Indexing for Data Models with Constraints and Classes

Paris Kanellakis * Sridhar Ramaswamy† Darren Vengroff‡ Jeffrey Vitter§

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). Let n be the size of the database, c the number of classes, B the page size on secondary storage, and t the size of the output of a query. (1) Indexing by one attribute in many constraint data models is equivalent to external dynamic interval management, which is a special case of external dynamic 2-dimensional range searching. We present a semi-dynamic data structure for this problem that has worst-case space $O(n/B)$ pages, query I/O time $O(\log_B n + t/B)$ and $O(\log_B n + (\log_B n)^2/B)$ amortized insert I/O time. Note that, for the static version of this problem, this is the first worst-case optimal solution. (2) Indexing by one attribute and by class name in an object-oriented model, where objects are organized as a forest hierarchy of classes, is also a special case of external dynamic 2-dimensional range searching. Based on this observation, we first identify a simple algorithm with good worst-case performance, query I/O time $O(\log_2 c \log_B n + t/B)$, update I/O time $O(\log_2 c \log_B n)$ and space $O((n/B) \log_2 c)$ pages for the class indexing problem. Using the forest structure of the class hierarchy and techniques from the constraint indexing problem, we improve its query I/O time to $O(\log_B n + t/B + \log_2 B)$.

Keywords: constraint and object-oriented databases, indexing, external dynamic k-dimensional range searching, dynamic interval management.

*Address: Dept. of Computer Science, Brown University, Box 1910, Providence, RI 02912. Email: `pck@cs.brown.edu`. Research supported by ONR Contract N00014–91–J–4052, ARPA Order 8225.

†*Contact Author.* Current address: Bell Communications Research, 445 South Street #2D332, Morristown NJ 07960. Research conducted while this author was at the Dept. of Computer Science, Brown University, Providence RI 02912, and supported by ONR Contract N00014–91–J–4052, ARPA Order 8225. Email: `sridhar@bellcore.com`.

‡Address: Dept. of Computer Science, Duke University, Box 90129, Durham, NC 27708–0129. Email: `dev@cs.duke.edu`. Also affiliated with Brown University. Support was provided in part by National Science Foundation research grant CCR–9007851 and by Army Research Office grant DAAL03–91–G–0035.

§Address: Dept. of Computer Science, Duke University, Box 90129, Durham, NC 27708–0129. Email: `jsv@cs.duke.edu`. Support was provided in part by National Science Foundation research grant CCR–9007851 and by Army Research Office grant DAAL03–91–G–0035.

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 [9] includes declarative programming in the form of relational calculus and algebra and expresses queries of low data complexity because every fixed relational calculus query is evaluable 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 [1,10] are examples of such data structures. They have been an unqualified success in supporting *external dynamic 1-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 1-dimensional searching have not been found for even the simplest cases of external k-dimensional range searching. In this paper, 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 [19] and class hierarchies in object-oriented databases [21,38].

We make the standard assumption that each secondary memory access transmits one page or B units of data (typically bytes), and we count this as one I/O. (We will use the terms page and disk block interchangeably, as also the terms in-core and main memory.) We also assume that at least $O(B^2)$ units of main memory are available. This is not an assumption that is normally made, but it 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 will 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 will be measured in terms of the number of I/O's that they need for querying and updating and the number of disk blocks they require for storage. The I/O bounds will be expressed in terms of n, c, t and B, i.e., all constants will be independent of these four parameters. (For a survey of state of the art I/O complexity, see [37].) We will first review B$^+$-tree performance since we will 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 1-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 is in worst-case $O(\log_B n + t/B)$ secondary memory accesses. If $a_1 = a_2$ and x is a key, i.e., it uniquely identifies the tuple, then this is key-based searching. (2) Insert or delete a given tuple are in worst-case $O(\log_B n)$ secondary memory accesses. The problem of *external dynamic k-dimensional range searching* on relational database attributes

$x_1, \ldots, x_k$ generalizes 1-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 paper we will be concerned with external 2-dimensional range searching: dynamic (Section 2), semi-dynamic and static (Sections 3 and 4). We are concerned with algorithms which have provably good worst-case I/O bounds.

In order 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 2-dimensional range searching, and thus the problems examined here, can be solved using worst-case $O((n/B)\log_2 n)$ pages, static query I/O time $O(\log_2 n + t/B)$ using fractional cascading, dynamic query I/O time $O(\log_2 n \log_B n + t/B)$, and amortized update I/O time $O(\log_2 n \log_B n)$. (Note that $\log_2 n = (\log_2 B)(\log_B n)$ is asymptotically much larger than $\log_B n$.) Based on the special structure of the indexing problems of interest we improve on the above bounds.

1.2 Contributions to 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 [19]. This framework adapts ideas of Constraint Logic Programming, e.g., from [18], 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 2.1 for a detailed exposition of the problem). Fortunately, it is possible to do this by combining CQLs with existing 2-dimensional range searching data structures [19]. The basis of this observation 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, to report all the input intervals which 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 2-dimensional range searching.

Dynamic interval management is interesting because it can be solved optimally in-core using the Priority Search Tree of McCreight [25] 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 2.1, we reduce indexing constraints to a special case of external dynamic 2-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.) In Section 3, 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).

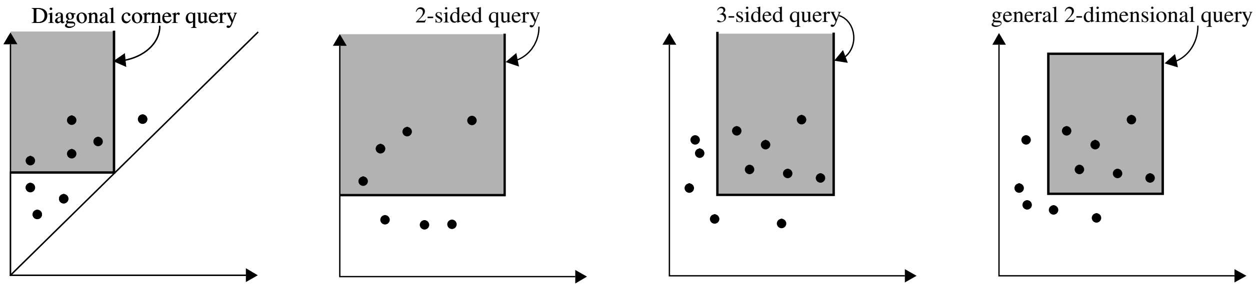


Figure 1: Diagonal corner queries, 2-sided, 3-sided and general 2-dimensional range queries.

