c be the number of classes, n the number of objects, and B the page size. We use the term *index a collection* when we build a B^+-tree on a collection of objects. (This B^+-tree will be built over some attribute which will always be clear from context. In Example 3.2.2, the attribute was the salary attribute.) One way of indexing classes is to create a single B^+-tree for all objects (i.e., index the collection of all objects) and answer a query by looking at this B^+-tree and filtering out the objects in the class of interest. This solution cannot compact a t-sized output into t/B pages because the algorithm has no control over how the objects of interest are interspersed with other objects. Another way is to keep a B^+-tree per class (i.e., index the full extent of each class), but this uses $O((n/B)c)$ pages, has query I/O time $O(\log_B n + t/B)$ and update I/O time $O(c \log_B n)$.

The indexing classes problem has the following special structure: (1) The class hierarchy

is a forest and thus it can be mapped in one dimension where subtrees correspond to intervals. (2) The class hierarchy is static, unlike the objects in it which are dynamic.

Based on this structure we show that indexing classes is a special case of external dynamic two-dimensional range searching on some attribute of the objects. We then use the idea of the two-dimensional range tree (see [9]), with classes as the primary dimension and the object attribute as a secondary dimension to devise an efficient storage and query strategy. These ideas are formalized in the proposition and theorem to follow.

Proposition 3.2.3 Indexing classes reduces to external dynamic two-dimensional range searching with one dimension being static.

Proof: We write a simple algorithm which attaches a new attribute called "class" to every object. This attribute has a value corresponding to the class to which the object belongs. Further, we associate a range with each class such that it includes all the class attribute values of each one of its subclasses. (The class attribute and ranges are for the purposes of this reduction only. In an actual implementation, these are computed once and stored separately.) We start out by associating the half-open range $[0, 1)$ with the root of the class hierarchy. We then make a call to the procedure *label-class* shown in Figure 3.3 with the root and the range as parameters. (If the hierarchy is a forest of k trees, we simply divide the range $[0, 1)$ into k equal parts, associate every root with a distinct range and call *label-class* once for each root.)

At the end of the procedure, every class is associated with a range and every object has a "class" value associated with it. If we apply *label-class* to the class hierarchy in Example 3.2.1, the root Person class is associated with the range $[0, 1)$ and all the objects in its extent have their class attribute set to 0. Similarly, the Student, Professor and Asst. Prof. classes are associated with ranges $[\frac{1}{3}, \frac{2}{3})$, $[\frac{2}{3}, 1)$, and $[\frac{5}{6}, 1)$ respectively. The objects in the Student, Professor and Asst. Prof. classes have their class attributes set to $\frac{1}{3}$, $\frac{2}{3}$ and $\frac{5}{6}$ respectively.

procedure *label-class* (*node*, $[a, b)$);
Associate $[a, b)$ with *node*
Let a be the value of attribute "class" for every object in class *node*
Let S = (*The number of children of node*) +1
if *node* has no children, terminate
Divide $[a, b)$ into S equal parts of size K
Recursively call *label-class* for each child with ranges
$[a + K, a + 2K)$, $[a + 2K, a + 3K)$, etc.

Figure 3.3: The procedure *label-class* used in the proof of Proposition 3.2.3.

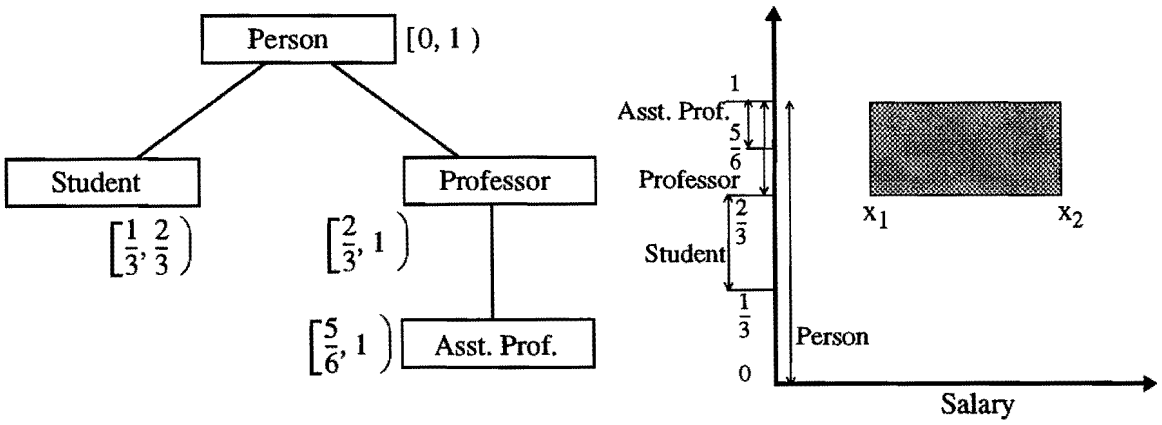


Figure 3.4: Using *label-class* to reduce indexing classes to two-dimensional range search

It is easy to see how querying some class over some particular attribute corresponds to two-dimensional range searching. The first dimension of this search is the class attribute and the second dimension is the attribute over which the search is specified. The range in the class dimension is the range that we associate with the query class in our *label-class* algorithm. Since we assume that the class hierarchy is static, the proposition follows. See Figure 3.4 for an example of the results of applying *label-class* to a class hierarchy. □

Theorem 3.2.4 *Indexing classes can be solved in dynamic query I/O time* $O(\log_2 c \log_B n + t/B)$ *and update I/O time* $O(\log_2 c \log_B n)$, *using* $O((n/B)\log_2 c)$ *pages. Here,* n *is the number of objects in the input database,* c *the size of the class hierarchy,* t *the number of objects in the output of a query, and* B *the size of a disk block.*

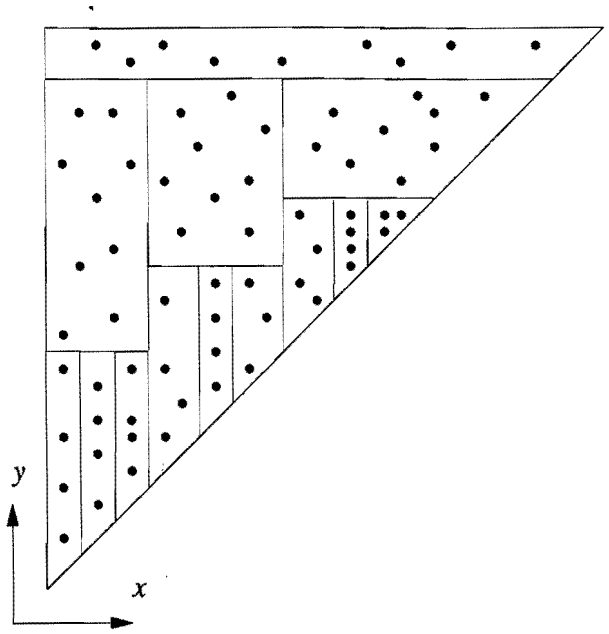


Figure 4.1: A metablock tree for $B = 3$ and $n = 70$. All data points lie above the line $y = x$. Each region represents a metablock. The root is at the top. Note that each non-leaf metablock contains $B^2 = 9$ data points.

4.1 An I/O Optimal Static Data Structure for Diagonal Corner Queries

At the outermost level, a metablock tree, whether static or dynamic, is a B-ary tree of *metablocks*, each of which represents B^2 data points. The root represents the B^2 data points with the B^2 largest y values. The remaining $n - B^2$ data points are divided into B groups of $(n - B^2)/B$ data points each based on their x coordinates. The first group contains the $(n - B^2)/B$ data points with the smallest x values, the second contains those with the next $(n - B^2)/B$ smallest x values, and so on. A recursive tree of the exact same type is constructed for each such group of data points. This process continues until a group has at most B^2 data points and can fit into a single metablock. This is illustrated in Figure 4.1.

Now let us consider how we can store a set of $k \leqslant B^2$ data points belonging to a metablock in blocks of size B. One very simple scheme is to put the data points into

(a) Vertically oriented

(b) Horizontally oriented

Figure 4.2: Vertically and horizontally oriented blockings of data points. Each thin rectangle represents a block.

horizontally oriented blocks by putting the B data points with the largest y values into the first block, the B data points with the next largest y values into the next block, and so on. Similarly, we can put the data points into *vertically oriented* blocks by discriminating on the x coordinates. These techniques are illustrated in Figure 4.2. Each metablock in our tree is divided into both horizontally and vertically oriented blocks. This means that each data point is represented more than once, but the overall size of our data structure remains $O(n/B)$.

In addition to the horizontally and vertically oriented blocks, each metablock contains pointers to each of its B children, as well as a location of each child's bounding box. Finally, each metablock M contains pointers to B blocks that represent, in horizontal orientation, the set $TS(M)$. $TS(M)$ is the set obtained by examining the set of data points stored in the left siblings of M and taking the B^2 such points with the largest y values. This is illustrated in Figure 4.3. Note that each metablock already requires $O(B)$ blocks of storage space, so storing $TS(M)$ for each metablock does nothing to the asymptotic space usage of the metablock tree.

The final bit of organization left is used only for those metablocks that can possibly contain the corner of a query. These are the leaf metablocks, the root metablocks, and all metablocks that lie along the path from the root to the rightmost leaf. (See the metablocks on the diagonal in Figure 4.1.) These blocks will be organized as prescribed by the following

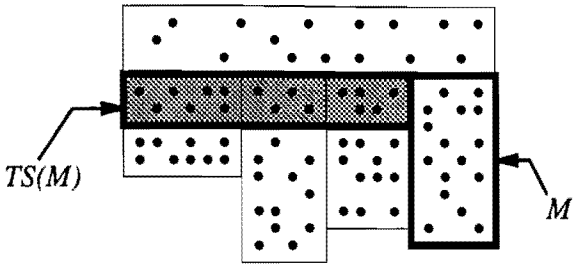


Figure 4.3: A metablock M and the set $TS(M)$. Note that $TS(M)$ spans all of M's left siblings in the metablock tree. Though it is not explicitly shown here, $TS(M)$ will be represented as a set of B horizontally oriented blocks.

lemma:

Lemma 4.1.1 *A set S of $k \leqslant B^2$ data points can be represented using $O(k/B)$ blocks of size B so that a diagonal corner query on S can be answered using at most $2t/B + 4$ I/O operations where t is the number of data points in S that lie within the query region.*

Proof: Initially, we divide S into a vertically oriented blocking of k/B blocks. Let C be the set of points at which right boundaries of the regions corresponding to the vertically oriented blocks intersect the line $y = x$. Now we choose a subset $C^* \subseteq C$ of these points and use one or more blocks to explicitly represent the answer to each query that happens to have a corner $c \in C^*$. This is illustrated in Figure 4.4.

In order to decide which elements of C will become elements of C^*, we use an iterative process. The first element of C^* is chosen to be at the intersection of the left boundary of the rightmost block in the vertically oriented blocking. We will call this point c_1^*. To decide which other elements of C should be elements of C^*, we proceed along the line $y = x$ from the upper right (large x and y) to the lower left (small x and y), considering each element of C we encounter in turn. Let c_j^* be the element of C most recently added to C^*; initially this is c_1^*. We now move down the line $y = x$, considering each $c_i \in C$ until we find one to add to C^*. In considering $c_i \in C$, we define the sets Ω_i, Δ_i^{-1} , Δ_i^{-2} and Δ_i^{+} to be subsets of S as shown in Figure 4.5. Let $\Delta_i^{-} = \Delta_i^{-1} \cup \Delta_i^{-2}$. Let $S_j^* = \Omega_i \cup \Delta_i^{-1}$ be the answer to a query whose corner is c_j^*. This was the last set of points that was explicitly blocked. Let

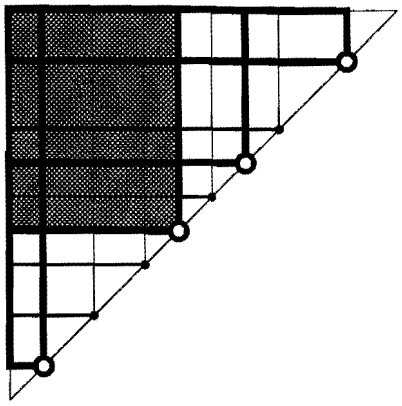


Figure 4.4: The sets C and C^* used in the proof of Lemma 4.1.1. The marked points lying along the diagonal line $y = x$ are the points in the set C. Those that are small and dark are points in $C \setminus C^*$. The larger open points are in the set C^*. The dark lines represent the boundaries of queries whose corners are at points $c \in C^*$. One such query is shaded to demonstrate what they look like.

$S_i = \Omega_i \cup \Delta_i^+$ be the answer to a query whose corner is c_i. We decide to add c_i to C^*, and thus explicitly store S_i, if and only if

$$|\Delta_i^-| + |\Delta_i^+| > |S_i|.$$

Intuitively, we do not add c_i to C^* when we can efficiently amortize the cost of a query cornered at c_i over a number of blocks that have already been constructed.

Having constructed C^*, we now explicitly block the set S_i^* answering a diagonal corner query for each element $c_i^* \in C^*$. Clearly the space used for each such set is $\lceil |S_i^*|/B \rceil$ blocks. An obvious concern is that by explicitly blocking these sets, we may already be using more space than the lemma we are trying to prove allows. This, however, is not the case. We can prove this by amortization. Each time we add a c_i to C^*, we will charge $|S_i|$ credits to the set $\Delta_i^- \cup \Delta_i^+$. The charge is divided equally among the elements of the set being charged. Once we add c_i to C^*, no element of Δ_i^- can ever be in another Δ_i^- for a larger value of i as the iteration continues. The same holds for elements of Δ_i^+, which can never appear as elements of Δ_i^+ again for larger i. Thus no element of S can be a part of a set that is charged for the insertion of a c_i into C^* more than twice. Since the size of the set being

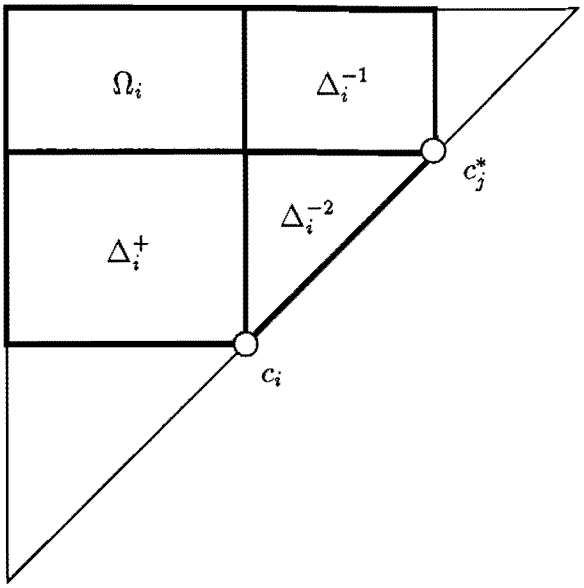
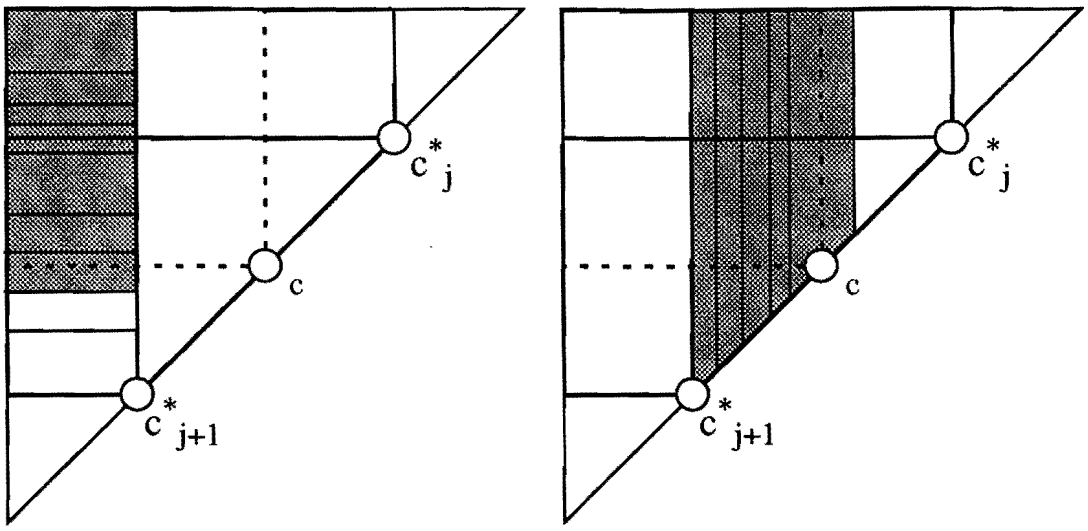


Figure 4.5: The sets Ω_i, Δ_i^{-1} , Δ_i^{-2} and Δ_i^+ as constructed in the proof of Lemma 4.1.1. c_j^* is the last point that was added to C^* and c_i is the point being considered for inclusion. The sets consist of subsets of S falling within the labeled regions.

charged is at least the size of the set being inserted, the total cost charged to any point is at most 2. Thus the total size of all the sets in C^* is at most $2k$, which we can clearly represent within $O(k/B)$ blocks. The only possible concern is that almost a full block can be wasted for each set in C^* due to roundoff. This is not a problem however, since $|C^*| \leqslant |C| = k/B$, so the number of blocks wasted is at most of the same order as the number used.

Now that we have a bound on the space used by our data structure, we have only to show that it can be used to answer queries in $2t/B + 4$ I/Os. To answer a query whose corner is at some $c_i^* \in C^*$, we simply read the blocks that explicitly store the answer to the query. This takes $\lceil t/B \rceil \leqslant t/B + 1$ I/Os, which is clearly within the required bounds.

The more complicated case is when the query point c lies between two consecutive elements c_j^* and c_{j+1}^* in C^*. Let T be the subset of S that is the answer to the query whose corner is c. We find T in two stages. In the first stage, we read blocks from S_{j+1}^*, which we assume is horizontally blocked, starting at the top and continuing until we reach the bottom of the query. This is illustrated in Figure 4.6(a). At most one block is wasted in this process.



(a) The first phase of the algorithm to answer a query whose corner is c. The shaded blocks correspond to blocks of the explicitly blocked set S^*_{j+1} that are read in this phase.

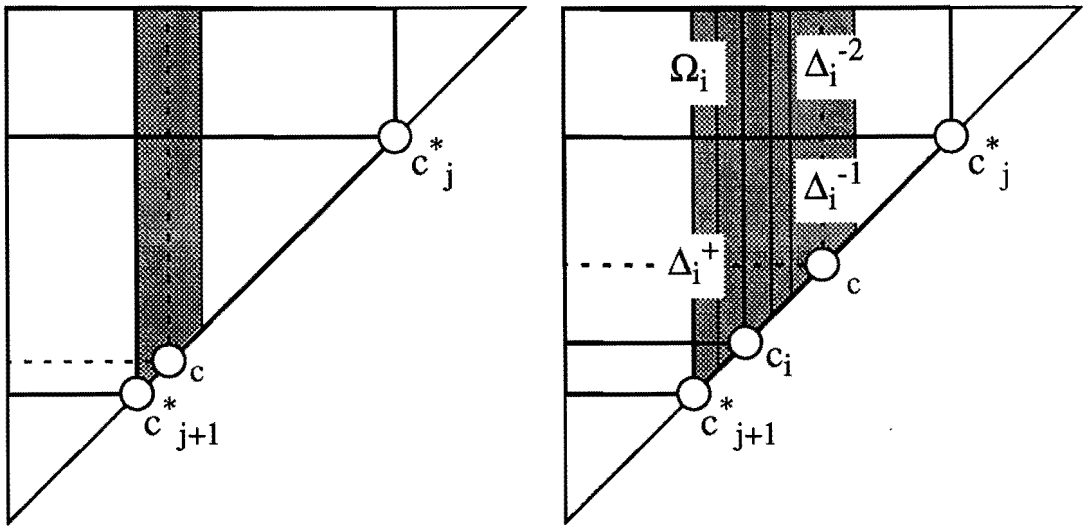
(b) The second phase of the algorithm to answer a query. In this phase the shaded vertical blocks are examined.