1.3 Contributions to 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 2-dimensional range searching. Together with the different problem of indexing nested objects, as in [24], it constitutes the basis for indexing in object-oriented databases. Indexing classes has been examined in [20], and more recently in [23], but the solutions offered there are largely heuristic with poor worst-case performance.

We assume in this paper that the class-subclass relationship is static, although objects can be inserted to or deleted from classes. Under this reasonable assumption, we show in Section 2.2 that indexing classes can be reduced to a special case of external dynamic 2-dimensional range searching where one dimension is static. For a class hierarchy with c classes having a total of n objects, we identify a simple 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 2-dimensional range searching. We show in Section 2.2 that it is impossible to achieve optimal query time for this problem ($O(\log_B n + t/B)$ disk I/O's) with only one copy of each object in secondary storage. (For lower bounds on range searching in main memory, see [13] and [7].) In Section 4, analyzing the hierarchy using the hierarchical decomposition of [34], we reduce the problem of indexing classes to a special case of external dynamic 2-dimensional range searching called 3-sided range searching. 3-sided range queries are a special case of 2-dimensional range queries. In 3-sided range queries, one of the four sides defining the query rectangle is always one of the coordinate axes or at infinity[1]. Using techniques from the constraint indexing problem to solve 3-sided range searching, we improve query I/O time for the class indexing problem 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 $O(\log_2 c(\log_B n + (\log_B n)^2/B))$.

1.4 Related Research

A large literature exists for in-core algorithms for 2-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

[1]Note that diagonal corner queries are a special case of 2-sided queries and 2-sided queries are a special case of 3-sided queries. See Figure 1.

$O(\log_2 n + t)$. 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 [8] 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, [28,35] 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 1-dimensional range searching using B^+-trees. We present here a brief survey of general purpose data structures to solve external 2-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 [31,32] were designed to organize 2-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 [26] 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 [26] 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 2d-grid of points (with a point on each integral coordinate) as input, the grid file would produce regions which are of size $O(\sqrt{B}) \times O(\sqrt{B})$. Consider a query retrieving points along a horizontal straight line. The time to report answers to this 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, 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 [30] and the hB-tree [22].

k-d-B-trees combine properties of balanced k-d-trees in a B-tree-like organization. In the 2-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 [30]. 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 [16], the R^*-tree [2], the R^+-tree [33], the cell tree [15] and many others. These data structures are applicable to point data and 1-dimensional range data and hence are relevant to our problems. 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 [27]. 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 [8]). As mentioned before, the best in-core bounds have been achieved using the priority search tree of [25], 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 [11,12], 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 [17], [5], [6]. [17] contains a claimed optimal solution for implementing static priority search trees in secondary memory. Unfortunately, the [17] 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.

1.5 Overview

To summarize, we present I/O-efficient data structures—with provably good worst case bounds—that provide indexing support for new data models. Section 2.1 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 Section 3, we propose a new data structure, optimal for the static case, for this problem. Section 2.2 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 Section 4 using the techniques developed for indexing constraints. Section 5 has the conclusions and open problems.

2 The Problems and Initial Approaches

2.1 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 [19] 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 2-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.

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).

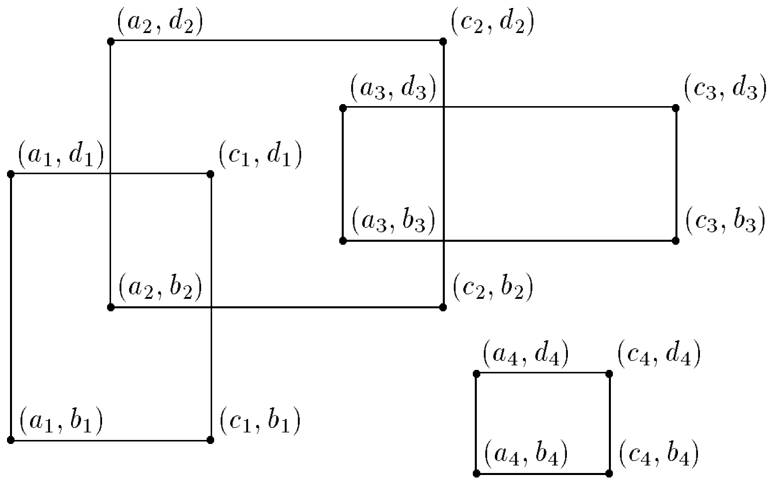


Figure 2: Rectangle intersection

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 2.1 The database consists of a set of tuples, with each tuple representing a rectangle in the 2-dimensional plane. We want to compute all pairs of distinct intersecting rectangles. (See Figure 2.)

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

$$\begin{aligned}\{(n_1, n_2) \quad | \quad & n_1 \neq n_2 \wedge (\exists a_1, a_2, b_1, b_2, c_1, c_2, d_1, d_2)(R(n_1, a_1, b_1, c_1, d_1) \wedge R(n_2, a_2, b_2, c_2, d_2) \wedge \\ & (\exists x, y \in \{a_1, a_2, b_1, b_2, c_1, c_2, d_1, d_2\}) \\ & (a_1 \leqslant x \leqslant c_1 \wedge b_1 \leqslant y \leqslant d_1 \wedge a_2 \leqslant x \leqslant c_2 \wedge b_2 \leqslant y \leqslant d_2))\}\end{aligned}$$

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 [19]. □

The CQL model for rational order + relational calculus has low data complexity, because 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 1-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 1-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 2.1, 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 1-dimensional searching on generalized database attribute x.

- A *generalized 1-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 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.

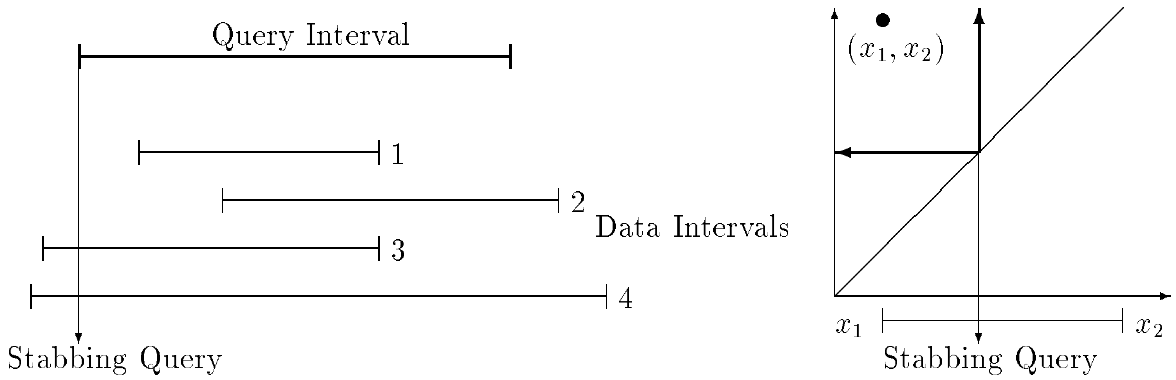


Figure 3: Reducing Interval Intersection to Stabbing Queries and Stabbing Queries to Diagonal Corner Queries

By the above discussion, the use of generalized 1-dimensional indexes reduces redundancy of representation and transforms 1-dimensional searching on generalized database attribute x into the problem of external dynamic interval management. In this paper we examine solutions for this problem with good I/O performance. Remember 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, as shown in Figure 1. We now can show the following proposition:

Proposition 2.2 Indexing constraints for convex CQLs reduces to external dynamic interval management which reduces to external dynamic 2-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. 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.) 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 2-dimensional space. Clearly, the stabbing query now translates into a 2-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 2-sided query (corresponding to a stabbing query) is anchored on this line as well. (See Figure 3.) The proposition follows. $\square$

2.2 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 2.3 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 5.) 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 1-dimensional range searching on some attribute of the objects, but for the full extent of each class in the hierarchy.*

Example 2.4 Consider the class hierarchy in Example 2.3. 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 2.4, 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 2-dimensional range searching on some attribute of the objects. We then use the idea of the 2-dimensional range tree (see [8]), 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 2.5 Indexing classes reduces to external dynamic 2-dimensional range searching with one dimension being static.

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 4: The procedure *label-class* used in the proof of Proposition 2.5.

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 4 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 2.3, 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.

It is easy to see how querying some class over some particular attribute corresponds to 2-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 5 for an example of the results of applying *label-class* to a class hierarchy. □

Theorem 2.6 *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.*

Proof: We know from the previous proposition that by treating the class value as a dimension, we can reduce indexing classes to 2-dimensional range searching with the values in the class dimension being static. We use the idea of the range tree in procedure *index-classes* shown in Figure 6 to create

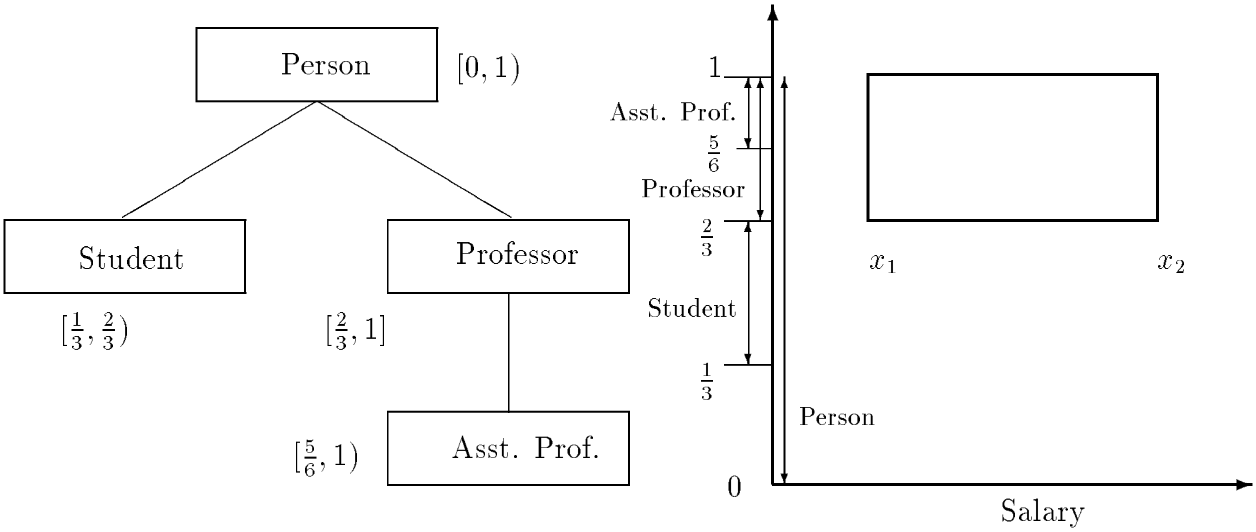


Figure 5: Using *label-class* to reduce indexing classes to 2-dimensional range search

procedure *index-classes*
  Let $C_1^1, C_2^1, \ldots, C_c^1$, the individual extents of the classes, be the initial set of collections.
  for i from 1 to $\lceil \log_2 c \rceil + 1$ **do**
    Index each collection $C_1^i, C_2^i, \ldots$ at current level
    if $i = \lceil \log_2 c \rceil + 1$ terminate
    Merge C_1^i with C_2^i to get C_1^{i+1}, C_3^i with C_4^i to get C_2^{i+1}, etc.
    If there are odd number of collections, let $C_{l_i}^i$, the last collection at round i be $C_{l_{i+1}}^{i+1}$,
      the last collection at round $i + 1$.
    /* Number of collections goes down by two at each iteration */
    /* $l_i = c/2^{i-1}$ */
  endfor

Figure 6: The procedure *index-classes* used in the proof of Theorem 2.6.

and index certain collections of objects. (Remember that indexing a collection means building a B^+-tree on the query attribute.) Our initial set of collections is the set of the individual extents of each class. Our final collection consists of the full extent of the root class. We will assume here without loss of generality that the class hierarchy is a tree. If it is a forest, we simply have to run *index-classes* on each tree in the forest. Our bounds will not change.