Figure 4.6: The two phases of the algorithm to answer a query whose corner is c as used in the proof of Lemma 4.1.1. c^*_j and c^*_{j+1} are consecutive elements of the set C^*, an thus answers to queries having them as corners are explicitly blocked.

In the second stage, we return to our original vertical blocking of S (into k/B blocks), and read blocks from left to right, starting with the one directly to the right of c^*_{j+1} and continuing until we reach the block containing c. This is illustrated in Figure 4.6(b). To show that we are still within the I/O bounds we are trying to prove, we consider two cases. In the first case, there is no $c_i \in C$ between c and c^*_{j+1}. This is illustrated in Figure 4.7(a). In this case, we only have to read one block from the vertical blocking, namely the one immediately to the right of c^*_{j+1}, and the proof is done.

In the second case, there is at least one $c_i \in C$ between c and c^*_{j+1}. Let us consider the leftmost such c_i. Since c^*_j and c^*_{j+1} are consecutive elements of C^*, it must be the case that c_i was not chosen in the iterative process that constructed C^*. This means that

$$|\Delta_i^-| + |\Delta_i^+| \leqslant |S_i|,$$



(a) In the first case c is in the vertical block immediately to the right of c^*_{j+1}.

(b) In the second case there is at least one element of C between c and c^*_{j+1}. c_i is the leftmost such element.

Figure 4.7: Two cases of the second phase of the algorithm (Figure 4.6(b)) used in the proof of Lemma 4.1.1.

otherwise c_i would have been added to C^*. Subtracting $|\Delta_i^+|$ from both sides, we get

$$|\Delta_i^-| \leqslant |\Omega_i|,$$

since Δ_i^+ and Ω_i are disjoint but their union is S_i. Referring to Figure 4.7(b), we see that Ω_i is a subset of T, the answer to the query whose corner is c. Thus

$$|\Delta_i^-| \leqslant |\Omega_i| \leqslant |T|.$$

We also see that except for the leftmost block, all the vertical blocks examined in the second phase in Figure 4.6(b) are fully contained in Δ_i^-. This means that in the worst case we will have to examine all of Δ_i^- plus one additional block. But since $|\Delta_i^-| \leqslant |T|$, the number of blocks examined is at most equal to the number of blocks needed to represent the output of our query. Even if all these blocks are wasted because the points they contain lie in the region below c, no more than $\lceil t/B \rceil$ blocks are wasted. If we add these to the blocks used

in the first stage, and the vertical block just to the right of c^*_{j+1}, the total number of blocks used is no more than $2t/B + 3$.

The final step of the proof is to show how we can determine where to begin looking for a query given a query point c. Since $k \leqslant B^2$, the size of C is at most B. We can thus use a single block to store an index with pointers to the appropriate locations to begin the two stages of the search for any query value falling between two consecutive elements of C. This gives us our overall result of $2t/B + 4$ I/Os.

The only cases we have not considered occur when c does not fall between two elements of C^*, but rather completely to either the left or right of them. These special cases can be handled by minor variations of the arguments given above. □

Now that we know how to structure the corner blocks of a metablock tree so that the portion of a query that falls within that metablock can be reported efficiently, we need only show that the portions of the query residing in the rest of the metablocks can also be reported efficiently. We do this with the following theorem:

Theorem 4.1.2 *If a set of n data points (x, y) in the planar region $y \geqslant x$ and $y > 0$ is organized into a metablock tree with blocks of size B, then a diagonal corner query with t data points in its query region can be performed in $O(\log_B n + t/B)$ I/O operations. The size of the data structure is $O(n/B)$ blocks of size B each.*

Proof: First we consider the space used by the tree. In Lemma 4.1.1 we showed that the number of blocks used by each corner metablock containing k points is $O(k/B)$. All other metablocks in the tree must be internal, and thus contain B^2 points. Each such metablock occupies $O(B)$ blocks as was shown in the discussion of their construction. This includes the associated TS data structures. In addition, we will use a constant number of disk blocks per metablock to store control information for that metablock. This will include split values and pointers to its children, boundary values and pointers to the horizontal organization, etc. This control information uses $O(n/B^2)$ disk blocks since there are only

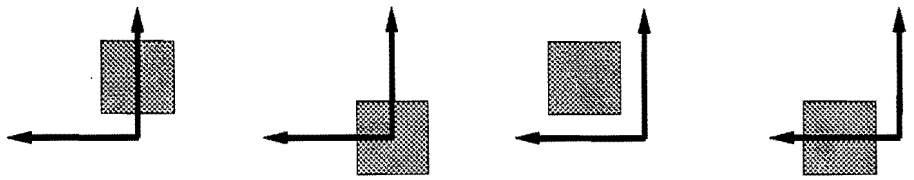
(1) **procedure** *diagonal-query* (query q, node M);
(2) if M contains the corner of query q then
(3) use the corner structure for M to answer the query; return;
(4) else /* M does not contain the corner of query q */
(5) use M's vertically oriented blocks to report points of M that lie inside q
(6) let M_c be the child of M containing the vertical side of the query
(7) if M_c has siblings to its left that fall inside the query, use the bottom boundary of $TS(M_c)$ to determine if $TS(M_c)$ falls completely inside the query.
(8) If $TS(M_c)$ does not fall completely inside the query, report the points in $TS(M_c)$ that lie inside the query.
(9) If $TS(M_c)$ falls completely inside the query, examine the siblings one by one using their horizontally oriented blocks.
(10) If any sibling is completely contained inside the query, look at its children using their (the children's) horizontally oriented organizations and continue downwards until the bottom boundary of the query is reached for every metablock so examined.
(11) invoke *diagonal-query*(q, M_c)

Figure 4.8: Procedure *diagonal-query* to answer diagonal corner queries using a metablock tree

$O(n/B^2)$ metablocks. Thus a total of $O(n/B)$ blocks are used.

Queries are answered using the procedure *diagonal-query* in Figure 4.8 (by invoking *diagonal-query (query q, root of metablock tree)*). We associate every metablock with the minimum bounding rectangle of the points inside it. Call this the metablock's *region*. Also, a diagonal corner query q is completely specified by the point at which it is anchored on the line $x = y$. We will use phrases like "if metablock M contains the corner of query q," etc. with the understanding that these are really short procedures that can be implemented easily by considering intersections of the appropriate regions.

To prove the bound on the query time, we consider the fact that once we know the query region, each block that has a non-empty intersection with it falls into one of four categories based on how it interacts with the boundary of the query. These four types are illustrated in Figure 4.9. To complete the proof we simply look at the contributions of each type of metablock to the total number of I/Os required.



Type I Type II Type III Type IV

Figure 4.9: The four types of metablocks. The shaded rectangles represent metablock boundaries and the thick lines are the boundaries of queries. The four types appear in processing diagonal corner queries as described in Section 4.1 and are used in the proof of Theorem 4.1.2

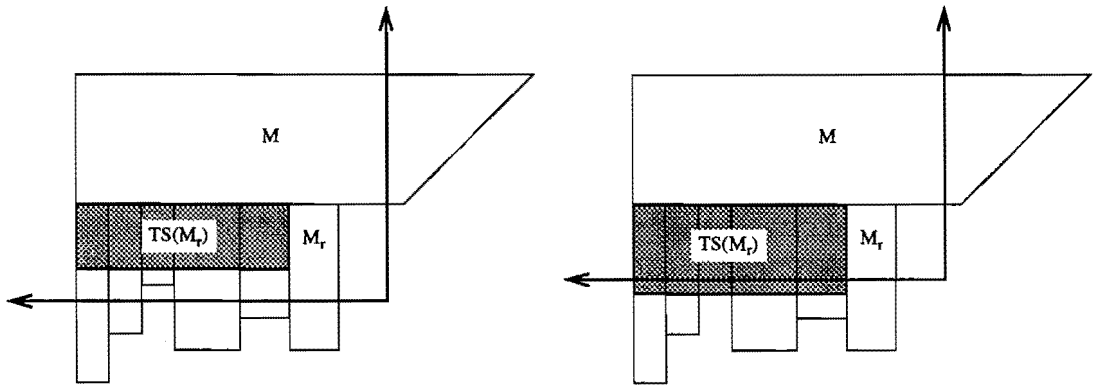
- There are at most $O(\log_{B^2} n) = O(\log_B n)$ Type I nodes, each of which can be queried, using its vertically oriented blocks, so as to visit at most one block that is not completely full. These potentially wasted blocks are accounted for in the $O(\log_B n)$ term.

- Only one Type II node can exist. Let t_c be the number of data points stored in the metablock at this node that are within the query region. By Lemma 4.1.1 the t_c points can be found using only $O(t_c/B)$ I/O operations. This gets absorbed into the $O(t/B)$ term.

- A Type III metablock returns B^2 data points and uses $O(B)$ I/O operations. These are accounted for in the $O(t/B)$ term.

- The set of all Type IV children of a Type III node can be queried, using their horizontally oriented blocks, so as to examine at most $O(B)$ blocks that are not entirely full (one per child). Since we used $O(B)$ I/O operations for the output from the Type III block, the extra Type IV I/O operations can be absorbed into the Type III I/O and added to the $O(t/B)$ term.

- The last type of block that has to be considered is a Type IV child of a Type I node. Up to B Type IV nodes can be children of a single Type I node M. Let M_r be the rightmost Type IV child of M. We can determine whether or not to examine the Type IV siblings of M_r by first examining $TS(M_r)$. If the bottom boundary of $TS(M_r)$ is below the bottom boundary of the query, we load the blocks that make up $TS(M_r)$ one at a time from top to bottom until we cross the bottom boundary of



(a) In this case the entire $TS(M_r)$ structure lies above the bottom of the query region, implying that we can individually examine all Type IV right siblings of M_r.

(b) In this case the $TS(M_r)$ structure intersects the bottom boundary of the query region, thus all points in the query that appear in Type IV siblings of M_r will be located when $TS(M_r)$ is examined.

Figure 4.10: Use of the TS structure to search Type IV metablocks as used in the proof of Theorem 4.1.2.

the query. If the bottom boundary of $TS(M_r)$ is above the bottom boundary of the query, then we know that the siblings contain at least B^2 points that are inside the query. Thus we can afford to examine each of M_r's Type IV siblings individually. This process may result in one block of overshoot for each such sibling, but we will get at least B^2 points overall, so we can afford this. This case is illustrated in Figure 4.10(a). If we hit the boundary of the query first, we can simply report all the points we saw as part of the output and we have no need to examine any of M_r's Type IV siblings at all. This case is illustrated in Figure 4.10(b). In both cases, all the blocks examined can be charged to the $O(t/B)$ term.

□

Theorem 4.1.2 gives us an upper bound for handling diagonal corner queries. The following proposition provides a matching lower bound, which proves that the metablock tree technique is optimal.

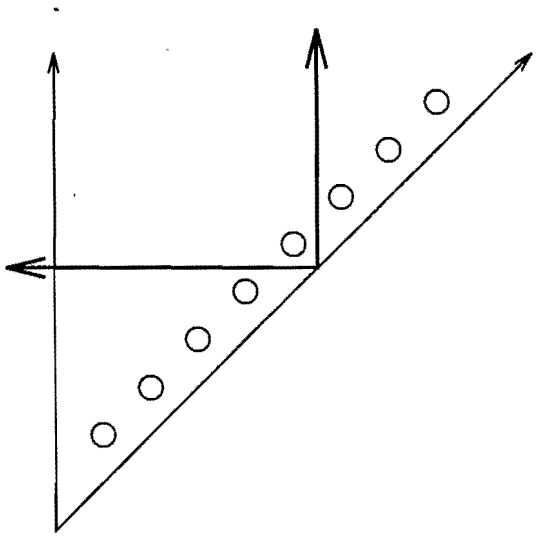


Figure 4.11: The set of points S and a query whose corner is an element of the set of Q as described in the lower bound proof of Theorem 4.1.3.

Proposition 4.1.3 Any method that performs diagonal corner queries on sets of n data points (x, y) in the planar region $y \geqslant x$ and $y > 0$ must use $\Omega(\log_B n + t/B)$ I/O operations, where t is the number of items in the answer to the query. Furthermore, the data structure it uses must occupy $\Omega(n/B)$ blocks.

Proof: Consider the set of points $S = \{(x, x+1) : x \in \mathbb{Z}_n^+\}$ and the set of n queries whose corners are the elements of the set $Q = \{(x + \frac{1}{2}, x + \frac{1}{2}) : x \in \mathbb{Z}_n^+\}$. Each of these queries contains exactly one point in the set S. This is illustrated in Figure 4.11. Since there are n distinct subsets of S that can be reported as answers to queries, we must have some means of deciding which, if any, answer our question. Each time we read a block, we can make a decision between at most $O(B)$ alternatives. This means that if we view our algorithm as a decision tree with a block read at each node it must have depth $\Omega(\log_B n)$ in order to have n leaves. This gives us the first term of our lower bound.

The second term of the lower bound comes from the fact that in general we must report the answer to any query of size $t \leqslant n$ and therefore at least t/B block reads will be required.

The space utilization follows since every element of S may appear in the answer to some query, and thus it must be represented somewhere in the data structure. □

4.2 Dynamization of Insertions

The data structure described in Section 4.1 can be made semi-dynamic in the sense that it will support the efficient insertion of points. Because of the complexity of the internal organization of a metablock, it will not be possible for us to reorganize the data structure immediately after the insertion of a point. Our strategy will be to defer reorganization until sufficient inserts have occurred for us to be able to pay for the reorganizations that we perform. Thus, the bounds we get are amortized.

The general strategy will be to collect inserts that go into a metablock in an *update block* that is associated with that metablock. When that update block fills up, we will perform certain reorganizations on the associated metablock. In order to keep the size of a metablock itself under limits, we will peform certain other reorganizations when the size of a metablock reaches $2B^2$ from the initial size of B^2. This will eventually cause the branching factors of the nodes in the metablock tree to go up. We will let branching factors increase from the initial value of B to $2B$ after which will perform certain other reorganizations that will restore the branching factor back to B. The precise details are below.

To start with, we first enumerate the different positions at which a point can be stored in a metablock tree:

a. in the vertically oriented organization of a metablock,

b. in the horizontally oriented organization of a metablock,

c. in the corner structure of a metablock (if the metablock intersects the diagonal line), and,

d. in the TS structures of the right siblings of the metablock.

It is relatively easy to handle the reorganization of the (a), (b), and (c) (where it exists) components of a metablock when insertions are allowed. Note however, that immediate reorganization after the insertion of a point is still not possible because it takes $O(B)$ I/Os

to reorganize components (a), (b), and (b) of a metablock. Our strategy will be to associate an additional update block in the control information of a metablock. When this update block fills up, we have had B inserts into the metablock. At this point, we reorganize the vertically and horizontally oriented components of the metablock, and rebuild the corner structure (if it exists for the metablock) optimally as described in Lemma 4.1.1. This costs $O(B)$ I/Os. This implies that we spend only $O(1)$ I/Os amortized per point for this reorganization. Let us call this reorganization of a metablock M a *level I* reorganization of M. Note that we will peform a level I reorganization once in every B inserts. When the size of a metablock reaches $2B^2$, we will perform other reorganizations that are outlined below.

Updating the TS structures of the siblings of a metablock when points are inserted into it is much more difficult because a point, potentially, can belong in the TS structures of $B-1$ siblings. Since rebuilding these TS structures (of total size $O(B^2)$ blocks) takes $O(B^2)$ I/Os, we cannot afford to rebuild the TS structures even once in B inserts.

In order to circumvent this problem, we use the following crucial observation: *Consider an internal (non-leaf) metablock M. Take the points that are inserted into its children and build a corner structure for these points as prescribed by Lemma 4.1.1. A diagonal corner query on the whole metablock tree is also a diagonal corner query on this new corner structure and can be answered optimally as long as the number of points in it is less than $O(B^2)$!* We call this corner structure the *TD corner structure* for M.

As in the case of a metablock, this TD corner structure will have an update block and will be rebuilt once in B insertions into it. When the size of the TD structure of a metablock M becomes B^2 points, we discard the TD corner structure and rebuild the TS structures of all the children of M (taking into account the points that were in the TD corner structure). Call this reorganization a TS reorganization of the children of M.

We cannot let the size of a metablock increase indefinitely. Our bounds hold only when the number of points in a metablock is $O(B^2)$. Therefore, when the number of points in a metablock reaches $2B^2$, we perform the following actions (if the metablock is a non-leaf

metablock):

1. select the top B^2 points (with the largest y values) of the metablock and build the vertical, horizontal and corner (if necessary) organizations of the metablock (this takes $O(B)$ I/Os),
2. insert the bottom B^2 points of the metablock into the appropriate children depending on the x values of the points,
3. perform a TS reorganization of M and its siblings (this takes $O(B^2)$ I/Os).

We call such a reorganization a *level II* reorganization of a metablock. Obviously, this scheme will not work if the metablock M is a leaf because there are no children to insert into. In that case, we perform the following actions:

1. split the metablock into two children containing B^2 points each and update the control information of the metablock's parents, and,
2. perform a TS reorganization of the two new metablocks and its siblings.

We call this reorganization a *level II* reorganization of a leaf.

The final detail has to do with the branching factor of the parent of a leaf that just split. We will let this grow from the initial value of B to $2B$. Once it reaches $2B$, we split the parent itself into two by reorganizing the entire metablock tree rooted at the parent. One subtree will contain the leftmost B leaves, and the other will contain the rightmost B leaves. We insert the two new roots of these subtrees above in place of the parent. This procedure has to be continued up recursively as necessary. We show that this will not take too much time by showing that such reorganizations cannot happen too often.

Figure 4.12 is an algorithm for performing updates in an augmented metablock tree that presents concisely the steps that we have discussed so far.

(1) **procedure** *insert-point* (p)
(2) $M :=$ the metablock p falls in
(3) $P :=$ M's parent
(4) Add p to M's update block
(5) Add p to P's TD update block
(6) If M's update block is full then
perform a level I restructuring of M
(7) If P's TD update block is full then
rebuild the TD corner structure
(8) If the size of P's TD corner structure is B^2 then
discard the points in the structure and perform a TS reorganization of the children of P
(9) If M is a non-leaf metablock and contains $2B^2$ points then
perform a level II reorganization of M
(10) If M is a leaf and contains $2B^2$ points then
perform a level II reorganization of the leaf and call *propagate-branching-factor(P)*

(11) **procedure** *propagate-branching-factor* (M)
(12) $P :=$ M's parent
(13) If the branching factor of M is $2B$ then
reorganize M into two equal subtrees rooted at metablocks M_L and M_R and insert these into P in place of M. Perform a TS reorganization of M_L, M_R and their siblings. Recursively call *propagate-branching-factor(P)*
(14) If (13) cannot be done because M is the root, create a new root with two children M_L and M_R

Figure 4.12: An algorithm for inserting a point into an augmented metablock tree. *insert-point()* is the main procedure for inserting a point. It inserts a point and propagates insertions down the tree as needed. *propagate-branching-factor()* is a subroutine which splits metablocks and propagates changes up the tree.

Lemma 4.2.1 *The space used by an augmented metablock tree containing n points is $O(n/B)$ blocks of size B.*

Proof: The additional structures that we have added to the metablock tree are as follows:

- the update block for every metablock, and
- the TD corner structure and its update block for every non-leaf node.