From the previous proposition, we know that every class query can be represented as a 2-dimensional range query. What *label-class* essentially does is to build a binary search tree on the class attribute values. It is easy to show that the range corresponding to each class can be covered by no more than $2\lceil \log_2 c \rceil$ nodes in this search tree. That is, one can answer class indexing queries

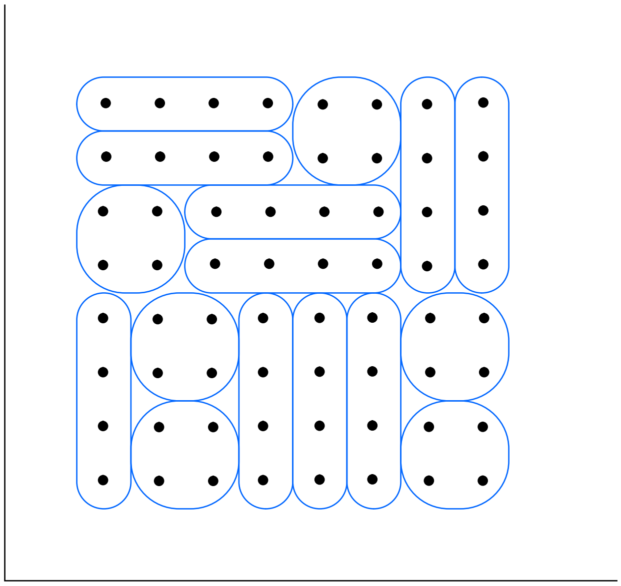


Figure 7: Showing that tessellating a grid optimally is hard.

on any class by looking at no more than $O(\log_2 c)$ collections (or B$^+$-trees). This gives us the query bound.

We establish the space and update bounds by noting that the number of levels in the class binary search tree is no more than $\lceil \log_2 c \rceil$. This automatically implies that no object is replicated more than $\lceil \log_2 c \rceil$ times and the storage bound follows. This also implies that in order to insert/delete an object, we have to access no more than $\lceil \log_2 c \rceil$ B$^+$-trees. The theorem follows. □

Theorem 2.6 offers a practical solution to class indexing. It does not have the disadvantages of indexing each class' individual extent separately (high query overhead), nor that of indexing each class' full extent separately (high storage overhead), nor that of maintaining only one index containing all the objects (loss of query efficiency). The query, update, and storage overheads are all low (a factor of $\log_2 c$) and the algorithm outlined in the proof of Theorem 2.6 is extremely simple. This makes this method an ideal choice for implementation.

Noticing the absence of data structures to perform external 2-dimensional range searching with performance comparable to that of B-trees, we might try to prove that the problem is inherently harder than external 1-dimensional range search. Such bounds have been established for range searching in main memory by [7,13]. The following lemma is a preliminary result that shows that B-tree-like performance is probably not possible for 2-dimensional range search.

We consider a simple grid of points and try to see if it is possible to place rectangles on this grid of points (i.e., tessellate the points) so that all range queries can be answered efficiently. The idea behind the lemma is that these rectangles correspond to disk blocks. We make the simplifying assumption that there is only one copy of each data item, which means that the rectangles cannot intersect. We are only concerned with reporting time in this lemma. Even if it is possible to produce partitions to answer range queries efficiently, it is far from clear that they can be located in optimal time.

Lemma 2.7 *Consider a grid of $p \times p$ points on the plane. It is not possible to tessellate these points with rectangles of size B (which do not intersect one another) such that range queries are answered in optimal time. That is, it is not possible to answer all range queries such that for a range query with q points in the output, the points are covered by only kq/B rectangles, where k is a constant independent of p, q and B.*

Proof: Figure 7 shows an example of the kinds of tessellations considered by this proof. The grid is of size 8×8, and $B = 4$.

This proof is by contradiction. Let us assume that an optimal tessellation is possible for a particular constant k. We assume without loss of generality that p is a multiple of B. Let a grid of $p \times p$ points be tessellated with rectangles of size B. There will be p^2/B rectangles tessellating the grid. We will use w_i and h_i to indicate the width and height of the i'th rectangle.

Consider range queries that retrieve p points along horizontal straight lines. There are p such queries, one for each row of the $p \times p$ grid. By our assumption, the number of rectangles the j'th such horizontal query intersects will be $k_j p/B$ for some constant $k_j \leqslant k$.

We observe that the height of each rectangle in our blocking is equal to the number of horizontal queries that intersect it. Thus, if we sum up the heights of all the rectangles we will get the same value that we get when we sum $k_j p/B$ over all j. This gives us

$$\begin{aligned} \sum_{i=1}^{p^2/B} h_i &= \sum_{j=1}^{p} k_j \frac{p}{B} \\ &\leqslant \sum_{j=1}^{p} k \frac{p}{B} \\ &= k\frac{p^2}{B}. \end{aligned} \tag{1}$$

Symmetrically, we can consider vertical queries, and get

$$\sum_{i=1}^{p^2/B} w_i \leqslant k\frac{p^2}{B}. \tag{2}$$

Recall that for each rectangle, $h_i w_i = B$. Thus the bounds on the sums of the heights of the rectangles given in (1) can be rewritten as

$$\sum_{i=1}^{p^2/B} \frac{B}{w_i} \leqslant k\frac{p^2}{B}.$$

Dividing through by B gives us

$$\sum_{i=1}^{p^2/B} \frac{1}{w_i} \leqslant k\frac{p^2}{B^2}. \tag{3}$$

Now, we use the fact that the harmonic mean of a set of numbers is at most their arithmetic mean, which tells us that

$$\frac{p^2/B}{\sum_{i=1}^{p^2/B} \frac{1}{w_i}} \leqslant \frac{\sum_{i=1}^{p^2/B} w_i}{p^2/B}.$$

If we substitute kp^2/B^2 for the denominator of the left hand side, then, by (3) the value of the left hand side will either decrease or remain the same. Thus

$$\begin{aligned} \frac{p^2/B}{kp^2/B^2} &\leqslant \frac{\sum_{i=1}^{p^2/B} w_i}{p^2/B} \\ \frac{B}{k} &\leqslant \frac{\sum_{i=1}^{p^2/B} w_i}{p^2/B} \end{aligned}$$

Now, by (2), we can replace the numerator of the right hand side by kp^2/B and it will either increase or remain the same. Thus

$$\begin{aligned} \frac{B}{k} &\leqslant \frac{kp^2/B}{p^2/B} \\ &= k \end{aligned}$$

and thus

$$B \leqslant k^2.$$

This produces our contradiction, since we were assuming that k was a constant independent of B. □

Two intriguing open questions are whether or not B-tree-like performance is possible for arbitrary partitions of the plane, not just rectangular tessellations and whether or not performance can be improved by allowing a small constant number of copies of each data item. There are data structures that attempt to obtain B-tree-like performance using one or more of these ideas. For example, the R-tree [16], the k-d-B-tree [30], and the grid file [26] use rectangular tessellations and single copies, the R^+-tree [33] uses a small number of copies and rectangular tessellations, the hB-tree [22], the cell tree [15] and the z-order [27] use non-rectangular decompositions of space. Experimental comparisons seem to indicate that none of these data structures is significantly better than the others for the different cases of range querying.

From the previous lemma, we can infer that the problem of indexing classes, despite its structure, is nontrivial. Consider a class hierarchy with c leaves all of which are children of the root. We can map the class indexing problem to 2-dimensional range searching as before. Now, we can use the previous lemma. Instead of having a grid of $p \times p$ points, we have a grid of $c \times p$ points. The following theorem then follows easily.

Theorem 2.8 *Consider the fully static problem of indexing classes, where the hierarchy has c leaves which are children of the root. We allow our disk blocks to cover only rectangular regions and only one copy of each object on secondary storage (i.e., no two disk blocks will contain the same objects). For any fixed k (independent of q and B), there will be class indexing queries whose q items of output cannot be covered by only kq/B disk blocks.*

3 An Algorithm for External Semi-Dynamic Interval Management

As was shown by Proposition 2.2, an efficient data structure for external dynamic 2-dimensional range searching with diagonal corner queries can be used to efficiently solve the external dynamic interval maintenance problem. In this section we describe the *metablock tree*, an optimal data structure for this problem.

Initially, we describe the metablock tree for the static case, where all data is given and can be preprocessed before any queries are processed. Once this is done we describe methods whereby the data structure can be made semi-dynamic.

3.1 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 8.

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 *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 9. 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 10. 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.

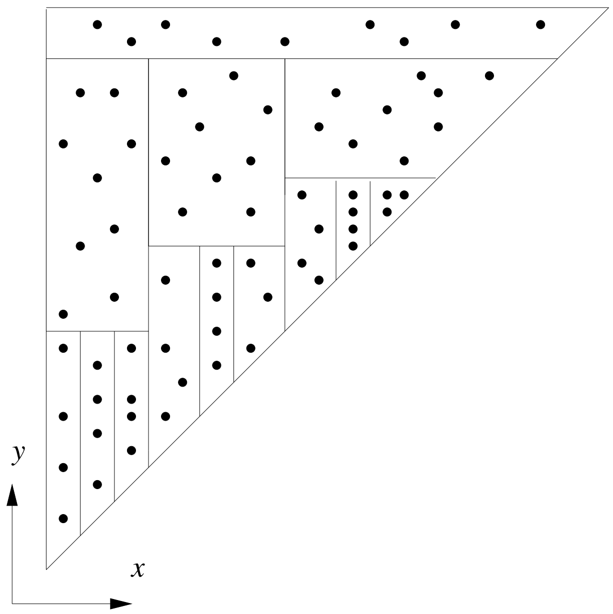
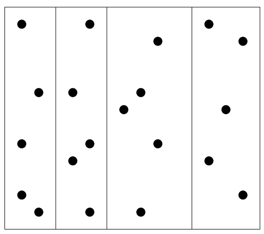
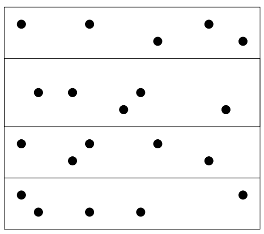


Figure 8: A metablock tree for $B = 3$ and $n = 70$. All data points lie above the line $y = x$. Each region represents a metablock. The root is at the top. Note that each non-leaf metablock contains $B^2 = 9$ data points.



(a) Vertically oriented



(b) Horizontally oriented

Figure 9: Vertically and horizontally oriented blockings of data points. Each thin rectangle represents a block.

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 8.) These blocks will be organized as prescribed by the following lemma:

Lemma 3.1 *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

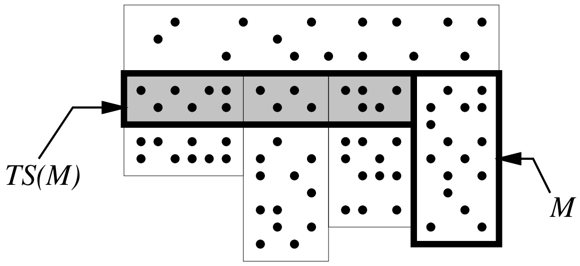


Figure 10: A metablock M and the set $TS(M)$. Note that $TS(M)$ spans all of M's left siblings in the metablock tree. Though it is not explicitly shown here, $TS(M)$ will be represented as a set of B horizontally oriented blocks.

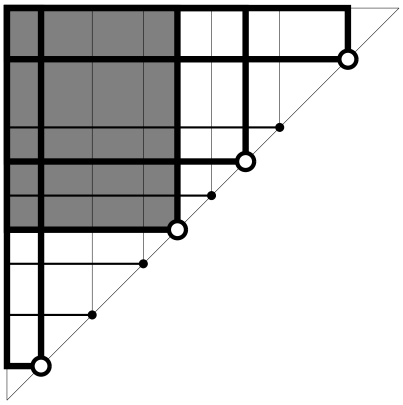


Figure 11: The sets C and C^* used in the proof of Lemma 3.1. The marked points lying along the diagonal line $y = x$ are the points in the set C. Those that are small and dark are points in $C \setminus C^*$. The larger open points are in the set C^*. The dark lines represent the boundaries of queries whose corners are at points $c \in C^*$. One such query is shaded to demonstrate what they look like.

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 11.