Every metablock contains $O(B^2)$ points and having one update block per metablock obviously does not add to the asymptotic complexity of the storage used. A TD corner structure for a non-leaf block contains at most B^2 points and therefore occupies no more than $O(B)$ blocks as per Lemma 4.1.1. Since every metablock contains at least $O(B^2)$ points, we can charge the cost of the TD corner structure to it. This implies that the total storage used by the augmented metablock tree is $O(n/B)$ disk blocks. □

Lemma 4.2.2 *An augmented metablock tree containing n points can answer diagonal corner queries in optimal $O(\log_B n + t/B)$ I/O operations.*

Proof: The search procedure for an augmented metablock tree is very similar to that of a normal metablock tree. The differences are as follows:

- Every time a horizontal organization, or a vertical organization, or a corner structure of a metablock is to be examined, we will examine the update block of the metablock as well. This obviously will not add to the asymptotic complexity of the querying because examining any of these organizations imposes an overhead of at least one disk I/O, and the update block increases it by at most one.
- Every time a TS structure of a metablock needs to be examined, we will examine the TD corner structure of the parent. This does not change the asymptotic complexity of the query because the TD corner structure imposes only a constant overhead for a search (outside of the reporting, which pays for itself), and this overhead is imposed by the TS structure anyway.

Modulo these changes, the query procedure remains exactly the same. The lemma follows from the previous proof for Theorem 4.1.2. □

Lemma 4.2.3 *Insertions into an augmented metablock tree containing n points can be performed in $O(\log_B n + (\log_B n)^2/B)$ amortized I/O operations.*

Proof: Finding and inserting a point into the appropriate metablock involves $O(\log_B n)$ disk I/Os.

Let us consider the different reorganizations that the insertion of a point into a metablock M can lead to and list the amortized costs of each of these reorganizations:

1. The insertion of a point can cause a level I reorganization of the metablock M into which it is inserted. Since this happens only once per B inserts, the amortized cost for this is $O(1)$.

2. The insertion of a point can cause a reorganization of the TD corner structure of M's parent metablock. Since this happens only once per B inserts, the amortized cost for this is $O(1)$.

3. The insertion of a point can cause a TS reorganization of M and its siblings. This happens once in $O(B^2)$ inserts and costs $O(B^2)$ I/Os. The amortized cost is $O(1)$.

4. The insertion of a point can cause a level II reorganization of the metablock into which it is inserted. The amortized cost of this is also $O(1)$ because this happens once in B^2 insertions and costs $O(B^2)$ disk I/Os. Note that we don't count the cost of inserting the points further down in the tree nor the cost of keeping the branching factor within $2B$ here. They are discussed separately below.

At the end of a level II reorganization (which must eventually happen as points are continuously being inserted into M), the inserted point has been either inserted into a child of M or has forced some other point to be inserted there. This is because after M reaches a size of $2B^2$, the bottom B^2 points are inserted into M's children. This means that an inserted point can cause these reorganizations all the way down to the leaf. This gives us a total amortized cost of $O(\log_B n)$ for these reorganizations.

At the end of the situation described above, the inserted points have trickled down to the bottom of the metablock tree or have caused other points to get trickled down. The final cost for the insertion comes from the fact that we have to reorganize subtrees once the branching factor of a node reaches $2B$. The following statements are easily proved by induction:

- If a subtree rooted at M has k points to start with and no branching factor in the subtree exceeds B, we have to insert at least k points into it before the branching factor of M becomes $2B$.
- It costs $O((k/B)\log_B k)$ disk I/Os to build a perfectly balanced metablock tree with k points. The amortized cost per point for this building is $((\log_B k)/B)$.

An inserted point contributes to the cost of these rebuilds from the leaf level all the way upto the root. The amortized cost per point is

$$\sum_{x=1}^{x=\log_B n} x/B = O((\log_B n)^2/B)$$

The lemma follows when we add up the costs for the trickling down and the rebuilding. □

We put everything together in the following theorem:

Theorem 4.2.4 *A set of n data points (x, y) in the planar region $y \geqslant x$ and $y > 0$ can be organized into an augmented metablock tree with blocks of size B, such that a diagonal corner query with t data points in its query region can be performed in $O(\log_B n + t/B)$ I/O operations and points can be inserted into this data structure at an amortized cost of $O(\log_B n + (\log_B n)^2/B)$ disk I/Os. The size of the data structure is $O(n/B)$ blocks of size B each.*

Chapter 5

A Class Indexing Algorithm Using Hierarchy Decomposition

In Chapter 3 we showed how to solve the class indexing problem such that the worst-case query time is $O(\log_2 c \log_B n + t/B)$, the worst-case update time is $O(\log_2 c \log_B n)$, and the storage used is $O((n/B)(\log_2 c))$ disk blocks. Here c is the size of the class hierarchy, n is the size of the problem and B is the disk block size.

In this chapter, we have a preliminary lemma concerning our ability to answer three-sided queries. We then consider two extremes of the class indexing problem and show that they both have efficient solutions. We call a class hierarchy *degenerate* when it consists of a tree where every node has only one child. We give efficient solutions to the class indexing problem when the hierarchy is degenerate and when the hierarchy has constant depth. Combining these techniques, we give an efficient solution to the whole problem.

5.1 Extremes of the Class Indexing Problem

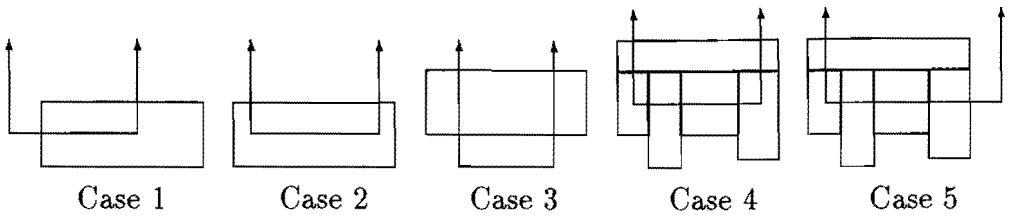


Figure 5.1: Problems with using the metablock tree for three-sided queries.

Lemma 5.1.1 *[18] There exists a data structure that can answer any three-sided query on a set of n points on the plane in $O(\log_2 n + t/B)$ disk I/Os. This data structure occupies $O(n/B)$ disk blocks and can be built in $O((n/B)\log_B n)$ disk I/Os.*

A data structure to achieve these bounds was presented in [18]. The data structure is essentially a priority search tree where each node contains B points. A simple recursive algorithm can build this tree in $O((n/B)\log_B n)$ disk I/Os.

Lemma 5.1.2 *Consider an instance of the class indexing problem where k is the maximum depth of the class hierarchy and n is the size of the problem instance. We can index the class hierarchy so that the worst-case query time is $O(\log_B n + t/B)$, the worst-case update time is $O(k \log_B n)$, and the scheme uses $O((n/B)\ k)$ storage. This is optimal when k is constant.*

Proof: We simply keep the full extent of a class in a collection associated with that class and build an index for this collection. This might entail copying an item at most k times, since k is the maximum depth of the hierarchy. The bounds for the query and update times, and the storage space follow. And clearly, this is optimal when k is a constant. □

It should be pointed out that this lemma does not contradict Theorem 3.2.6 which applies when we have only one copy of the objects in the database. Lemma 5.1.2 uses k copies to index efficiently.

Lemma 5.1.3 *When the hierarchy is degenerate, the class indexing problem reduces to answering three-sided queries in secondary memory and can be solved using a variant of the metablock tree such that the worst-case query time is* $O(\log_B n + \log_2 B + t/B)$.

Proof: In Chapter 3, we reduced the class indexing problem to two-dimensional range searching in secondary memory. To see why this reduces to answering three-sided queries if the hierarchy is degenerate, consider the behavior of procedure *label-class* in Figure 3.3 on a degenerate hierarchy. The root of this hierarchy will be associated with the range $[0, 1)$, its child with $[\frac{1}{2}, 1)$, the grandchild with $[\frac{3}{4}, 1)$ and so on. Each successive range is completely contained in its previous range. From this, we can infer that a query on the full extent of a class is precisely a three-sided query.

We will try to modify the metablock tree to solve this problem. The metablock tree solves two-sided range queries in secondary memory where the corner always lies on the diagonal. Three-sided queries are different for the following reasons: (1) the corners need not lie on the diagonal of a metablock; (2) both corners may lie on the same metablock for this problem, forcing us to answer a three-sided query on a metablock; (3) both the vertical sides of a three-sided query may pass through the same metablock (remember that a two-sided query has only one vertical side); (4) the two vertical sides of the query may lie on metablocks which are children of the same metablock (This makes the TS structures useless for this case because they contain points from *all* the siblings to the left. We now are interested in only a subset of them.); and (5) in the construction of the TS structures in the metablock tree, we build them assuming that the query will always contain the left siblings, never the right siblings. With three-sided queries, this assumption is no longer true. See Figure 5.1 illustrating these cases.

We deal with these problems one by one. To handle (1) and (2), we build, for the points in each metablock, a data structure to answer three-sided queries as prescribed in Lemma 5.1.1. Since that data structure uses optimal storage, our asymptotic storage does not change. Also, since three-sided queries are generalizations of diagonal queries, we dispense with the corner structures we used for the metablock tree. Case (3) requires no

special handling because we can determine the points of a metablock that lie in between two vertical lines by looking at the vertical organization for the metablock. We handle (5) by building two TS structures for every metablock. One will contain points from left siblings and the other from right siblings.

The most difficult problem is that of case (4). In order to handle cases like this, where the two vertical sides of the query lie on metablocks that are siblings, we perform the following action for every interior (non-leaf) metablock M: we combine the points of the children of M (to get a total of $O(B^3)$ points) and build a data structure to answer three-sided queries as prescribed in Lemma 5.1.1. The understanding is that this structure will be used whenever case (4) occurs. This three-sided structure will be called the three-sided structure for the children of M. Figure 5.2 gives a modified version of procedure *diagonal-query* to handle three-sided queries.

The time bound analysis is very similar to the one in Theorem 4.1.2. While answering a three-sided query, no more than three three-sided strucures have to be accessed: one each for the two corners of the query and one for the case where the vertical sides of the query fall on sibling metablocks(i.e., case (4)). All these three-sided structures have B^3 or less points in them and by the bounds of Lemma 5.1.1 can be used to answer three-sided queries in $O(\log_2 B + t/B)$ disk I/Os. Combining this with the bounds of Theorem 4.1.2, we get this lemma. □

Lemma 5.1.4 *There exists a data structure that can answer any three-sided query on a set of n points in $O(\log_B n + \log_2 B + t/B)$ disk I/Os. Points can be inserted to this data structure at an amortized cost of $O(\log_B n + (\log_B^2 n)/B)$ per operation, and the storage space required is $O(n/B)$.*

Proof: The proof of this lemma parallels that of Lemma 4.2.3. The corner structures that we build for that proof become three-sided structures. In particular, the TD corner structure for an internal node M becomes a three-sided structure also. This will have an update block as the TD corner structure did before and will be rebuilt once in B insertions.

```
(1) procedure 3sided-query ( query q, node M);
(2)  if query q is three-sided then
(3)       if M contains both the corners of q
(4)            use the three-sided structure for M to answer the query; return;
(5)       else if both the vertical sides of query q fall inside one child of M
(6)            let this child be M_c
(7)            use M's vertically oriented blocks to report points that lie in between
                    the vertical sides of the query q
(8)            invoke 3sided-query (q, M_c);
(9)       else /* the two vertical sides fall on different children */
(10)           use M's vertically oriented blocks to report points that lie
                    in between the vertical lines of q
(11)           let M_l and M_r be the children of M containing the left and right sides
                    of q resp.
(12)           let M_1, M_2, ..., M_k be the children of M in between M_l and M_r
(13)           use the three-sided structure for M's children to determine points of
                    M_1, M_2, ..., M_k that fall inside the query
(14)           if any of M_1, M_2, ..., M_k fall completely inside the query, examine its
                    children using the horizontally oriented blocks and continue
                    downwards until the boundary of the query is reached for every
                    metablock so examined.
(15)           invoke 3sided-query(left side of q, M_l)
(16)           invoke 3sided-query(right side of q, M_r)
(17) else /* query q is two-sided */
(18)      if M contains the corner of query q
(19)           use the three-sided structure for M to answer the query; return;
(19)      else /* corner of q does not fall inside M */
(20)           use M's vertically oriented blocks to report points of M that lie inside q
(21)           let M_l be the child of M containing the vertical side of q
(22)           let M_1, M_2, ..., M_k be the other children of M intersecting q
(23)           use the appropriate TS structure of M_l to determine the points of
                    M_1, M_2, ..., M_k and their descendants that fall inside the
                    query (as above)
(24)           invoke 3sided-query(q, M_l)
```

Figure 5.2: Procedure to answer three-sided queries.

When its size reaches B^2, the three-sided structure for the children of M will be rebuilt, as will the TS structures for the children of M.

As before, a level I reorganization of a metablock involves the rebuilding of the vertical,

horizontal and three-sided organizations of a metablock. A level II reorganization (remember that a level II reorganization is done when the number of points in a metablock reaches $2B^2$) of a non-leaf metablock M involves: (1) the rebuilding of the vertical, horizontal and three-sided organizations for the top B^2 points in it; (2) the insertion of the bottom B^2 points into the children of M; (3) a TS reorganization of M and its siblings and (4) a rebuilding of the three-sided structure built for M and its siblings.

Similarly, a level II reorganization of a leaf metablock M involves: (1) the splitting of the leaf into two; (2) a TS reorganization of the siblings of M; and (3) a rebuilding of the three-sided structures for the siblings of M. Procedure *insert-point* in Figure 4.12, with minor modifications that take into account the three-sided structures, can be used to perform inserts.

To get the bounds for the number of I/Os for an insert, we note that a three-sided structure with B^2 (respectively B^3) points can be rebuilt in $O(B)$ ($O(B^2)$) disk I/Os as per Lemma 5.1.1. The analysis for Lemma 4.2.3 applies here and gives us the required bounds. □

5.2 Rake and Contract

In this section, we show how to combine the two lemmas above so that we can deal with any class hierarchy. We restrict our attention to hierarchies that are trees. The procedure trivially extends to forest hierarchies. Before that, we need an algorithm that enables us to decide which of the two lemmas to apply on which part of the hierarchy. The idea for the hierarchy tree labeling algorithm is from a dynamic tree algorithm of [37]. The following lemma is easily proven by induction.

Lemma 5.2.1 *Let the procedure* label-edges *shown in Figure 5.3 be applied to an arbitrary hierarchy tree of size c. The number of thin edges from a leaf of this hierarchy tree to the root is no more than* $\log_2 c$.

```
(1) procedure label-edges ( root );
(2)       S := { children of root };
(3)       Max := element of S with the maximum number of descendants;
              /* Break ties arbitrarily */
(4)       Label edge between Max and root as thick;
(5)       Label edges between other children and root as thin;
(6)       Apply procedure label-edges recursively to each child of root;
              /*order irrevelant*/
```

Figure 5.3: An algorithm for labeling a tree with thick and thin edges as used in Lemma 5.2.1.

```
(1) procedure rake-and-contract ( root );
(2)  repeat
(3)       for each leaf L connected by means of a thin edge to the tree do
(4)            index collection associated with L;
(5)            copy items in L's collection to its parent's collection;
(6)            delete L from the tree and mark L as indexed;
(7)       endfor
(8)       for each path in the hierarchy tree composed entirely of thick edges whose sole
              connection to the rest of the tree is by means of a thin edge
              (or ends in the root) do
(9)            build a three-sided structure for the path (as described in Lemma 5.1.3);
(10)           copy all the collections associated with the nodes of the path into the
                   collection of the parent of the topmost node in the path;
(11)           mark all the nodes in the path as indexed;
(12)           delete all the nodes in the path from the hierarchy tree;
(13)      endfor
(14) until hierarchy tree is made up of one node only;
```

Figure 5.4: The rake and contract algorithm for reducing an arbitrary hierarchy tree.

We are now ready to prove the key lemma. The procedure *rake-and-contract* shown in Figure 5.4 takes as input a hierarchy processed by *label-edges* and applies procedures outlined in the proofs of Lemmas 5.1.2 and 5.1.3 appropriately to parts of the hierarchy. Initially, we associate a unique collection with each class. This collection will contain the extent (not the full extent) of the class.

Lemma 5.2.2 *Let an instance of the class indexing problem have class hierarchy size c, problem size n. Let us index the collections as per the procedure* rake-and-contract. *Then we have:*

1. *No extent of any class is duplicated more than $\log_2 c$ times;*
2. *Every class in the input class hierarchy gets indexed in the sense that either an explicit index is built for its full extent (which means that a query on this class is simply a one-dimensional query on a B^+-tree) or a three-sided structure is built for it as per Lemma 5.1.3 (which means that a query on this class can be answered by performing a three-sided query on the three-sided structure).*

Proof: Consider the first part of the lemma. In procedure *rake-and-contract*, we copy the extent of a class as many times as there are thin edges from it to the root of the hierarchy. We know from Lemma 5.2.1 that there are no more than $\log_2 c$ thin edges from any leaf to the root. Part 1 of the lemma follows.

It is easy to see that one of the two *for* loops in procedure *rake-and-contract* runs at least once unless the hierarchy has size one. This is a simple proof: (1) In the beginning, there are leaves attached by means of thin edges to their parents; and (2) if the nodes attached by means of thin edges are removed, there are path(s) composed entirely of thick edges (these paths exist because every interior node has at least one thick edge coming out of it), whose sole connection to the rest of the tree is a thin edge. Once the nodes in a thick path are deleted, case (2) has to apply again for the same reason. This implies that every iteration of the repeat loop reduces the size of the hierarchy, which implies that the algorithm will terminate.

To prove part 2, we first claim that if a node is deleted from the hierarchy in the first *for* loop in procedure *rake-and-contract*, its collection must have contained the full extent of the class that the node corresponds to. This trivially follows because only leaves get deleted in the first *for* loop (because every interior node has a thick path coming out of it and therefore

will be deleted only in the second *for* loop). We now claim that if a node is deleted from the hierarchy in the second *for* loop in procedure *rake-and-contract*, its collection contains the extents of all the classes in its descendants except for the ones attached to it by means of a thick path. This is easily proved by noting that whenever we delete nodes, we copy their collections to the parent upwards in the tree.

It is easy to show Part 2 of the lemma now. If a node is deleted in the first *for* loop, a class indexing query on the corresponding class is simply a one-dimensional query on the index built for it, since this index contains the complete extent of the class. If a node is deleted in the second *for* loop, it must have been part of a path consisting entirely of thick edges. Further, we know from our previous claim that every node in this path contains its complete extent except for the nodes in the thick path below it. In other words, the class corresponding to the node can be thought of as belonging to a degenerate hierarchy. Lemma 5.1.3 applies here and therefore an indexing query on the class can be answered by looking at the three-sided query structure built in *rake-and-contract*. The lemma follows. See Figure 5.5 for an example of how a hierarchy is processed. □

We put everything together in the following theorem. Note that the bounds for insertion come from the fact that an object can be represented no more than $\log_2 c$ times in the indexes built by procedure *rake-and-contract*.