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 12. 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 $S_i = \Omega_i \cup \Delta_i^+$ be the answer to a query whose corner is c_i. We decide

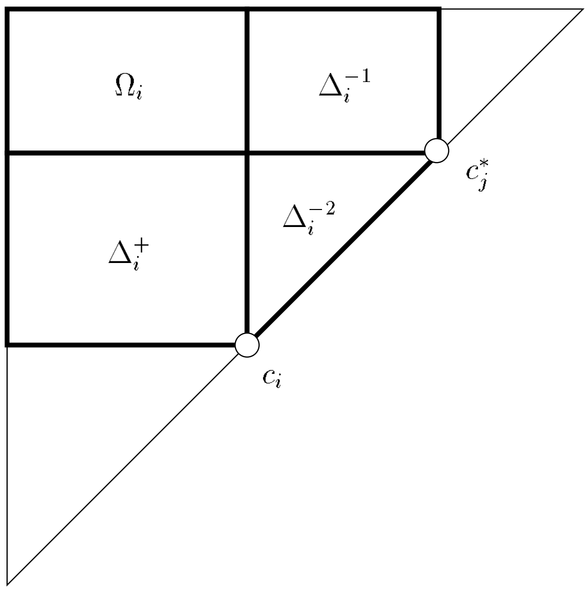


Figure 12: The sets Ω_i, Δ_i^{-1} , Δ_i^{-2} and Δ_i^+ as constructed in the proof of Lemma 3.1. c_j^* is the last point that was added to C^* and c_i is the point being considered for inclusion. The sets consist of subsets of S falling within the labeled regions.

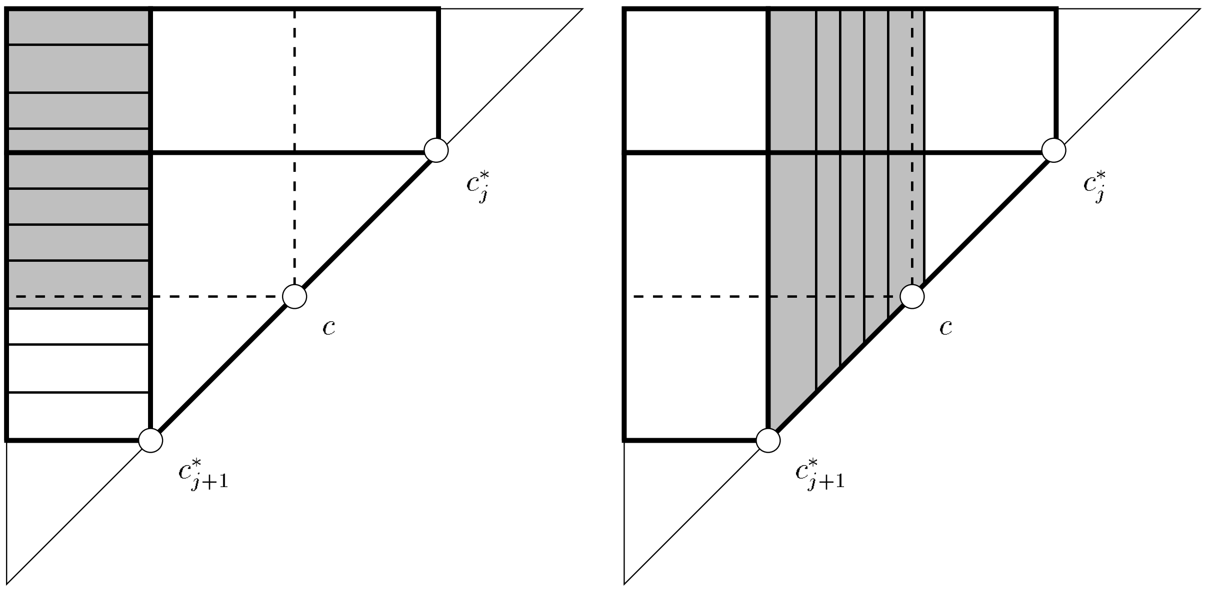
to add c_i to C^*, and thus explicitly store S_i, if and only if

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

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 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.



(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.

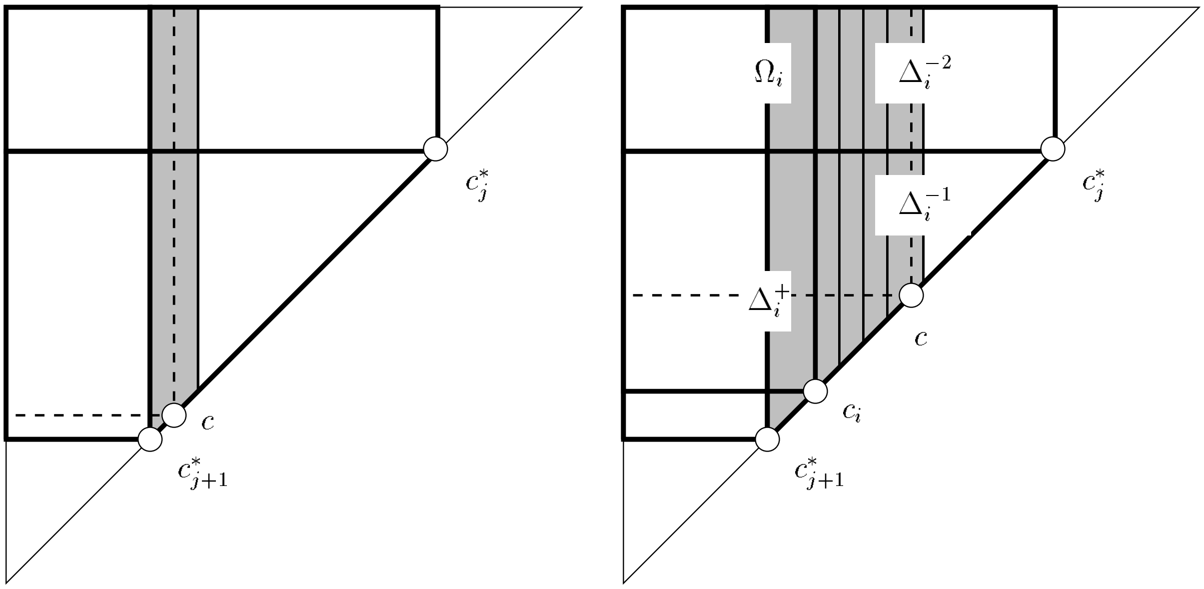
Figure 13: The two phases of the algorithm to answer a query whose corner is c as used in the proof of Lemma 3.1. c_j^* and c_{j+1}^* are consecutive elements of the set C^*, an thus answers to queries having them as corners are explicitly blocked.

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 13(a). At most one block is wasted in this process.

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 13(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 14(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 14: Two cases of the second phase of the algorithm (Figure 13(b)) used in the proof of Lemma 3.1.

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 14(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 13(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 3.2 *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 3.1 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 $O(n/B^2)$ metablocks. Thus a total of $O(n/B)$ blocks are used.

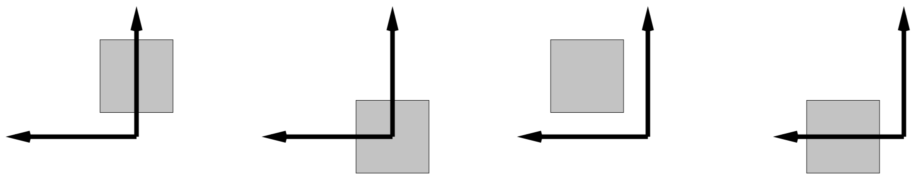
Queries are answered using the procedure *diagonal-query* in Figure 15 (by invoking *diagonal-query (query q, root of metablock tree))*. We associate every metablock with the minimum bounding rectangle of the points inside it. Call this the metablock's *region*. Also, a diagonal corner query q is completely 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 16. To complete the proof we simply look at the contributions of each type of metablock to the total number of I/Os required.

- 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 3.1 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.

```
(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 15: Procedure *diagonal-query* to answer diagonal corner queries using a metablock tree

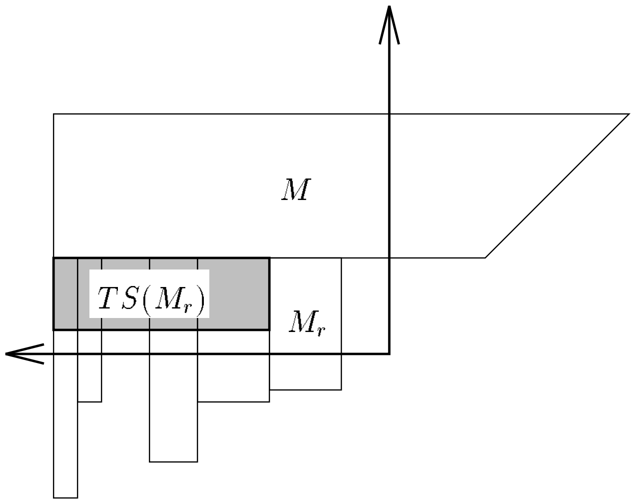


Type I Type II Type III Type IV

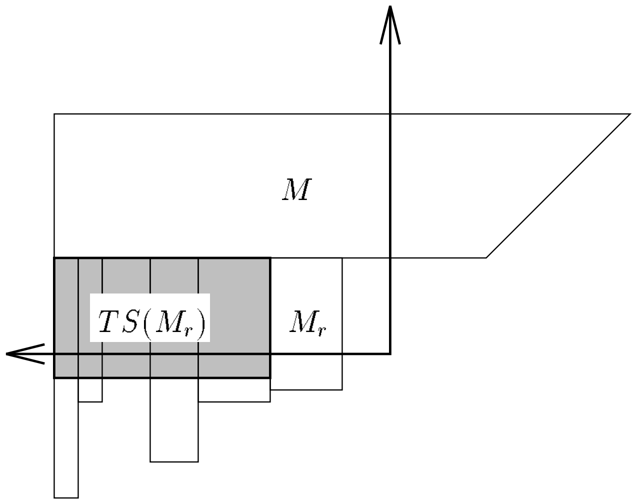
Figure 16: The four types of metablocks. The shaded rectangles represent metablock boundaries and the thick lines are the boundaries of queries. The four types appear in processing diagonal corner queries as described in Section 3 and are used in the proof of Theorem 3.2

- 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 the query. If the bottom boundary of $TS(M_r)$



(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 left 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 17: Use of the TS structure to search Type IV metablocks as used in the proof of Theorem 3.2.

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 17(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 17(b). In both cases, all the blocks examined can be charged to the $O(t/B)$ term.

□

Theorem 3.2 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.

Proposition 3.3 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

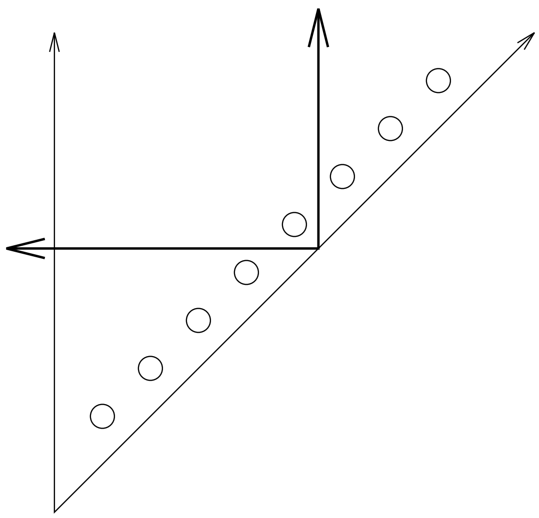


Figure 18: The set of points S and a query whose corner is an element of the set of Q as described in the lower bound proof of Theorem 3.3.

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 18. 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. □

3.2 Dynamization of Insertions

The data structure described in Section 3.1 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/O's to reorganize components (a), (b), and (c) 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 3.1. This costs $O(B)$ I/O's. This implies that we spend only $O(1)$ I/O's 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 into that particular metablock. 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/O's, 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 3.1. 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/O's),
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/O's).

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

Lemma 3.4 *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 3.1. 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. $\square$

(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 19: 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 3.5 *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 3.2. □

Lemma 3.6 *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/O's.

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/O's. 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/O's. 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/O's 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. $\Box$

We put everything together in the following theorem:

Theorem 3.7 *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/O's. The size of the data structure is $O(n/B)$ blocks of size B each.*

4 A Class Indexing Algorithm Using Hierarchy Decomposition

In Section 2 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 section, we have a preliminary lemma concerning our ability to answer 3-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 has constant depth and when the hierarchy is degenerate. Combining these techniques, we give an efficient solution to the whole problem.

Lemma 4.1 *[17] There exists a data structure that can answer any 3-sided query on a set of n points on the plane in $O(\log_2 n + t/B)$ disk I/O's. This data structure occupies $O(n/B)$ disk blocks and can be built in $O((n/B)\log_B n)$ disk I/O's.*

A data structure to achieve these bounds was presented in [17]. 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/O's.

Lemma 4.2 *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.*

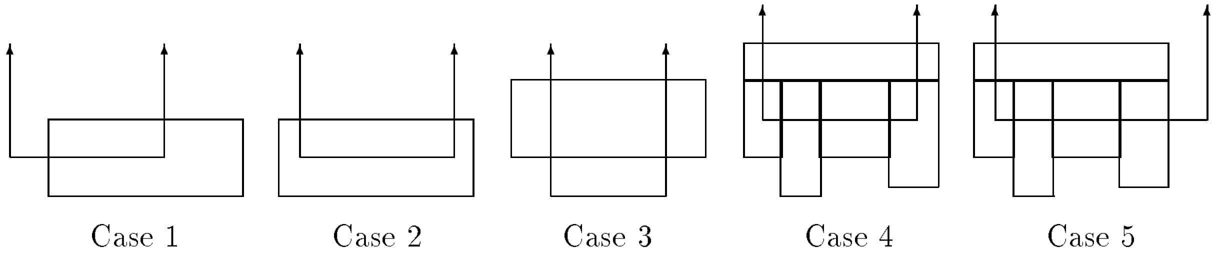


Figure 20: Problems with using the metablock tree for 3-sided queries.

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. $\square$

It should be pointed out that this lemma does not contradict Theorem 2.8 which applies when we have only one copy of the objects in the database. Lemma 4.2 uses k copies to index efficiently.

Lemma 4.3 *When the hierarchy is degenerate, the class indexing problem reduces to answering 3-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 Section 2, we reduced the class indexing problem to 2-dimensional range searching in secondary memory. To see why this reduces to answering 3-sided queries if the hierarchy is degenerate, consider the behavior of procedure *label-class* in Figure 4 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 3-sided query.

We will try to modify the metablock tree to solve this problem. The metablock tree solves 2-sided range queries in secondary memory where the corner always lies on the diagonal. 3-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 3-sided query on a metablock; (3) both the vertical sides of a 3-sided query may pass through the same metablock (remember that a 2-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 3-sided queries, this assumption is no longer true. See Figure 20 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 3-sided queries as prescribed in Lemma 4.1. Since that data structure uses optimal storage, our asymptotic storage does not change. Also, since 3-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 3-sided queries as prescribed in Lemma 4.1. The understanding is that this structure will be used whenever case (4) occurs. This 3-sided structure will be called the 3-sided structure for the children of M. Figure 21 gives a modified version of procedure *diagonal-query* to handle 3-sided queries.

The time bound analysis is very similar to the one in Theorem 3.2. While answering a 3-sided query, no more than three 3-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 3-sided structures have B^3 or less points in them and by the bounds of Lemma 4.1 can be used to answer 3-sided queries in $O(\log_2 B + t/B)$ disk I/O's. Combining this with the bounds of Theorem 3.2, we get this lemma. □

Lemma 4.4 *There exists a data structure that can answer any 3-sided query on a set of n points in* $O(\log_B n + \log_2 B + t/B)$ *disk I/O's. 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 3.6. The corner structures that we build for that proof become 3-sided structures. In particular, the TD corner structure for an internal node M becomes a 3-sided structure also. This will have an update block as the TD corner structure did before and will be rebuilt once in B insertions. When its size reaches B^2, the 3-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 3-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 3-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 3-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 3-sided structures for the siblings of M. Procedure *insert-point* in Figure 19, with minor modifications that take into account the 3-sided structures, can be used to perform inserts.

To get the bounds for the number of I/O's for an insert, we note that a 3-sided structure with B^2 (respectively B^3) points can be rebuilt in $O(B)$ $(O(B^2))$ disk I/O's as per Lemma 4.1. The analysis for Lemma 3.6 applies here and gives us the required bounds. □