Theorem 5.2.3 *An instance of the class indexing problem, where c is the size of the input class hierarchy, n is the size of the problem, and B is the disk block size, can be solved such that the worst-case query time is $O(\log_B n + t/B + \log_2 B)$, the amortized insertion time is $O((\log_2 c)(\log_B n + (\log_B^2 n)/B)$ per operation, and the storage space required is $O((n/B)\log_2 c)$.*

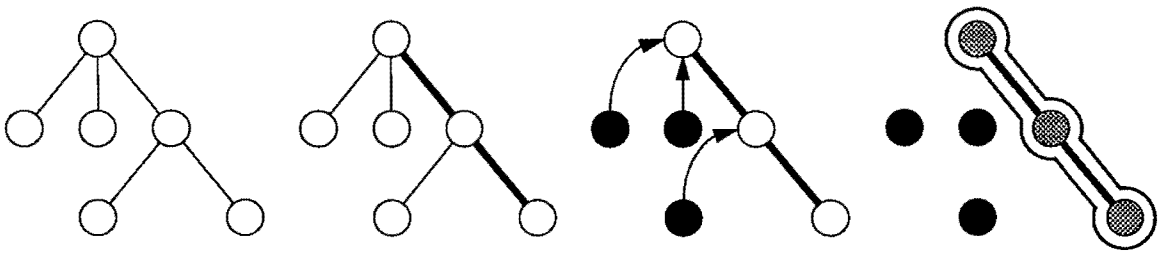


Figure 5.5: An example class hierarchy decomposition

Chapter 6

Path Caching

In this chapter, we present a simple technique called path caching that can be used to transform many in-core data structures like the priority search tree [27], segment tree [3], and interval tree [12,13] to efficient secondary storage structures. These data structures are important because they solve the problems that we are concerned with—like interval management and three-sided searching— efficiently in main memory.

Section 6.1 explains the general principles behind path caching by applying it to segment trees. Section 6.2 explains the application of path caching to priority search trees to obtain reasonably good worst-case bounds for two-sided range searching. We also present results about the application of path caching to three-sided range searching, segment trees and interval trees. Section 6.3 applies recursion and path caching to two-sided queries to improve on the bounds of Section 6.2. It also briefly touches on applying similar ideas to three-sided queries. Section 6.4 shows how updates to the data structures can be handled in optimal amortized time.

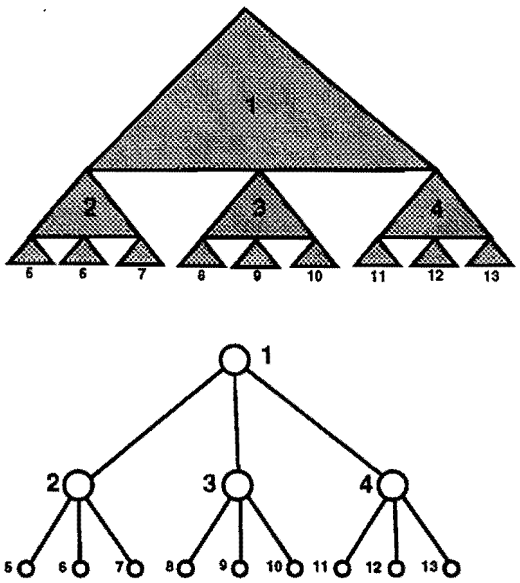


Figure 6.1: Constructing the skeletal graph

it can be shown that we require $\Omega(\log_B n)$ I/Os to identify the path to y. Thus an ideal implementation of segment trees in secondary memory would require $O(\frac{n}{B}\log n)$ disk blocks of space and would answer stabbing queries with $O(\log_B n + t/B)$ I/Os.

To lower the time required to locate the root-to-y path P we can store the tree T in a blocked fashion by mapping subtrees of height $\log B$ into disk blocks. The resulting search structure is called the *skeletal B-tree* and is similar in structure to a B-tree (see Figure 6.1). With this blocking, and a searching strategy similar to B-trees we can locate a $\log B$-sized portion of P with every I/O. If the cover-list of each node is stored in a blocked fashion (with B intervals per block) then we could examine the cover-list $CL(x)$ of each node x on P and retrieve the intervals in $CL(x)$ B at a time. A closer look reveals that this approach could result in a query time of $O(\log n + t/B)$. This is because even though we can identify P in $O(\log_B n)$ time we still have to do at least $O(\log n)$ I/Os, one for each cover-list on the path (see Figure 6.2). These I/Os may be wasteful if the cover-lists contain fewer than B intervals. To avoid paying the additional $\log n$ in the query time we need to avoid *wasteful* I/Os (ones that return fewer than B intervals) as much as possible. In particular, if the number of wasteful I/Os are smaller than the number of *useful* I/Os (ones that return B

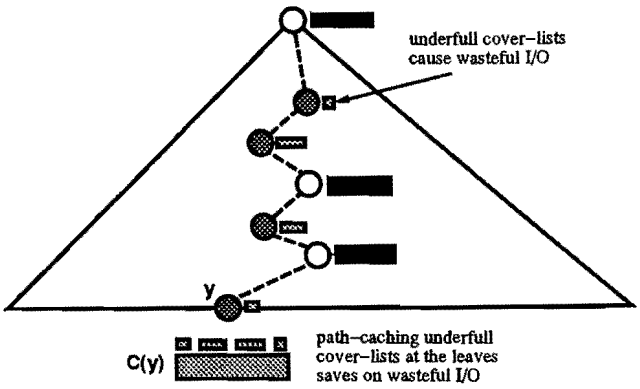


Figure 6.2: Underfull cover-lists along the query path result in wasteful I/Os. Path-caching alleviates this problem.

intervals) then the I/Os required to report all the intervals on P will be $\leqslant 2t/B = O(t/B)$[2]. This combined with fact that we need $O(\log_B n)$ time to identify P would give us the desired query time.

Consider a node x on P such that its cover-list has at least B intervals. It is then easy to see that the first I/O at node x will be useful. In fact all but the last I/O at node x will return B intervals. This implies that the number of wasteful I/Os at node x is upper-bounded by the number of useful I/Os at that node. Thus the problem nodes are only those that have fewer than B intervals stored at them.

This leads us to the idea of **path caching:** *If we coalesce all the cover-lists on P that have $\leqslant B$ elements and store it in a cache $C(y)$ at y then we could look at $C(y)$ instead of looking at* $\log n$ *possibly underfull cover-lists.* If $C(y)$ is stored in a blocked fashion then retrieving intervals from it would cause at most one wasteful I/O instead of $\log n$ wasteful ones (see Figure 6.2). The time for reporting all the intervals would then be $\leqslant 2t/B + 1$. This combined with the time for finding P would give us the desired query time.

We therefore make the following modification to the segment tree:

For each leaf y identify the underfull cover-lists $CL_1, \cdots CL_k$ (cover-lists that contain less

[2]In other words each wasteful I/O will be *paid for* by performing a useful one.

than B intervals) along the root-to-y path. Make copies of the intervals in all the underfull cover-lists and store it in a cache $C(y)$ in y. Block $C(y)$ into blocks of size B on to secondary memory.

From the above discussions we can see that using this modified version of a segment tree we can answer stabbing queries with $O(\log_B n + t/B)$ I/Os. The only thing left to analyze is the amount of storage required for the modified data structure. The number of disk blocks required to block the search tree T is $O(n/B)$. The total number of intervals in all the cover-lists is $O(n \log n)$. These can be stored in $O(\frac{n}{B} \log n)$ disk blocks. At each leaf y we have a cache $C(y)$ that contains up to $B \log n$ intervals (from the $\log n$ nodes along the root-to-y path). Therefore to store the caches from the $2n$ leaves we need $2n \log n$ disk blocks. Putting all of this together we see that the space required is $O(n \log n)$ blocks.

The space overhead can be reduced to the optimal value $O(\frac{n}{B} \log n)$ by performing two optimizations: (1) Building caches at the leaf nodes of the skeletal tree instead of the complete binary tree (we therefore have to build only $O(n/B)$ path-caches); and (2) By requiring the query to look at $O(\log_B n)$ path-caches instead of just one (this would allow us to build smaller path-caches). We can get a secondary memory implementation of the segment tree that requires $O(\frac{n}{B} \log n)$ space and answers queries with $O(\log_B n + t/B)$ I/Os.

6.2 Priority search trees and path caching

In this section, we apply the idea of path caching to priority search trees and use them to solve special cases of two-dimensional range searching. We first consider two-sided two-dimensional queries that are bound on two sides and free on the other two as shown in Figure 1.1. As before, n will indicate the number of data items, B the size of the disk page, and t the number of items in the query result.

Let us consider the implementation of priority search trees in secondary memory for answering two-sided queries. The input is a set of n points in the plane. The priority search

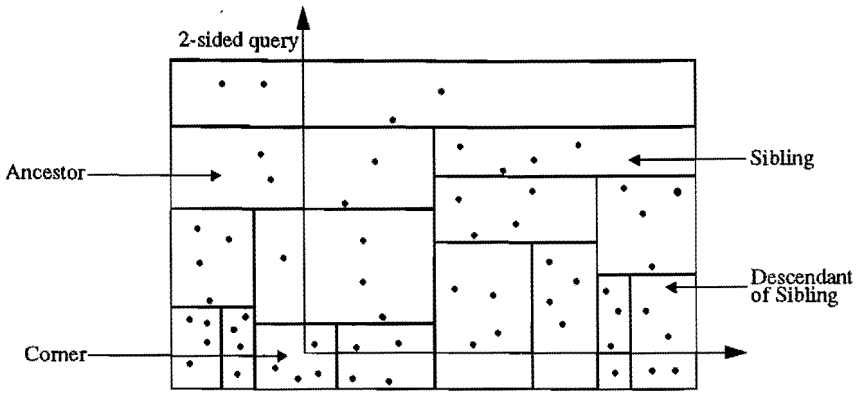


Figure 6.3: Binary tree implementation of Priority Search Tree in secondary memory showing corner, ancestor, sibling and sibling's descendant. Here, B is 4.

tree is a combination of a heap and balanced binary search tree. The solution proposed in [18] works as follows: Find the top B points (on the basis of their y values) in the input set, store them in a disk block and associate this disk block with the root of the search tree. Divide the remaining points into two equal sets based on their x values. From these two sets, choose the top B points in each set and associate them with the two children of the root. Continue this division recursively. The skeletal structure for the binary tree itself is stored in a B-tree (called the skeletal B-tree). It is clear that to store n points in this fashion, we will use only $O(n/B)$ disk blocks.

As illustrated in Figure 6.3, each node in the priority search tree defined above corresponds to a rectangular region in the plane that contains all the points stored in that node. Furthermore, the tree as a whole defines a hierarchical decomposition of the plane. As is shown in [18], this organization has the following crucial property: A point in a node x can belong in a two-sided query if (1) the region corresponding to x's parent is completely contained within the query or, (2) the region corresponding to x or the one corresponding to its parent is cut by the left side of the query.

With this division, we can show that a two-sided query with t points in the output can be answered by looking at only $O(\log n + t/B)$ disk blocks. To do that, we classify nodes that contain points inside the query into four categories as follows:

- *The corner:* This is the node whose region contains the corner of the query.
- *Ancestors of the corner:* These are nodes whose regions are cut by the left side of the query and there can be at most $O(\log n)$ such nodes.
- *Right siblings of the corner and the ancestors:* These are nodes whose parents' regions are cut by the left side of the query. There can be at most $O(\log n)$ such nodes.
- *Descendants of right siblings:* There can be an unbounded number of them, but for every such node, its parent's region has to be completely contained inside the query. That pays for the cost of looking into these nodes. That is, for every k descendant blocks that are partially cut by the query, there will be at least $\frac{k}{2}$ blocks that lie completely inside the query.

The algorithm proceeds by locating the nodes intersecting the left side of our query. This is done by performing a search on the skeletal B-tree. The nodes are examined to find the points inside the query. Next, right siblings of these nodes and their descendants are examined in a top-down fashion until the bottom boundary of the query is crossed. In this algorithm, for each node examined, we perform one I/O operation. The corner, ancestor, and sibling nodes can cause wasteful I/Os but there are at most $O(\log n)$ such nodes. For every descendant of a "sibling" that is examined, its parent would have contributed an useful I/O. From this analysis, we can conclude that we can answer two-sided queries in $O(\log n + t/B)$ I/Os.

We now show how to avoid the $\log n$ wasteful I/Os by caching the data in the ancestor and sibling nodes. We store two caches associated with the corner. One cache will contain all the data in the ancestors sorted in right-to-left (largest x value first) fashion. Call this cache the A-list. The second cache will contain all the data in the siblings stored in top-to-bottom (largest y value first) fashion. Call this cache the S-list. Using these caches we can answer two-sided queries in $O(\log_B n + t/B)$ I/Os. In order to answer a two-sided query, we simply look at the skeletal B-tree and locate the corner in $O(\log_B n)$ time. We then look at the caches (performing at most two wasteful I/Os) to determine which points

from the ancestors/siblings fall into the query. After this, we look into the descendants if necessary. As discussed above, any wasteful query that is caused by examining a descendant can be counted off against a useful query that is the result of examining its parent. The descendants thus pay for looking into them through their parents. The following lemma follows.

Lemma 6.2.1 *Given n input points on the plane, the data structure described above answers any two-sided query containing t points using $O(\log_B n + t/B)$ I/Os. The storage used is $O(\frac{n}{B} \log n)$ disk blocks of size B each.*

Now, we show that we can bring the storage overhead down to $O(\frac{n}{B} \log B)$. We do this by observing that maintaining caches of size $O(\log n)$ at each node is wasteful. We cut the total path length of $\log n$ into $\log_B n$ segments of size $\log B$. We maintain A-lists and S-lists at each node as before. However, to construct these lists at a node we only examine the ancestors and siblings that are in the $\log B$ segment of the root-to-node path that the node belongs to. Thus the lists at any node contain at most $O(\log B)$ disk blocks. Therefore the total storage required comes down to $O(\frac{n}{B} \log B)$. To answer a query, we now have to look at a total of $\log n / \log B = \log_B n$ A-lists and S-lists(one for each of the $\log B$-sized subpaths). The descendants of siblings are handled as they were in the previous construction. We get the following theorem.

Theorem 6.2.2 *Given n input points on the plane, path caching can be used to construct a data structure that answers any two-sided query using $O(\log_B n + t/B)$ I/Os. Here t is the output size of the query. The data structure requires $O(\frac{n}{B} \log B)$ disk blocks of storage.*

In the next section, we show how we can combine the idea of path caching with recursion to make the storage required even less while keeping queries efficient.

Using similar ideas, we can obtain the following bounds for three-sided queries, segment trees and interval trees.

Theorem 6.2.3 *Given n input points on the plane, path caching can be used to construct a data structure that answers any three-sided query using $O(\log_B n + t/B)$ I/Os. Here, t is the number of points in the query. The data structure requires $O(\frac{n}{B}\log^2 B)$ disk blocks of storage.*

Theorem 6.2.4 *We can implement Segment Trees in secondary memory so that a point enclosure query containing t intervals can be answered in $O(\log_B n + t/B)$ I/Os. For n input intervals, the storage used is $O(\frac{n}{B}\log n)$ disk blocks.*

Theorem 6.2.5 *We can implement Interval Trees in secondary memory so that a point enclosure query containing t intervals can be answered in $O(\log_B n + t/B)$ I/Os. For n input intervals, the storage used is $O(\frac{n}{B}\log B)$ disk blocks.*

We now explore two-sided queries in more detail to improve the bounds obtained in this section. We then discuss algorithms for updating the resulting data structures.

6.3 Using recursion to reduce the space overhead

In this section we describe how to extend the ideas of Section 6.2 to develop a recursive data structure that has a much smaller space overhead and still allows queries to be answered in optimal time. We restrict ourselves to the problem of answering general two-sided queries by using a secondary memory priority search tree. Similar ideas can be used to get better space overheads for the other data structures as well.

We first describe a two-level scheme for building a secondary memory priority search tree that requires only $O(\frac{n}{B}\log\log B)$ storage while still admitting optimal query performance. We then briefly describe a multi-level version of this idea that requires only $O(\frac{n}{B}\log^* B)$ storage.

Recall that the basic scheme divides the points into regions of size B. A careful look at

this scheme shows that the $\log B$ overhead is due to the fact that the ancestor and sibling caches of each of the n/B regions can potentially contain $\log B$ blocks. To reduce the space overhead we could either (1) reduce the amount of information stored in each region's cache or (2) reduce the total number of regions.

A closer look shows that to get optimal query time, with any region, we must store the information associated with $\log B$ of its ancestors. This is because in the priority tree structure the path length from any block to the root is $O(\log n)$. Thus to achieve a query overhead of at most $\log_B n$ we must divide such a path into no more than $\log_B n$ pieces. Since $\log_B n = \log n / \log B$, we see that with each node we must store the information associated with $O(\log B)$ of its ancestors. We therefore turn to the second idea. To get a linear space data structure we build a basic priority search tree that divides the points into regions of size $B \log B$ instead of B. We thus have $n/B \log B$ regions.

To build the caches associated with each of the regions we proceed in a slightly different fashion. First, we sort the points in each region R right-to-left (i.e. largest to smallest) according to their x-coordinates. We store these points (B at a time) in a list of disk blocks associated with R. In the same fashion we also sort the points top-to-bottom (i.e. largest to smallest) and store them in a list of disk blocks associated with R. Thus the points in each region are blocked according to their x as well as their y coordinates. These lists are called the X-list and the Y-list of R respectively.

To build the ancestor cache associated with a region R we look at its $\log B$ immediate ancestors. From each of the ancestor's X-list we copy the points in the first block. We then sort all these points right-to-left according to their x-coordinates and store them in a list of disk blocks associated with region R. These blocks constitute the ancestor list (A-list) of R. Similarly, to build the sibling cache (S-list) of R we consider the first blocks from the Y-lists of the siblings of R and its ancestors. Adding up the all the storage requirements we get the following lemma.

Lemma 6.3.1 *The total storage required to implement the top level priority search tree and*

the associated A, S, X, and Y lists is $O(\frac{n}{B})$.

Unlike in the basic scheme we now have regions which contain $O(B \log B)$ points or $O(\log B)$ disk blocks. To complete our data structure we therefore build secondary level structures for each of these regions. For each region we build a priority search tree as per Lemma 6.2.1. In other words, we divide the region into blocks of size B and build ancestor and sibling caches (as before) for each of the blocks. In this case the height of any path is at most $O(\log \log B)$; therefore for each region we use all of its ancestors and the siblings to construct the ancestor and sibling caches.

We now count the space overhead incurred due to the priority search trees built for each region. For each block in a given region R we need no more than $O(\log \log B)$ disk blocks to build its ancestor and sibling caches. This follows from the fact that the priority search tree of R has height $O(\log \log B)$. A given region R has $O(\log B)$ disk blocks; therefore the space required for storing the ancestor and sibling caches of all the blocks in R is $O(\log B \log \log B)$. Adding this over all the regions we get the following lemma.

Lemma 6.3.2 *The total storage required by the two-level data structure is* $O(\frac{n}{B} \log \log B)$.

Answering queries using the two-level data structure: We now show how to answer two-sided queries using this data structure. To answer such a query we proceed as follows: As in the basic scheme we first determine the region R (in the top level priority search tree) that contains the corner of the query. As discussed in Section 6.2, the points in the query belong to either the region R, or one of R's ancestors Q, or a sibling T of Q (or R), or to a descendant of some sibling T.

To find the points that are in the ancestors and their siblings we look at the $\log_B n$ ancestor and sibling caches along the path from R to the root. From these caches we collect all the points that lie inside the query. However, just looking at these two caches is not enough to guarantee that we have collected all the points from the ancestors and their siblings. This is because we only use one block from each ancestor (and each sibling) to

build the A and S-lists.

To collect the other points in these regions that are in the two-sided query we examine the X and Y-lists of the ancestors and their siblings respectively. These lists are examined block by block until we reach a block that is not fully contained in our query. The X-list of an ancestor Q of R is examined if and only if all the points from Q that were in the ancestor cache of R are found to be inside the two-sided query. Similarly the Y-list of a sibling T (along the path from R to the root) is examined if and only if the points from T that are in the sibling cache are all inside our two-sided query.

We claim that this algorithm will correctly find all points in the ancestor and the sibling regions that are in our query. We now show that all the points in the ancestor regions are found correctly. The case for the siblings of the ancestors is similar. Consider some ancestor region Q of R and its associated right-to-left ordering of the points as represented in its X-list. Since Q is an ancestor, it is cut by the vertical line of the two-sided query (see Figure 6.3). Therefore, all the points in Q that are to the right of the vertical line are in the query and the rest aren't. Thus, all the points in the query are present in consecutive disk blocks (starting from the first one) in the X-list of Q. We therefore need to examine the ith block in the X-list if and only if all the previous $i-1$ blocks are completely contained in our query. Since the first block is part of R's ancestor cache we need to look at the X-list of Q if and only if all of the points of Q in the ancestor cache of R are contained in the two-sided query.

To account for the time taken to find these points we note that there are $O(\log_B n)$ caches that we must look at. It is not hard to see that apart from these initial lookups we only look at a disk-block from a region Q if our last I/O yielded B points (inside the query) from this region. Therefore all the other I/Os are paid for. Thus the time to find these points is $O(\log_B n + (t_A + t_S)/B)$ where t_A and t_S denote the number of points contained in the ancestors and their siblings.

To find the points in the query that are in the descendants of the siblings, we use the same approach as the basic scheme. We find the points in all these regions by scanning

their Y-lists. We traverse a region Q if and only if its parent P is fully contained in the query. An argument similar to the one above shows that all such points are found by this algorithm, and that the number of I/Os required is $O(t_D/B)$. Here t_D denotes the number of points contained in the descendants of the siblings.

To find the points in the query that are in region R we use the second level priority tree associated with R. Let t_R denote the number of points in R that belong to the query. We find these points by asking a two-sided query inside region R. This requires at most $O(t_R/B)$ I/Os (by the arguments in Section 6.2).

Therefore, the total number of I/Os required to answer a two sided query is $O(\log_B n + t/B)$ where t is the size of the output. This in conjunction with Lemma 6.3.2 gives us the following theorem.

Theorem 6.3.3 *There exists a secondary memory static implementation of the priority search tree that can be used to answer general two-sided queries using $O(\log_B n + t/B)$ I/Os; where t is the size of the output. This data structure requires $O(\frac{n}{B} \log\log B)$ disk blocks of space to store n points.*

A multi-level scheme to further lower the space over-head: It is possible to reduce the space overhead further by using more than two levels. The idea is similar to the one used before. At the second stage instead of building a basic priority search tree for each region we build a tree that contains regions of size $B \log\log B$ and build the X, Y, A, and S lists as before. A three-level scheme gives us a space overhead of $O(\frac{n}{B} \log\log\log B)$ while maintaining optimum query time. If we carry this multi-level scheme further then we get a data structure with the following bounds.

Theorem 6.3.4 *There exists a secondary memory static implementation of the priority search tree that can be used to answer general two-sided queries using $O(\log_B n + t/B + \log^* B)$ I/Os; where t is the size of the output. This data structure requires $O(\frac{n}{B} \log^* B)$ disk blocks of space to store n points.*

These ideas can be applied to three-sided queries as well, to reduce the space overhead incurred by the sibling caches. In particular we can get the following bounds for answering three-sided queries.

Theorem 6.3.5 *There exists a secondary memory static implementation of the priority search tree that can be used to answer general three-sided queries using* $O(\log_B n + t/B + \log^* B)$ *I/Os; where* t *is the size of the output. This data structure requires* $O(\frac{n}{B} \log B \log^* B)$ *disk blocks of space to store* n *points.*

6.4 A fully dynamic secondary memory data structure for answering two-sided queries

In this section we show how to dynamize the two-level scheme discussed in Section 6.3. Our data structure is fully dynamic, i.e. it can handle both additions and deletions of points. The amortized time bound for an update is $O(\log_B n)$.

Before we describe our dynamic data structure we first discuss an alternate way of visualizing the top level priority search tree in the two-level scheme. In this tree a node corresponds to a region of size $B \log B$. We partition this priority search tree into subtrees of height $\log B - \log\log B$. Each such subtree is considered a *supernode*. As in Section 6.2, in order to build the ancestor and sibling cache of any region R, we only consider those ancestors (and their siblings) of R that are in the same supernode as R. Considering subtrees of height $\log B - \log\log B$ (instead of $\log B$) does not change the query times because we are now dealing with regions of size $B \log B$.

The advantage of viewing these subtrees as supernodes is that now we can isolate the duplication of information due to the S and A-lists to within a supernode, since none of the caches ever cross the supernode boundary. The layer of supernodes (regions) immediately following the last layer in a supernode N can be thought of as children of N in this

visualization. Note that each supernode N contains $B/\log B$ regions and has $O(B/\log B)$ supernodes as its children. Also note that the number of supernodes on any path in the tree is $O(\log_B n)$.

Our dynamic data structure associates an update buffer U of size B with each supernode N in the tree. It also associates an update buffer u (also of size B) with each region R. These buffers are used to store incoming updates until we have collected enough of them to account for rebuilding some structure. To process a query, we first use the algorithm from Section 6.3 to collect the points that have been entered into the data structure. We then look through the associated update buffers to add new points that have been added and to discard old points that have been deleted by an unprocessed update.

We now need to be careful in claiming optimal query time because the points that we collect by searching the priority search structures may then have to be deleted when we look at the respective update buffers. However, it can be shown that for every $B \log B$ points we collect we can lose at most B points, thus resulting in at most two wasted I/Os for $\log B$ useful ones. Therefore, the loss of points due to unprocessed deletes is very small and does not affect the overall query performance.

Whenever an update occurs, we first locate the supernode N where the update should be made. The update could be a point insertion or a deletion. We then log the update in the associated update buffer U. If the buffer does not overflow we don't do anything. If U overflows then we take all the updates collected and propagate them to the regions in N where they should go. For instance, a point insertion is trickled down to the region of N that contains its coordinates. In each such region we then log the update into the local update buffer u associated with it. We now rebuild the X and Y lists of each region in N taking into account the updates that have percolated into it. We also rebuild each region's ancestor and sibling caches; again taking into account the updates that have percolated into that region.

The number of I/Os required to rebuild the A, S, X, and Y lists of one region R is $O(\log B)$. Therefore, the number of I/Os required to rebuild all the caches is

$(B/\log B)\times O(\log B) = O(B)$. Since we do this only once in B updates the amortized cost of rebuilding the caches is $O(1)$ per update.

If for none of the regions in N any of their buffers overflow, we don't do anything further. Otherwise for each region R whose update buffer has overflowed we rebuild the second level priority search tree associated with it. As before, we take into account the updates in the buffer u of R.

The number of I/Os required to rebuild the priority search tree associated with a given region R is $O(\log B \log\log B)$. This is because we need to rebuild the caches of $\log B$ blocks each of which could contain up to $\log\log B$ disk-blocks of information. Since this is done only once every B updates the amortized time per update is $O((\log B \log\log B)/B) = O(1)$.

For every supernode N; once every $B\log B$ updates we do a rebuild. This rebuilding keeps the same x-division as the old one. Keeping the x-divisions same we change the y-lines of the regions so that each region now contains exactly $B\log B$ points. Using these new regions structure we rebuild the A, S, X, and Y lists for each region as well as the secondary level priority search trees associated with each region. Note that it is important to keep the same x-division to preserve the underlying binary structure of the priority search tree. We cannot view the regions in a given supernode in isolation since they are part of a bigger priority search tree.

To keep the invariant that each region in N contain $B\log B$ points we may have to push points into its children or we may have to borrow points from them. These are logged as updates in the corresponding supernodes. Pushing points into a node is equivalent to adding points to that region while borrowing points from a node is the same as deleting points from the region. These updates may then cause an overflow in the buffers associated with one or more of those supernodes. We repeat the same process described above with any such supernode.

It is easy to show that the number of I/Os required to rebuild a supernode is $O(B\log\log B)$. Since a rebuild is done once every $B\log B$ updates, the amortized time

required for one such rebuild is $O(1)$. However, since we push updates down when we rebuild supernodes, we may have to do up to $\log_B n$ rebuilds (along an entire path) due to a single overflow. Therefore, the amortized cost of a rebuild is $O(\log_B n)$.

A moment of thought reveals that pushing points down is not enough to keep the priority search tree balanced. Repeated additions or deletions to one side can make subtrees unbalanced. We therefore periodically rebuild subtrees in the following manner.

With each node in the tree we associate a size which is the number of points in the subtree rooted at that node. We say that a node is unbalanced if the size of one of its children is more than twice the size of its other child. Whenever this happens at a node R we rebuild the subtree rooted at R. The number of I/Os required to rebuild a priority search tree with x points is $O((x/B)\log_B x + (x/B)\log\log B)$. This is because we need to rebuild the secondary level priority search trees as well as the primary level tree along with all the caches at both levels. Since a subtree of size x can get unbalanced only after $O(x)$ updates we get an amortized rebuilding time of $O((\log_B x + \log\log B)/B) = O(1)$.

Summing up all the I/Os that are required to rebuild various things we see that the total amortized time for an update is $O(\log_B n)$. We therefore have the following theorem.

Theorem 6.4.1 *There exists a fully dynamic secondary memory implementation of the priority search tree that can be used to answer general two-sided queries using* $O(\log_B n + t/B)$ *I/Os; where* t *is the size of the output. The amortized I/O-complexity of processing both deletions and additions of points is* $O(\log_B n)$. *This data structure requires* $O(\frac{n}{B}\log\log B)$ *disk blocks of space to store* n *points.*

Similar ideas can be used to get a dynamic data structure for answering three-sided queries as well. The time to answer queries is still optimal but the time to process updates is not as good. In particular we get the following bounds for answering three-sided queries.

Theorem 6.4.2 *There exists a fully dynamic secondary memory implementation of the priority search tree that can be used to answer general three-sided queries using* $O(\log_B n +$

t/B) I/Os; where t is the size of the output. The amortized I/O-complexity of processing both deletions and additions of points is $O(\log_B n \log^2 B)$. This data structure requires $O(\frac{n}{B} \log B \log\log B)$ disk blocks of space to store n points.

To summarize, we have presented in this chapter, a technique called path caching which can be used to implement many main memory data structures for these problems in secondary memory. Our data structures have optimal query performance at the expense of a slight overhead in storage. Furthermore, our technique is simple enough to allow inserts and deletes in optimal or near optimal amortized time.

Chapter 7

Experimental Work

In this chapter we study the implementation and benchmarking of the class-divisions algorithm of Chapter 3. Section 7.1 reconsiders the possible options for indexing classes, which were discussed briefly in Section 3.2. We identify one particular method that is used very often in OODBs and use that as the point of comparison for the class-divisions algorithm. Section 7.2 then discusses the important implementation details for the comparison we make and outlines the experimental data we want to collect. Section 7.3 presents a series of experiments and the results we obtain from them, and the conclusions derived from the experiments are presented in Section 7.4.

7.1 Indexing Classes Revisited

As before, let c be the number of classes, n the number of objects, and B the page size. One way of indexing classes is to create a B^+-tree per class by indexing the full extent of each class. This gives the optimal query time of $O(\log_B n + t/B)$ I/Os but has a big price: the storage requirement of this algorithm is $O((n/B)c)$ pages and the update time is also

multiplied by c to become $O(c \log_B n)$ I/Os. In a database system, the storage overhead of this method is unacceptable. Even a moderately sized class hierarchy of 20 classes can make the storage overhead of this method very high.

Two other methods for indexing classes are: (1) to build a B^+-tree on the individual extent of each class, and (2) to build a single B^+-tree for all objects by indexing the collection of all objects. We refer to the latter as the single-index method. The extensive comparison of these two methods in [22] concludes that the technique of indexing the collection of all objects is better than indexing each class's individual extent separately. The main reason for this is that the technique of indexing individual extents has a very high query time, $O(c \log_B n + t/B)$: that is, the query time increases linearly with the size of the class hierarchy. For input data whose density on the indexed attribute does not vary dramatically (i.e., for reasonably "well behaved" data), the single-index method offers performance superior to the method of building B^+-trees on the individual extents of each class.

The main disadvantage of the single-index method is its lack of a good worst-case guarantee for query time. This is because we have to combine the collections of objects from all the classes to build the index. When performing a range query on a particular class, we get a mixture of objects that belong to the queried class and objects that do not. We get the correct answer to the range query by "filtering" out the objects of interest. In spite of this disadvantage, this is the method used most often in object-oriented database systems, for example, the O2 database system [1], and thus we use it as our point of reference for the methods we propose.

The only other method for indexing classes appearing in the literature is a scheme called the H-tree [25], based on the idea of threading many B^+-trees together to facilitate simultaneous search. (This idea is known as "fractional cascading" in the data structures literature [8].) The H-tree scheme, however, offers no good performance guarantees for querying and in addition, the update algorithm is complicated and potentially unbounded. Therefore, we do not include this method in our benchmarking.

The solution we proposed in Chapter 5 using rake and contract is too complicated to

implement in practice. The main reason for this is the lack of a data structure to answer three-sided queries efficiently, in a practical sense. The data structure proposed in Chapter 5 to handle three-sided queries is based on the metablock tree, which in itself is a complicated data structure. And the data structure for three-sided queries obtained using path caching has a very high storage overhead (a factor on the order of 40), so that it is effectively unrealistic in a practical setting.

However, the class-divisions algorithm proposed in Section 3.2 is a practical solution. This algorithm uses an existing index structure, namely the B^+-tree. The storage overhead is low, logarithmic in the size of the class hierarchy. For example, for a class hierarchy containing 128 classes, the storage overhead is a factor of seven or less. Query and update times are also low. Accordingly, our experimental work involves the comparison of the single-index and class-divisions algorithms. We use many different class hierarchies and many different experimental situations to gain an overall perspective on the relative performances of the two algorithms.

The goal of our experiments, therefore, is to compare the performance of the class-divisions and single-index algorithms for class indexing. We implement the two algorithms under identical experimental conditions and compare them. Specifically, we compare the following three important characteristics of the two algorithms: (1) the querying efficiency, (2) the storage overhead, and (3) the update time. The next section defines precisely these terms and our techniques for measuring them.

7.2 The Implementation

The successful incorporation of an indexing structure into a database system depends on many factors. While efficiency of the index is an obvious concern, other factors like ease of implementation and interaction with the database system often play an important role. In particular, index structures have intricate interactions with the lock manager. Concurrency control and recovery issues frequently become difficult problems.

Both the single-index method and the class-divisions algorithm are good candidate solutions for the problem of indexing classes because they can be easily integrated into an OODB. This is because both the methods use only the B^+-tree, an index structure that is available in almost all database systems. Because of this, we can avoid many of the integration problems discussed above.

However, we restrict ourselves to measuring the efficiency of the index structures, since it can be thought of as a necessary prerequisite for an index to be successful. In addition, implementing data structures as part of a database kernel is not an option for commercial databases nor for most research prototypes. (This is because source code is often not available for commercial databases, and where available, poses difficult and time-consuming problems in system integration.)

The first design issue to be faced is whether the data structures should be implemented under the aegis of a database system or as a stand-alone unit. We have chosen to do the latter for several reasons, the most important reason being that implementing a data structure from the application programming level of a database system (in O_2C, for example, under the O_2 database system) often introduces many overheads in the measurements we make that might or might not exist in another database system.

A modified version of the Berkeley DB code [36] that went into the making of the POSTGRES database system [39] was used to implement B^+-trees. The code builds B^+-trees on disk using the OS-level file system. The experiments were conducted on Sparc 10 workstations running SunOS 4.1.3. The Berkeley DB code runs at the user level, does its own buffer management, and exercises no control over the virtual memory and paging activity of the operating system. Therefore, in order to count I/Os, we restricted ourselves to measuring the explicit read requests issued by the program. In other words, implicit reads generated by the operating system pager did not figure in the statistics collected.

A hand-optimized version of the class-divisions algorithm was used to create collections that were then indexed using B^+-trees. The class-divisions algorithm of Chapter 3, even though asymptotically optimal, does not exploit the structure of the input class hierarchy

when creating collections. This can result in a query time of $2\log_2 c \log_B n + O(t/B)$ I/Os for range querying (the asymptotic complexity is $O(\log_2 c \log_B n + t/B)$). By precisely tuning the way in which collections are created, we can almost always keep the query time under $\log_2 c \log_B n + O(t/B)$. In most cases, how to do this will be fairly apparent. Rather than go into the details here, we explain the optimizations whenever we present a class hierarchy to be processed by the class-divisions algorithm.

The six example class hierarchies in the experiments were carefully selected to be a mixture of hierarchies that were difficult to index and typical examples from real situations. (For example, some of them came from the O2 book [1].) When populating these class hierarchies, several simplifying assumptions were made:

1. It was assumed that the objects were small. (The objects used in our experiments were always less than 100 bytes long.) Large objects, typically of size greater than the page size of the underlying secondary storage, pose difficult problems that are orthogonal to the ones being considered here.

2. In each class hierarchy, it was assumed that the number of objects in the individual extent of each class was the same.

3. Further, it was assumed that the keys of the objects were uniformly distributed over a range. Other experiments conducted with normalized data (i.e., where the keys were distributed in a "normal" or Gaussian distribution over the range) produced similar results. Queries were also assumed to be uniformly distributed over the range of the keys as well as over the different classes in the class hierarchy.

In the actual experiment, there are many variables we can control:

- data size. This is the number of objects that belong to the individual extent of a class.
- ordering of input data. The Berkeley DB code for B^+-trees has a hook that optimizes the space utilization of the B^+-trees produced when the input data is given in sorted

order. Both sorted and unsorted input data are realistic situations, since occasional reorganizations in a database can be considered similar to presenting sorted input to the B-tree algorithm.

- buffer size. Typical values range from 100 kilobytes to 5 megabytes.
- page size (or disk block size). Typical values for the page size range from 512 bytes to 64 kilobytes.
- query size. Queries can be either restricted to retrieve only a small portion of the database (typically $< 10\%$) or totally unrestricted, possibly retrieving the entire database.
- number of queries. This is the number of queries the algorithms are asked to answer in one experiment.

The experiments were performed as follows. First, a class hierarchy was selected for the experiment and values were chosen for each of the variables that could be controlled. Input data was then generated and the B^+-trees required by the single-index and the class-divisions algorithm were created. A query file was then produced. (We usually generated a set of 200 queries.) One process was started to answer queries using the single-index method and after it finished, another was started to answer queries using the class-divisions method. These processes had identical parameters. Queries from the query file were answered one by one and the number of disk I/Os requested for each query was recorded in a file. After this concluded, the output files were processed to compare the number of I/Os each method took for the queries. Since all the test conditions were identical for the two processes, this was a fair comparison. The ratio of the I/Os for the single-index method and the class-divisions algorithm was computed for each query answered and recorded for further processing. This ratio, called the *query efficiency ratio (QE ratio)*, was the most important statistic derived from the experiments.

While the QE ratio helps us determine the relative merits of the two methods for querying, we need other numbers to calculate the storage and update overheads. We use

the following parameters:

- *storage overhead factor.* This is the ratio of the space used by the class-divisions algorithm to that used by the single-index algorithm.
- *maximal replication factor.* In the single-index method, an object is represented at most once, while in the class-divisions method, an object may be replicated. This factor measures the maximum number of times an object is replicated. Note that this factor directly determines the ratio of the time to update objects in the two methods.
- *maximal query factor.* A query in the single-index method can always be answered by looking at only one B^+-tree. This factor measures the overhead per *failed* query incurred by the class-divisions method. In other words, this factor measures the maximum number of B^+-trees we have to search in order to answer a class indexing query when using the class-divisions method.

The QE ratio depends on the experimental conditions under which queries are answered. The other parameters can be determined directly by looking at the indexes needed for the class-divisions algorithm on the target class hierarchy. Consequently, the QE ratio is the statistic we present graphically the most frequently. The other parameters are computed and presented whenever we present a class hierarchy.

We conducted two major sets of experiments. In the first set, hierarchy H1 in Figure 7.1 was used to study the effect of buffer pool size and page size on the QE ratio. A variety of values were assigned to data size, buffer pool size, and page size. Further, both sorted and unsorted inputs as well as small and large queries were considered (for a total of 2400 test runs, answering 48,000 queries and performing many gigabytes of disk activity). In the second set of experiments, which were performed on all six class hierarchies H1–H6 (Figure 7.1 and Figures 7.6–7.10), "moderate" values were chosen for data size, buffer pool size and page size and the QE ratio was computed while varying the other parameters.

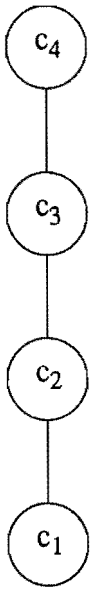


Figure 7.1: Hierarchy H1. For this hierarchy, the storage overhead factor is 2.25, the maximal replication factor 3 and the maximal query factor 2.

7.3 The Experiments

Before presenting the experiments and class hierarchies, we give some notation that will be useful in explaining the workings of the algorithms. $c_1, c_2, c_3, \ldots$ are used to denote classes in the input class hierarchies. They are also used to denote the individual extents of the classes and also the size of these individual extents. The meaning will always be clear from context. C_i indicates the full extent of class i and C_0 refers to the collection of all objects. Since all our class hierarchies are trees, this is the full extent of the root hierarchy.

Example 7.3.1 Consider hierarchy H1 in Figure 7.1. The class-divisions algorithm proceeds by first indexing c_1. c_1 and c_2 are combined to give C_2, which is then indexed. c_3 is indexed next. Finally, all the individual extents are combined to give C_0 (which is the same as C_4), which is also indexed.

Queries on class c_1 are answered using the index on c_1, on c_2 using the index on C_2, on c_3 by using the indexes on c_3 and C_2 simultaneously, and on c_4 using the index on C_0.

The total storage used for the indexes is:

$$(4c_1 + 2c_2 + 2c_3 + c_4)/S$$

where S is the average storage utilization of the B$^+$-trees. S is normally around 0.7 for unordered input for the B$^+$-trees and 0.99 for ordered input.

The single-index method needs only the index on C_0. If we assume that the size of the individual extents is the same, the storage overhead factor for hierarchy H1 is 2.25. (It is equal to the ratio of $(4 + 2 + 2 + 1)/S$ to $4/S$, which is 9/4.) The maximal replication factor is 3, because objects in c_1 are replicated three times. The maximal query factor is 2, because c_3 must be queried using two B$^+$-trees.□

Figures 7.2–7.5 present a set of four graphs that plot the QE ratio against page size and buffer pool size. In the experiments generating these four graphs, the size of each individual class extent was 20,000 objects, and 200 queries uniformly distributed over the range of the key and the four classes in hierarchy H1 were used. The graphs differ in the nature of the queries asked (small or large) and the nature of the input used to construct the B$^+$-trees (sorted or unsorted input).

The graphs show a fairly high QE ratio, proving the usefulness of the class-divisions algorithm even with a relatively small class hierarchy. The single-index method takes at least twice as many I/Os as the class-divisions method in almost all cases, and frequently much more. Given that we are dealing with uniformly distributed data, this is an impressive gain. When the input data is skewed, as real data often tends to be, the performance improvement with the class-divisions method will be even more.

Figures 7.2–7.5 also show that the QE ratio tends to increase with buffer pool size. The reason for this is that, once a significant portion of the database fits into memory, the class-divisions algorithm performs extremely well when queries fall nearby, and this is bound to happen when a large number of queries are asked. The single-index method, on the other hand, cannot take advantage of the large buffer pool size because it has no control

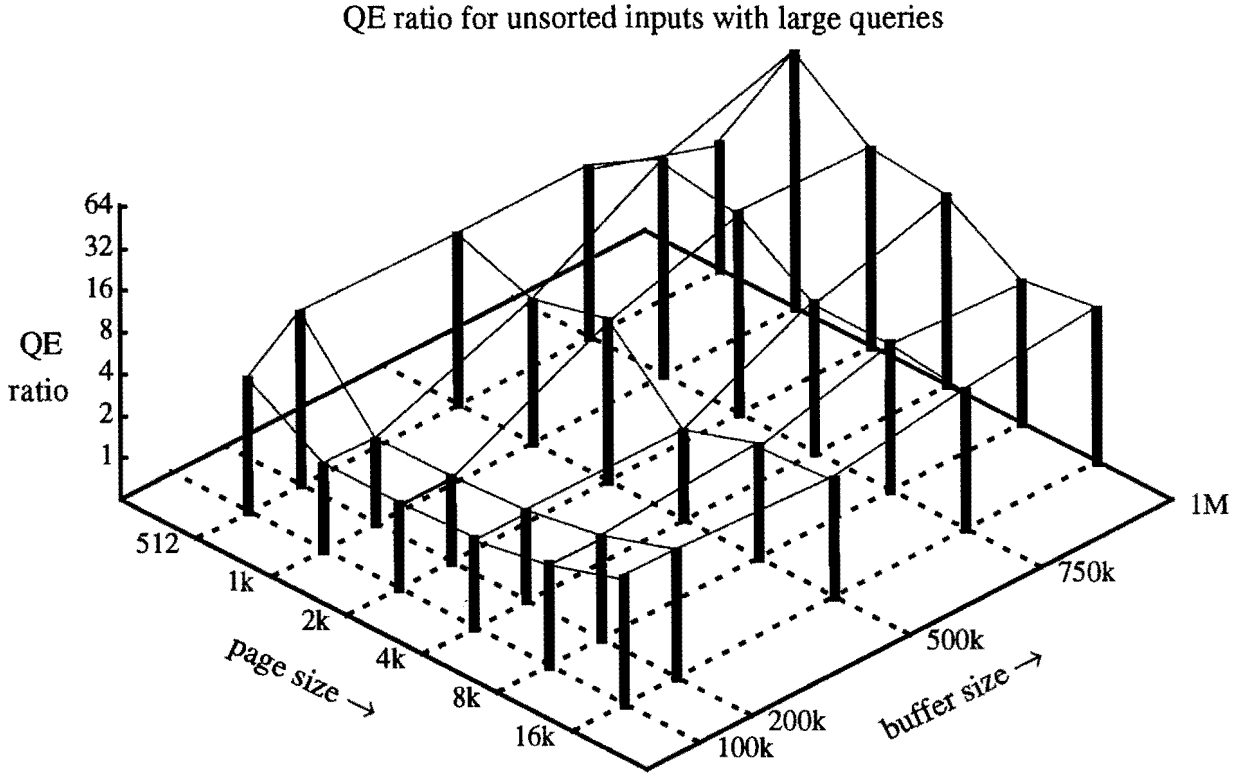


Figure 7.2: Plot of QE ratio against page size and buffer pool size. This experiment, on hierarchy H1, was conducted with unsorted input to the B-tree code, large queries were used, and each class had 20,000 objects.

over how data from different classes are interspersed together. The QE ratio shows little or no change with increasing page size, except when the page size becomes so large that no more than a few pages can fit into the buffer pool. At this point, the QE ratio drops slightly but quickly increases again as the size of the buffer pool increases.

The graphs also show that the QE ratio is not greatly affected by the small/large nature of the queries or the sorted/unsorted nature of the input. It can be observed, however, that the QE ratio is somewhat larger for large queries than for small queries. Similarly, it is a little larger for unsorted input than sorted input.

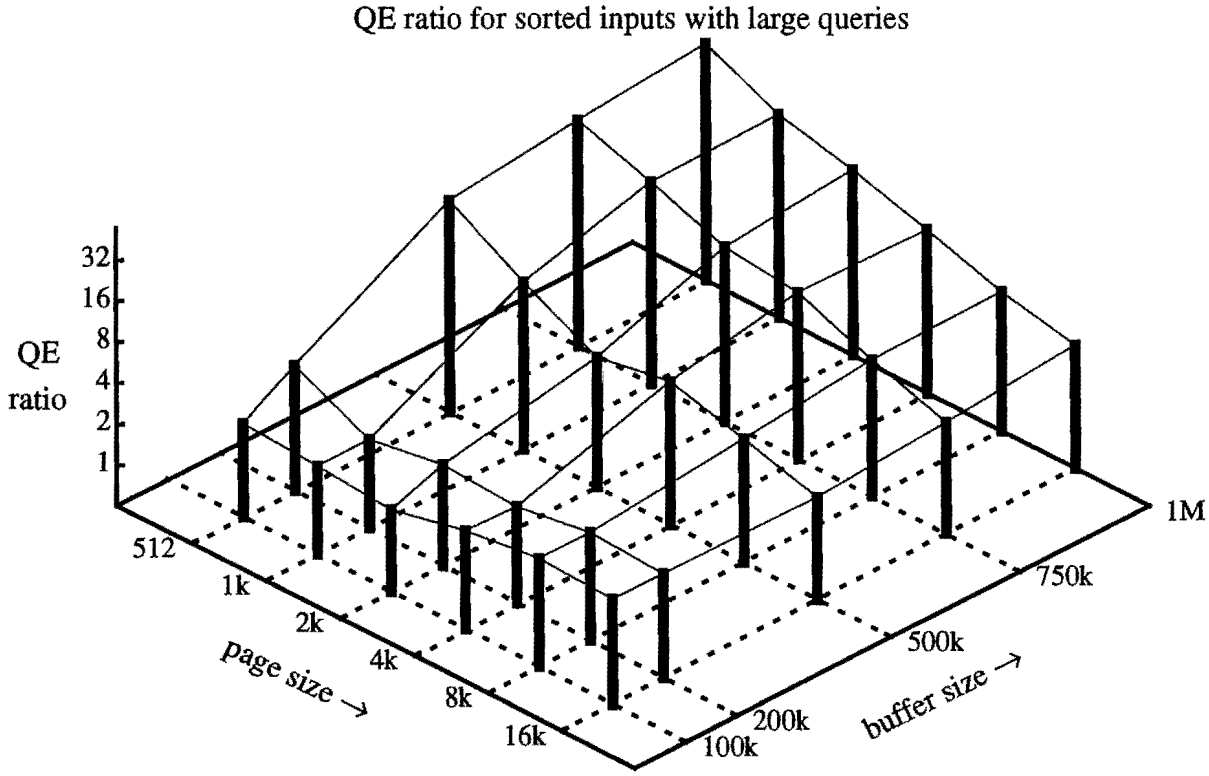


Figure 7.3: Plot of QE ratio against page size and buffer pool size. This experiment, on hierarchy H1, was conducted with sorted input to the B-tree code (leading to better space utilization), large queries were used, and each class had 20,000 objects.

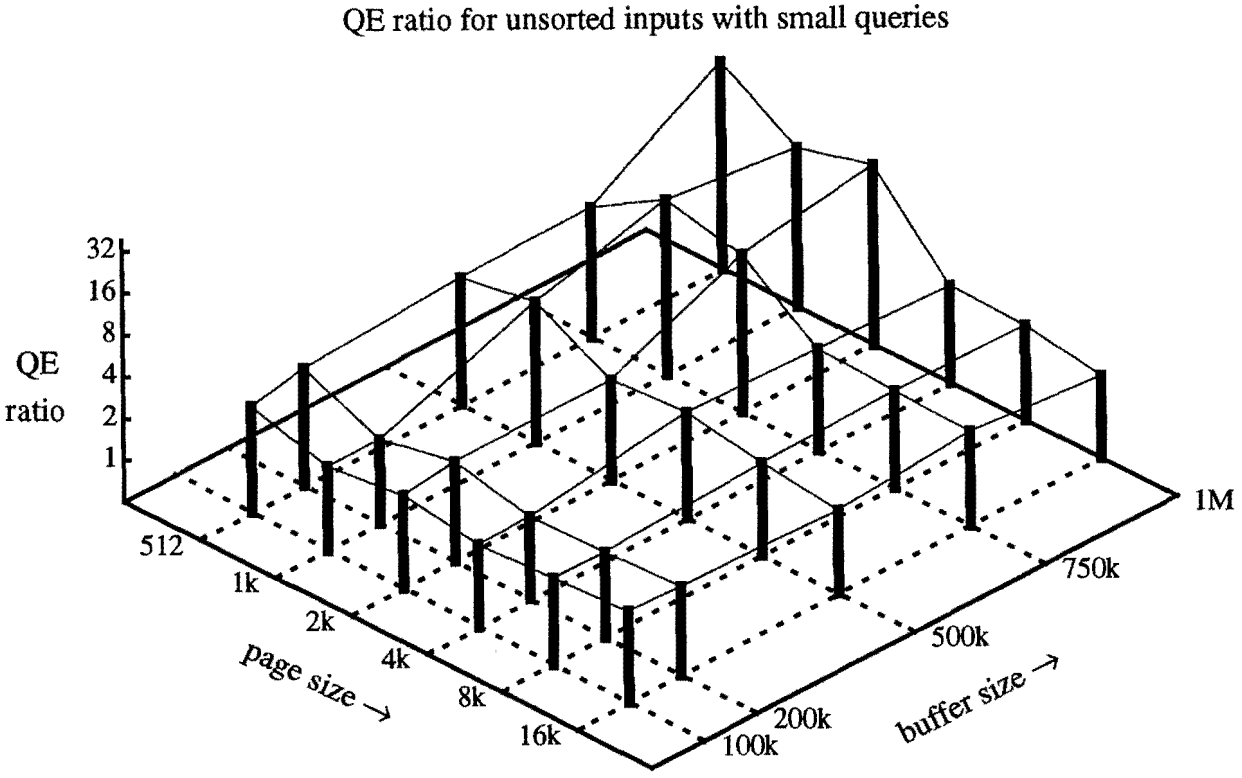


Figure 7.4: Plot of QE ratio against page size and buffer pool size. This experiment, on hierarchy H1, was conducted with unsorted input, queries usually retrieved only a small portion of the database, and each class had 20,000 objects.

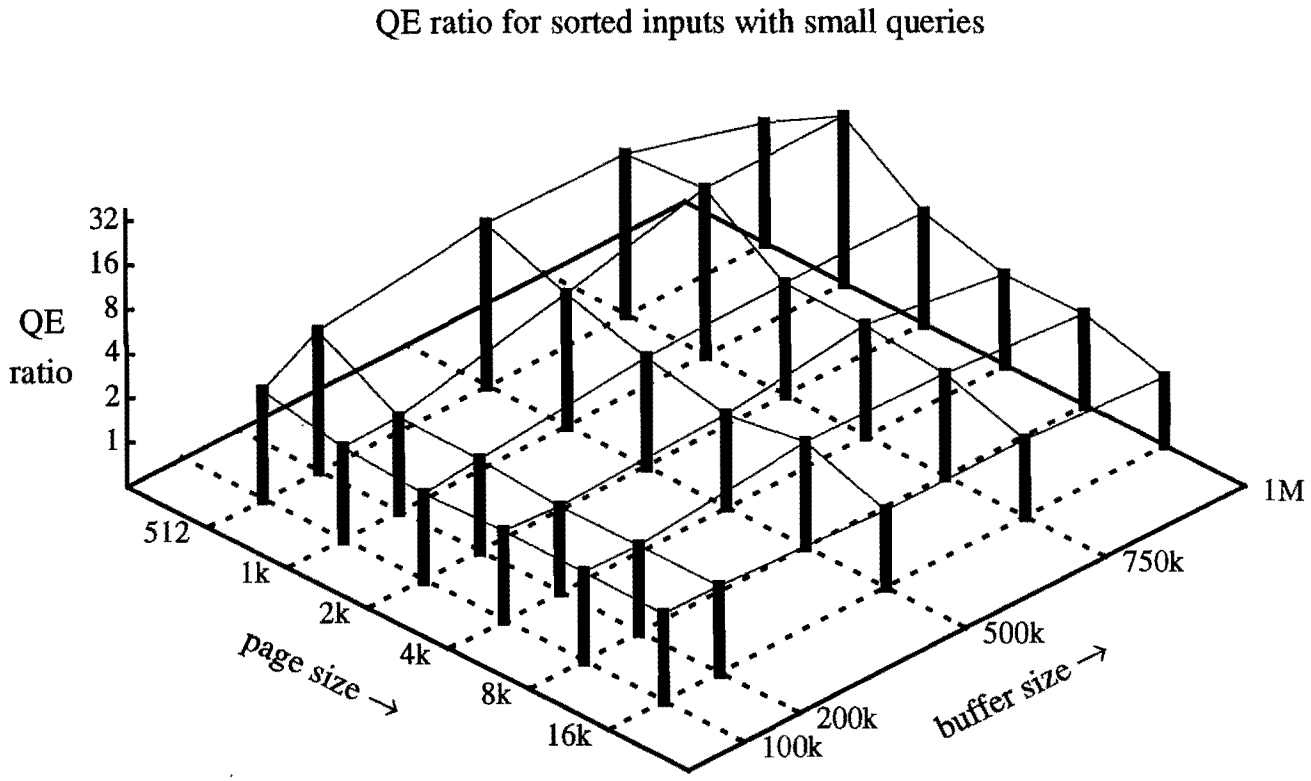


Figure 7.5: Plot of QE ratio against page size and buffer pool size. This experiment, on hierarchy H1, was conducted with sorted input, queries usually retrieved only a small portion of the database, and each class had 20,000 objects.

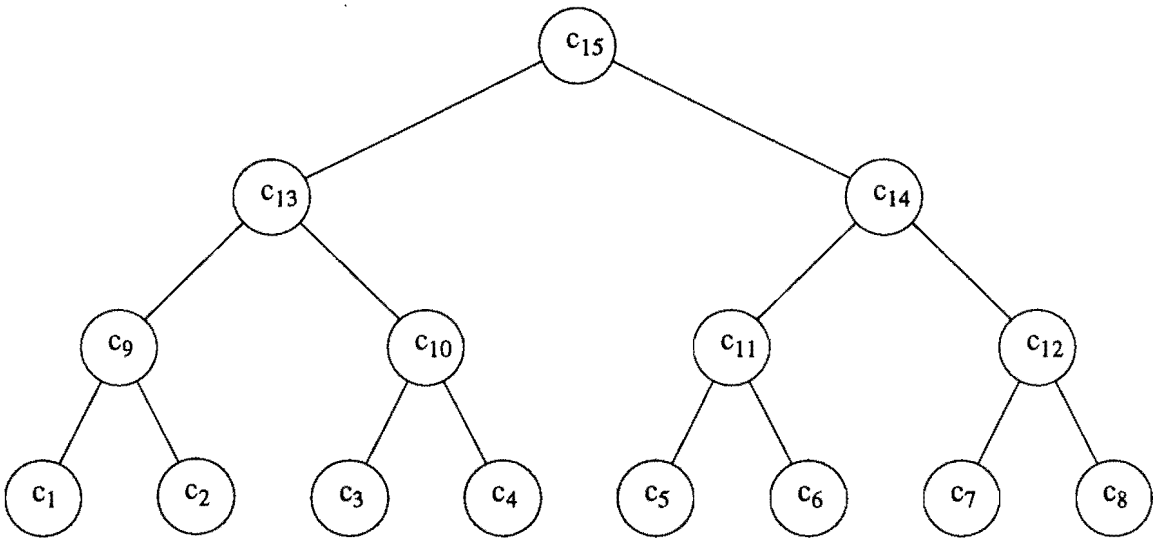


Figure 7.6: Hierarchy H2. For this hierarchy, the storage overhead factor is 3.27, the maximal replication factor 4 and the maximal query factor 1.

We now present the remaining five class hierarchies used, in addition to hierarchy H1, in the second set of experiments.

Example 7.3.2 Hierarchy H2 (in Figure 7.6) is a complete binary tree of height 3. $c_1, c_2, \ldots, c_8$ are indexed first using their individual extents. C_9 is obtained by combining c_1, c_2 and c_9, C_{10} by combining c_3, c_4 and c_{10}, C_{11} by combining c_5, c_6 and c_{11}, and C_{12} by combining c_7, c_8 and c_{12}. C_{13} is obtained by combining C_9, C_{10} and c_{13}, and C_{14} by combining C_{11}, C_{12} and c_{14}. Finally, C_0 (C_{15}) is obtained by combining C_{13}, C_{14} and c_{15}. $C_9, C_{10}, \ldots, C_{15}$ are then indexed.

The storage overhead factor for this example can easily be shown to be

$$(4\Sigma_1^8 c_i + 3\Sigma_9^{12} c_i + 2c_{13} + 2c_{14} + c_{15})/\Sigma_1^{15} c_i.$$

As before, if we assume that all the individual extents are of the same size, the storage overhead factor is $49/15 = 3.27$. In this example, the full extent of every class is indexed as a separate collection, and therefore the maximal query factor is 1. The maximal replication factor is 4.□

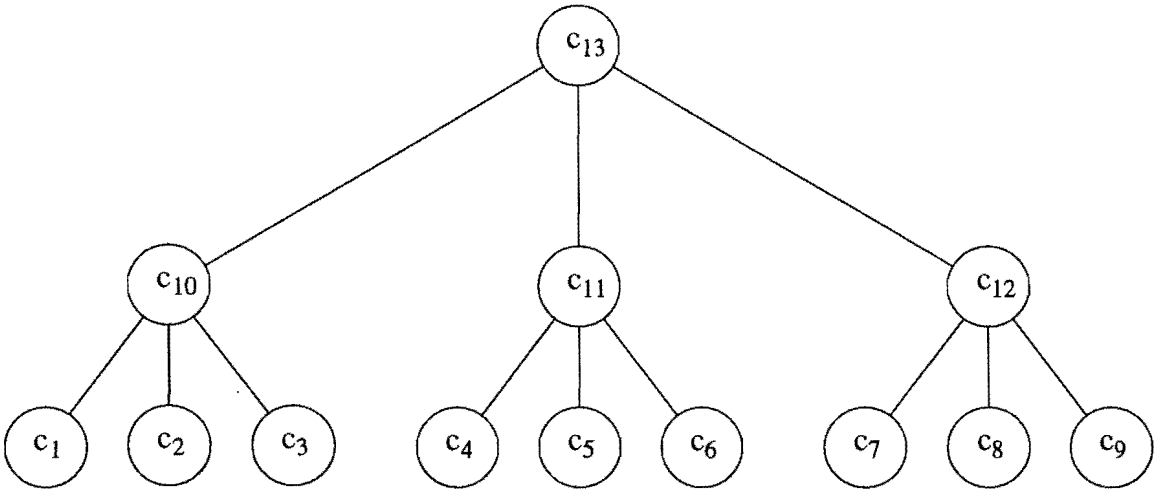


Figure 7.7: Hierarchy H3. For this hierarchy, the storage overhead factor is 2.62, the maximal replication factor 3 and the maximal query factor 1.

Example 7.3.3 Hierarchy H3 (in Figure 7.7) is a complete ternary hierarchy of height 2. Like in hierarchy H2, $c_1, c_2, \ldots, c_9$ are indexed first using their individual extents. C_{10} is created by combining c_1, c_2, c_3 and c_{10}. C_{11} and C_{12} are created in a similar fashion. Finally, C_0 (C_{13}) is created by combining C_{10}, C_{11}, C_{12} and c_{13}. $C_{10}, \ldots, C_{13}$ are then indexed.

The storage overhead factor for this example, making the usual assumption of equal individual extents, is

$$(3\Sigma_1^9 c_i + 2\Sigma_{10}^{12} c_i + c_{13})/\Sigma_1^{13} c_i.$$

This equals $34/13 = 2.62$. The maximal query factor is 1 since the full extent of every class is indexed as a separate collection. The maximal replication factor is 3.□

Example 7.3.4 Hierarchy H4 (in Figure 7.8) combines the "long, skinny" nature of hierarchy H1 and the "bushy" nature of hierarchy H2. $c_1, c_2, \ldots, c_6$ are indexed separately, and after this C_7, C_8, C_9 and C_0 (C_{10}) are created in the usual manner and indexed. By doing this, every class other than c_6 has its full extent indexed. Queries on c_6 are thus answered by looking at the indexes on c_5 and c_6.

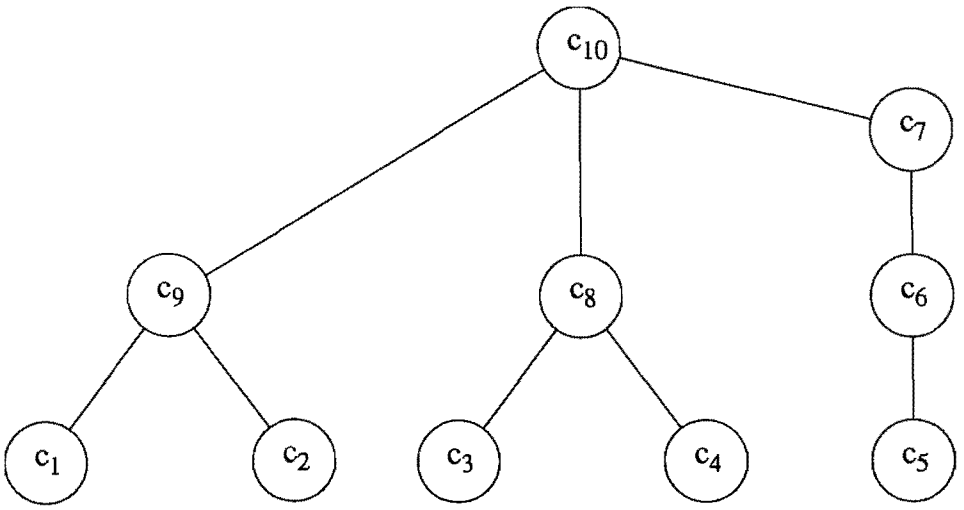


Figure 7.8: Hierarchy H4. For this hierarchy, the storage overhead factor is 2.5, the maximal replication factor 3 and the maximal query factor 2.

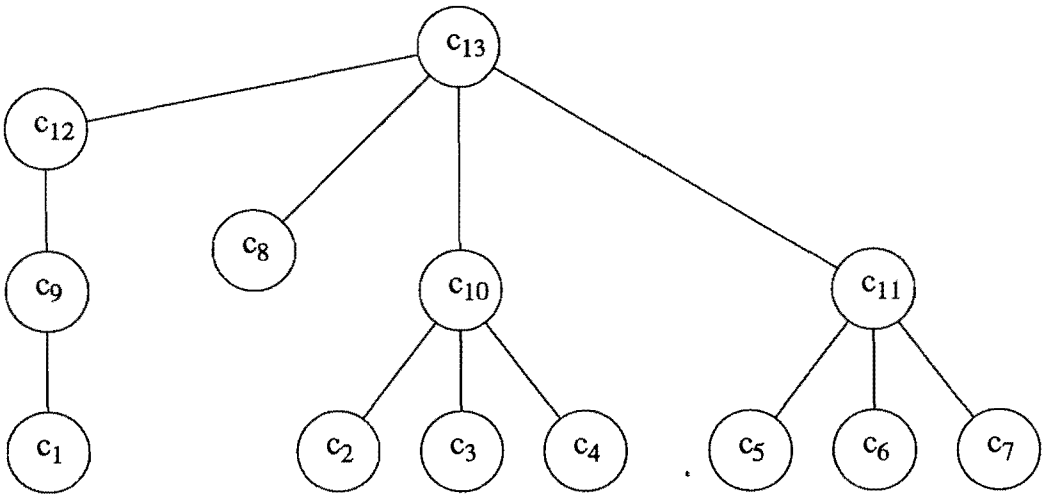


Figure 7.9: Hierarchy H5. For this hierarchy, the storage overhead factor is 2.54, the maximal replication factor 3 and the maximal query factor 2.

The storage overhead factor is

$$(3\Sigma_1^6 c_i + 2\Sigma_7^9 c_i + c_{10})/\Sigma_1^{10} c_i.$$

This equals $25/10 = 2.5$ with the usual assumption of individual extents of equal size. The maximal query factor is 2, and the maximal replication factor is 3.□

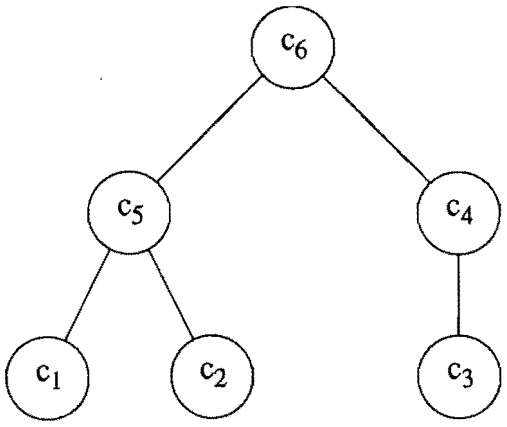


Figure 7.10: Hierarchy H6. For this hierarchy, the storage overhead factor is 2.17, the maximal replication factor 3 and the maximal query factor 2.

Example 7.3.5 Hierarchy H5 (in Figure 7.9) is similar to H4 in its combination of "skinny" and "bushy" hierarchies. As in the previous example, $c_1, c_2, \ldots, c_8$ are indexed separately. This is followed by the creation and indexing of C_{10}, C_{11}, C_{12} and C_0 (C_{13}). All classes except c_9 have their full extents indexed as a separate collection. Queries on c_9 are answered using the indexes on c_1 and c_9.

The storage overhead factor is

$$(3\Sigma_1^7 c_i + 2c_8 + 3c_9 + 2c_{10} + 2c_{11} + 2c_{12} + c_{13})/\Sigma_1^{13} c_i.$$

This equals $33/13 = 2.54$ with the assumption of individual extents of equal size. The maximal query factor is 2 and the maximal replication factor is 3.□

Example 7.3.6 Hierarchy H6 (in Figure 7.10) is a small hierarchy of 6 classes. c_1, c_2, c_3 and c_4 are indexed first. This is followed by the creation and indexing of C_5 and C_6. Under this scheme, every class except c_4 has its complete extent indexed. Queries on c_4 are answered using the indexes on c_3 and c_4.

The storage overhead factor is

$$(3\Sigma_1^2 c_i + 2\Sigma_3^5 c_i + c_6)/\Sigma_1^6 c_i.$$

This equals 13/6 = 2.17 with the assumption of equal individual extents. The maximal query factor is 2 and the maximal replication factor 3.□

We can see from these examples that with relatively small overheads in space, it is possible to index class hierarchies so that queries on them can be answered extremely efficiently. Figures 7.11–7.16 present a series of six graphs showing the results of experiments on the six class hierarchies H1–H6 presented above. These graphs plot the QE ratio on a per class basis. (That is, the ratio is averaged over each class separately rather than over the entire hierarchy.) The average over all the classes is presented as the QE ratio for class value zero. In these experiments, the size of each individual class extent was set to 10,000 objects, the page size was set to 4096 bytes, and the buffer pool size was set to 500 kilobytes. We generated a set of 200 queries for each run and conducted the experiments. We plot four separate lines on each graph to get an understanding of the effect of sorted input and query size on the QE ratio.

As before, we can make the general observation that unsorted input and large queries lead to somewhat higher QE ratios. The QE ratio tends to be very high for the classes at the bottom of the hierarchy and tends to come down as we go up the class hierarchy. It usually is very close to 1 for the root hierarchy. All this is to be expected because the single-index method performs the worst for classes low in the hierarchy: in the B^+-tree this method uses, objects in a "low" class are interspersed with many objects from other classes but none of the other objects are needed for queries asked on the "low" class. The class-divisions algorithm avoids this problem by keeping separate indexes. The QE ratio is close to 1 for the root hierarchy because the two methods are querying the same B^+-tree to answer queries on the root class. Slight deviations arise because of the action of the buffer pool.

On the whole, the QE ratio obtained tends to be many times the storage overhead factor, proving the efficacy of trading space for better query times.

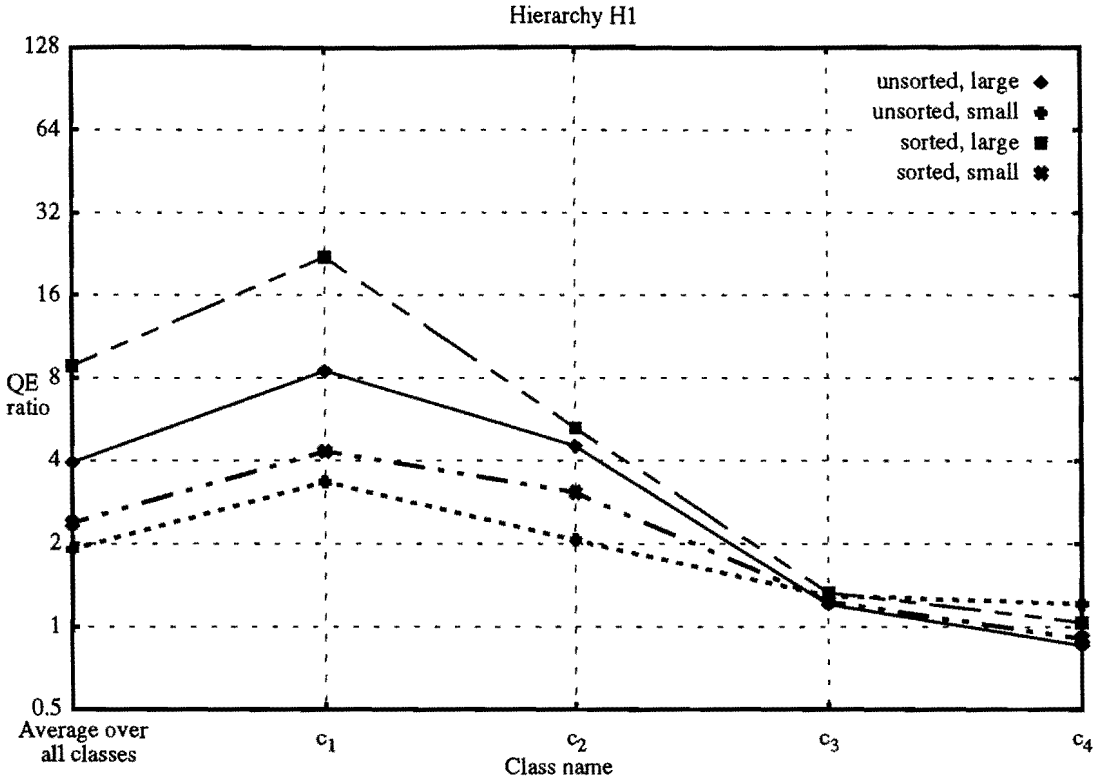


Figure 7.11: Plot of QE ratio for the classes in hierarchy H1. (The average taken over all the classes is shown on the extreme left.) Each class had 10,000 objects, the page size was 4 kilobytes, and the buffer pool size was 500 kilobytes.

7.4 The Class-Divisions Algorithm: Conclusions

The results of our experiments clearly show the superior performance of the class-divisions method over the single-index method. As the size of the class hierarchy increases, the storage overhead of the class-divisions method increases relatively slowly as a logarithmic function. The QE ratio, however, seems to increase almost linearly with class size. For example, for hierarchies H2 and H3, the QE ratio is well over 8 when averaged over all the queries, and some individual classes have QE ratios of over 64. This means that the single-index method can take 64 times as long as the class-divisions method for some queries. That these results can be obtained with uniformly distributed and extremely well behaved data is further proof of the robustness of the method.

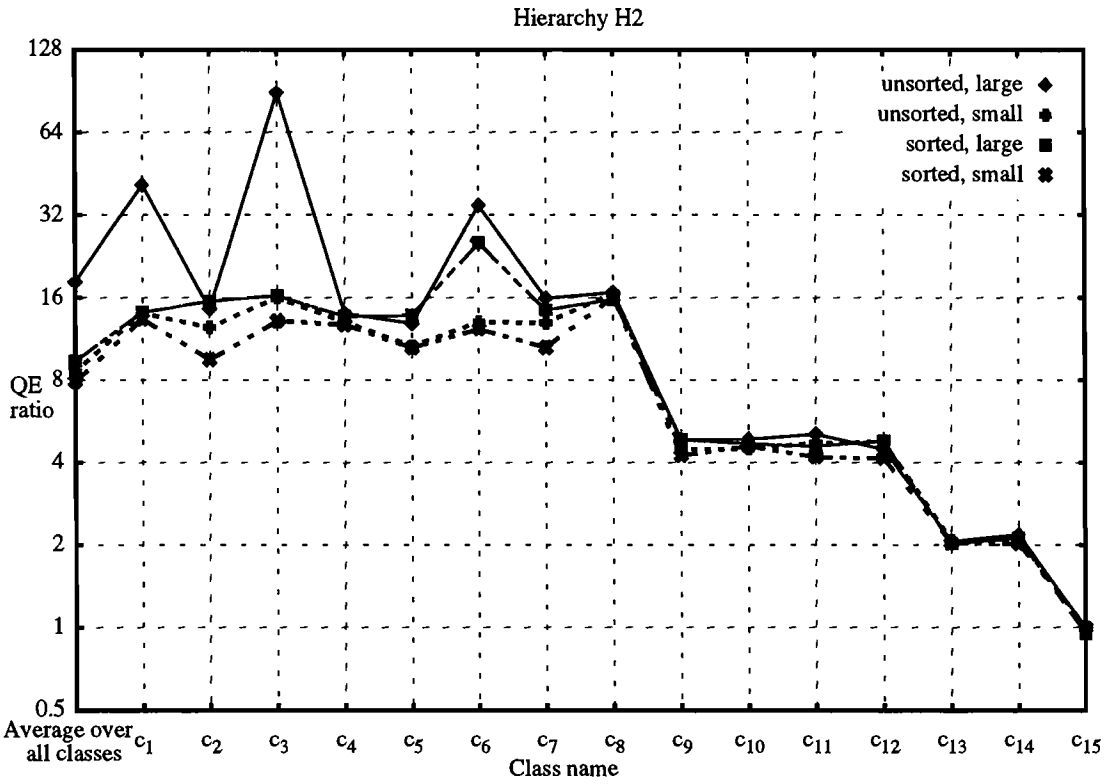


Figure 7.12: Plot of QE ratio for the classes in hierarchy H2. (The average taken over all the classes is shown on the extreme left.) Each class had 10,000 objects, the page size was 4 kilobytes, and the buffer pool size was 500 kilobytes.

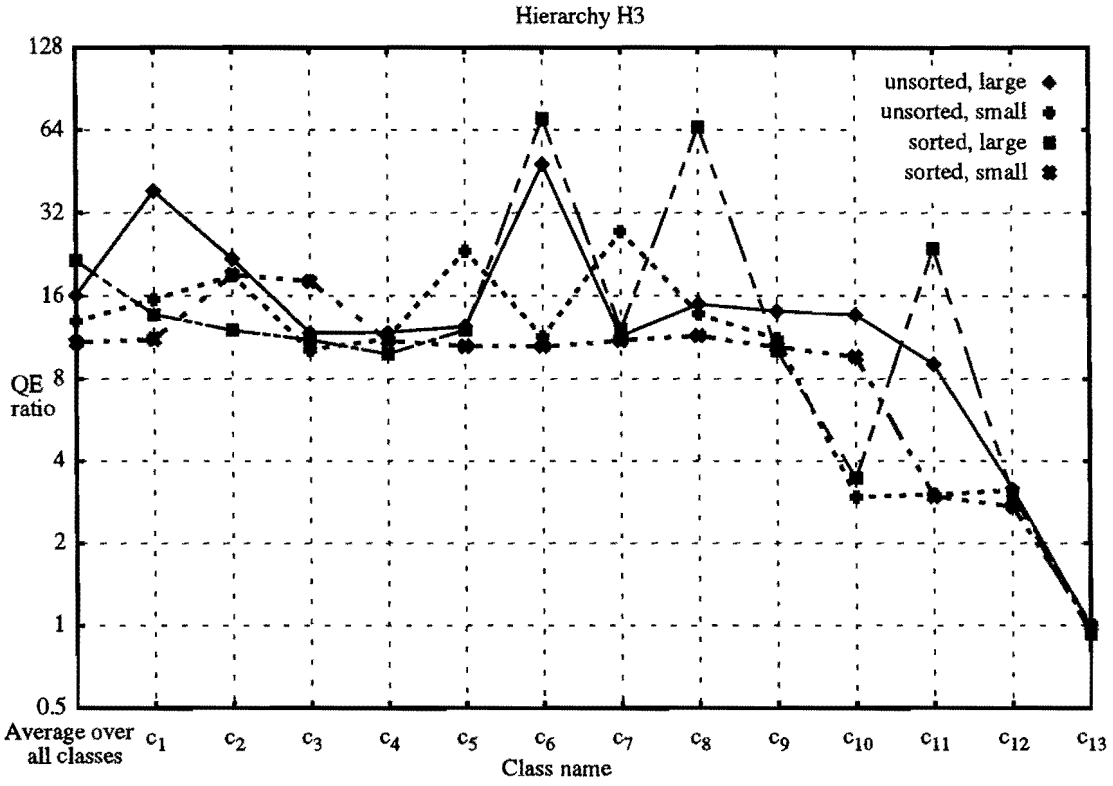


Figure 7.13: Plot of QE ratio for the classes in hierarchy H3. (The average taken over all the classes is shown on the extreme left.) Each class had 10,000 objects, the page size was 4 kilobytes, and the buffer pool size was 500 kilobytes.

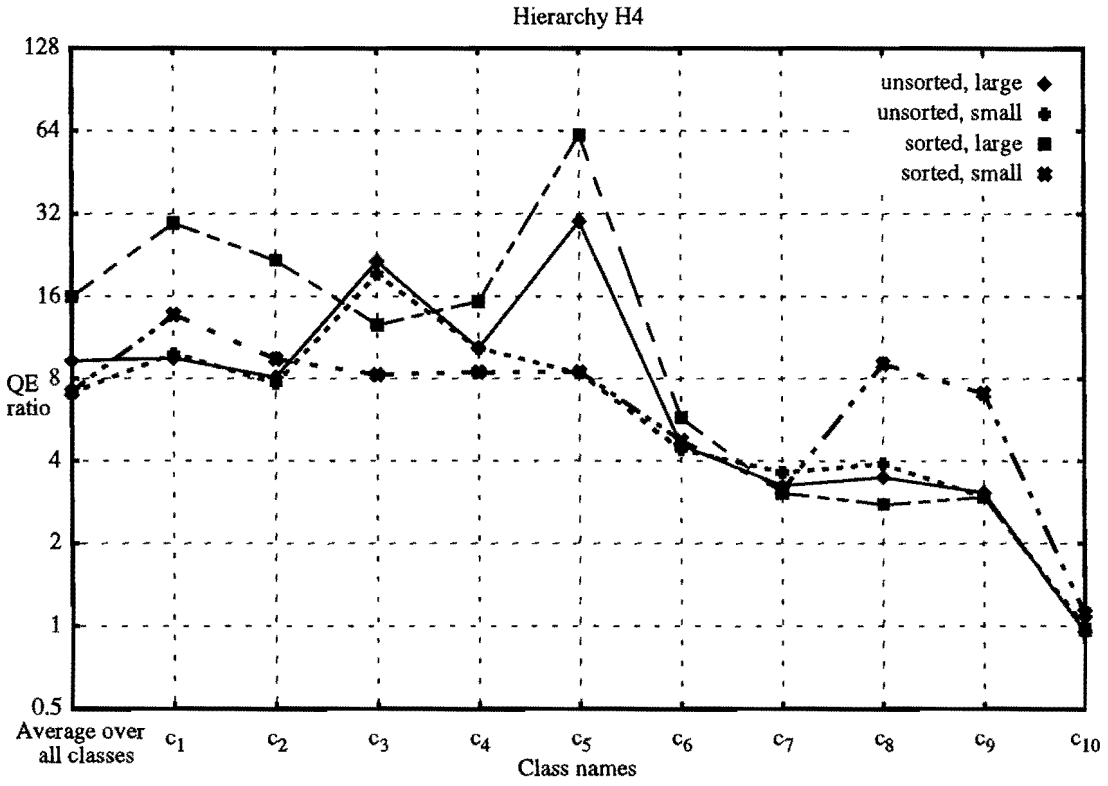


Figure 7.14: Plot of QE ratio for the classes in hierarchy H4. (The average taken over all the classes is shown on the extreme left.) Each class had 10,000 objects, the page size was 4 kilobytes, and the buffer pool size was 500 kilobytes.

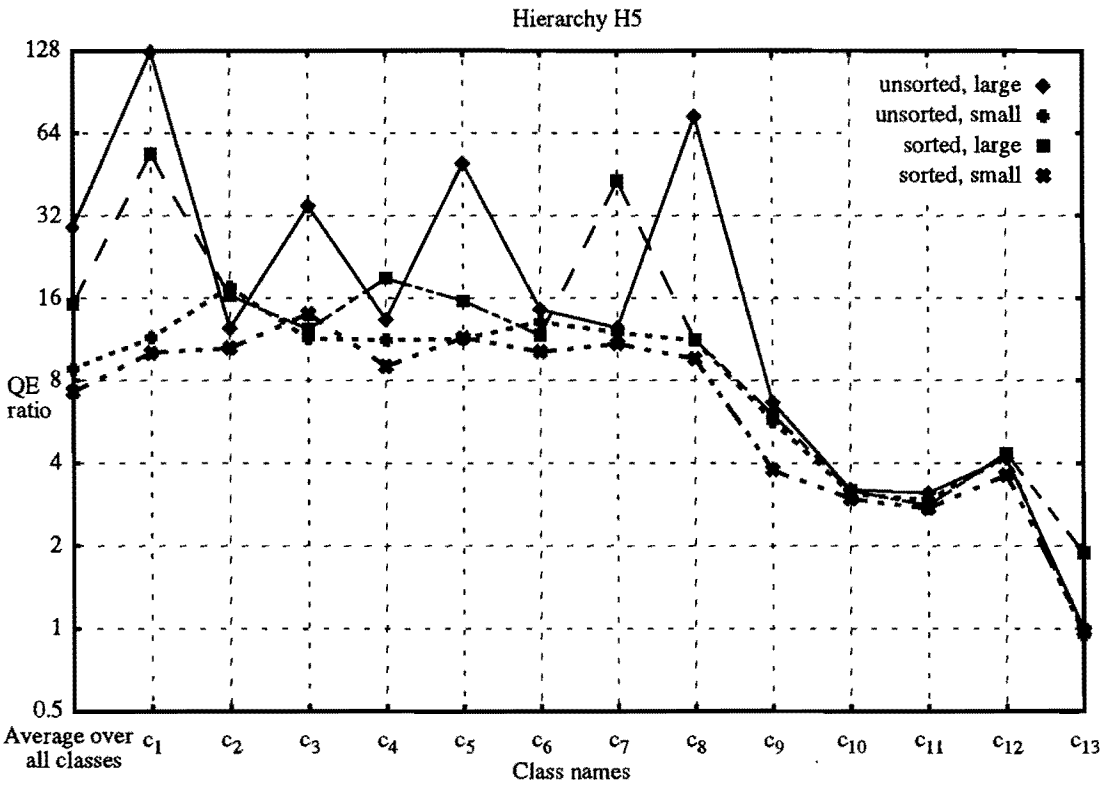


Figure 7.15: Plot of QE ratio for the classes in hierarchy H5. (The average taken over all the classes is shown on the extreme left.) Each class had 10,000 objects, the page size was 4 kilobytes, and the buffer pool size was 500 kilobytes.

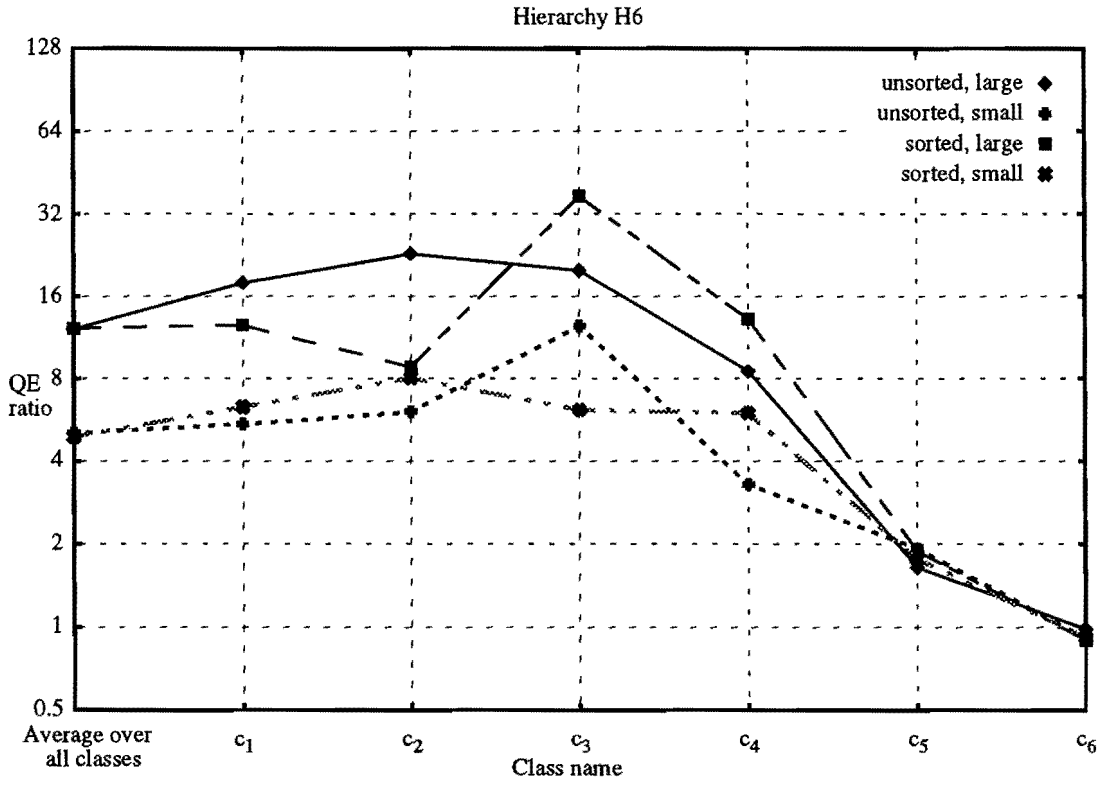


Figure 7.16: Plot of QE ratio for the classes in hierarchy H6. (The average taken over all the classes is shown on the extreme left.) Each class had 10,000 objects, the page size was 4 kilobytes, and the buffer pool size was 500 kilobytes.

Chapter 8

Conclusions and Open Problems

We have examined I/O-efficient data structures which provide indexing support for data models with constraint programming and object-oriented programming features. There are two components to this thesis: (1) to obtain algorithms with good worst-case performance (in an asymptotic sense), and (2) to obtain algorithms with good worst-case performance that also perform well in practice.

We believe that this thesis has successfully demonstrated both the components. We have proved that it is possible to devise data structures that have provably good worst-case bounds on storage, query, and update times. By doing so, we have examined important tradeoffs between space and query efficiency for special cases of two-dimensional searching.

By devising algorithms with good worst-case performance bounds that also perform well in practical settings, we have demonstrated that we need not always rely on heuristics to handle difficult indexing problems.

There are some questions that still elude a good answer. One of the more important questions concerns our ability to solve the interval management problem well. Can dynamic interval management on secondary storage be achieved optimally in $O(n/B)$ pages, query

I/O time $O(\log_B n + t/B)$ and update time $O(\log_B n)$? In other words, is there a B-tree for this problem?

Chapter 9

References

[1] F. Bancilhon, C. Delobel, and P. Kanellakis, eds., *Building an Object-Oriented Database System – The Story of O_2*, Morgan Kaufmann Publishers, 1992.

[2] R. Bayer and E. McCreight, "Organization of Large Ordered Indexes," *Acta Informatica* 1(1972), 173–189.

[3] J. L. Bentley, "Algorithms for Klee's Rectangle Problems," Dept. of Computer Science, Carnegie Mellon Univ., unpublished notes, 1977.

[4] J. L. Bentley, "Multidimensional Divide and Conquer," *CACM* 23(6)(1980), 214–229.

[5] G. Blankenagel and R. H. Güting, "XP-Trees – External Priority Search Trees," FernUniversität Hagen, Informatik–Bericht Nr. 92, 1990.

[6] G. Blankenagel and R. H. Güting, "External Segment Trees," FernUniversität Hagen, Informatik–Bericht, 1990.

[7] B. Chazelle, "Lower Bounds for Orthogonal Range Searching: I. The Reporting Case," *J. ACM* 37(2)(1990), 200–212.

[8] B. Chazelle and L. J. Guibas, "Fractional Cascading: I. A Data Structuring Technique," *Algorithmica* 1(1986), 133–162.

[9] Y.-J. Chiang and R. Tamassia, "Dynamic Algorithms in Computational Geometry," *Proceedings of IEEE, Special Issue on Computational Geometry* 80(9)(1992), 362–381.

[10] E. F. Codd, "A Relational Model for Large Shared Data Banks," *CACM* 13(6)(1970), 377–387.

[11] D. Comer, "The Ubiquitous B-tree," *Computing Surveys* 11(2)(1979), 121–137.

[12] H. Edelsbrunner, "A New Approach to Rectangle Intersections, Part II," *Int. J. Computer Mathematics* 13(1983), 221–229.

[13] H. Edelsbrunner, "A New Approach to Rectangle Intersections, Part I," *Int. J. Computer Mathematics* 13(1983), 209–219.

[14] M. L. Fredman, "A Lower Bound on the Complexity of Orthogonal Range Queries," *J. ACM* 28(1981), 696–705.

[15] M. T. Goodrich, J.-J. Tsay, D. E. Vengroff, and J. S. Vitter, "External-Memory Computational Geometry," *Proc. 34th Annual IEEE Symposium on Foundations of Computer Science*(1993), 714–723.

[16] O. Günther, "The Design of the Cell Tree: An Object-Oriented Index Structure for Geometric Databases," *Proc. of the Fifth Int. Conf. on Data Engineering*(1989), 598–605.

[17] Antonin Guttman, "R-Trees: A Dynamic Index Structure for Spatial Searching," *Proc. 1984 ACM-SIGMOD Conference on Management of Data*(1985), 47–57.

[18] C. Icking, R. Klein, and T. Ottmann, *Priority Search Trees in Secondary Memory (Extended Abstract)*, Lecture Notes In Computer Science #314, Springer-Verlag, 1988.

[19] J. Jaffar and J. L. Lassez, "Constraint Logic Programming," *Proc. 14th ACM POPL* (1987), 111–119.

[20] P. C. Kanellakis, G. M. Kuper, and P. Z. Revesz, "Constraint Query Languages," *Proc. 9th ACM PODS* (1990), 299–313, invited to the special issue of *JCSS* on Principles of Database Systems (to appear). A complete version of the paper appears as Technical Report 90-31, Brown University.

[21] P. C. Kanellakis, S. Ramaswamy, D. E. Vengroff, and J. S. Vitter, "Indexing for Data Models with Constraints and Classes," *Proc. 12th ACM PODS* (1993), 233–243, invited to the special issue of *JCSS* on Principles of Database Systems (to appear). A complete version of the paper appears as Technical Report 93-21, Brown University.

[22] W. Kim, K. C. Kim, and A. Dale, "Indexing Techniques for Object-Oriented Databases," in *Object-Oriented Concepts, Databases, and Applications*, W. Kim and F. H. Lochovsky, eds., Addison-Wesley, 1989, 371–394.

[23] W. Kim and F. H. Lochovsky, eds., *Object-Oriented Concepts, Databases, and Applications*, Addison-Wesley, 1989.

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

[25] C. C. Low, B. C. Ooi, and H. Lu, "H-trees: A Dynamic Associative Search Index for OODB," *Proc. ACM SIGMOD* (1992), 134–143.

[26] D. Maier and J. Stein, "Indexing in an Object-Oriented DBMS," *IEEE Proc. International Workshop on Object-Oriented Database Systems* (1986), 171–182.

[27] E. M. McCreight, "Priority Search Trees," *SIAM Journal of Computing* 14(2)(1985), 257–276.

[28] J. Nievergelt, H. Hinterberger, and K. C. Sevcik, "The Grid File: An Adaptable, Symmetric Multikey File Structure," *ACM Transactions on Database Systems* 9(1)(1984), 38–71.

[29] J. A. Orenstein, "Spatial Query Processing in an Object-Oriented Database System," *Proc. ACM SIGMOD* (1986), 326–336.

[30] M. H. Overmars, M. H. M. Smid, M. T. de Berg, and M. J. van Kreveld, "Maintaining Range Trees in Secondary Memory: Part I: Partitions," *Acta Informatica* 27 (1990), 423–452.

[31] S. Ramaswamy and S. Subramanian, "Path Caching: A Technique for Optimal External Searching," *Proc. 13th ACM PODS* (1994), 25–35.

[32] J. T. Robinson, "The K-D-B Tree: A Search Structure for Large Multidimensional Dynamic Indexes," *Proc. ACM SIGMOD* (1984).

[33] H. Samet, *Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS*, Addison-Wesley, 1989.

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

[35] T. Sellis, N. Roussopoulos, and C. Faloutsos, "The R^+-Tree: A Dynamic Index for Multi-Dimensional Objects," *Proc. 1987 VLDB Conference, Brighton, England* (1987).

[36] M. Seltzer, K. Bostic, and O. Yigit, *The Berkeley DB code*, available by ftp from ftp.cs.berkeley.edu as ucb/4bsd/db.tar.Z .

[37] D. D. Sleator and R. E. Tarjan, "A Data Structure for Dynamic Trees," *J. Computer and System Sciences* 24 (1983), 362–381.

[38] M. H. M. Smid and M. H. Overmars, "Maintaining Range Trees in Secondary Memory: Part II: Lower Bounds," *Acta Informatica* 27 (1990), 453–480.

[39] M. Stonebraker and L. Rowe, "The Design of POSTGRES," *Proc. ACM SIGMOD* (1986).

[40] J. S. Vitter, "Efficient Memory Access in Large-Scale Computation," *1991 Symposium on Theoretical Aspects of Computer Science (STACS), Lecture Notes in Computer Science* (1991), invited paper.

[41] S. Zdonik and D. Maier, *Readings in Object-Oriented Database Systems*, Morgan Kaufmann, 1990.