```
procedure 3sided-query ( query q, node M);
 if query q is 3-sided then
      if M contains both the corners of q
           use the 3-sided structure for M to answer the query; return;
      else if both the vertical sides of query q fall inside one child of M
           let this child be M_c
           use M's vertically oriented blocks to report points that lie in between the vertical
                sides of the query q
           invoke 3sided-query (q, M_c);
      else /* the two vertical sides fall on different children */
           use M's vertically oriented blocks to report points that lie
                in between the vertical lines of q
           let M_l and M_r be the children of M containing the left and right sides of q resp.
           let M_1, M_2, ..., M_k be the children of M in between M_l and M_r
           use the 3-sided structure for M's children to determine points of M_1, M_2, ..., M_k
                that fall inside the query
           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.
           invoke 3sided-query(left side of q, M_l)
           invoke 3sided-query(right side of q, M_r)
 else /* query q is 2-sided */
      if M contains the corner of query q
           use the 3-sided structure for M to answer the query; return;
      else /* corner of q does not fall inside M */
           use M's vertically oriented blocks to report points of M that lie inside q
           let M_l be the child of M containing the vertical side of q
           let M_1, M_2, ..., M_k be the other children of M intersecting q
           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)
           invoke 3sided-query(q, M_l)
```

Figure 21: Procedure to answer 3-sided queries.

We now 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 [34]. The following lemma is easily proven by induction.

Lemma 4.5 *Let the procedure* label-edges *shown in Figure 22 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* $\log_2 c$.

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

Figure 22: An algorithm for labeling a tree with thick and thin edges as used in Lemma 4.5.

```
procedure rake-and-contract ( root );
  repeat
        for each leaf L connected by means of a thin edge to the tree do
                index collection associated with L;
                copy items in L's collection to its parent's collection;
                delete L from the tree and mark L as indexed;
        endfor
        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
                build a 3-sided structure for the path (as described in Lemma 4.3);
                copy all the collections associated with the nodes of the path into the collection
                        of the parent of the topmost node in the path;
                mark all the nodes in the path as indexed;
                delete all the nodes in the path from the hierarchy tree;
        endfor
until hierarchy tree is made up of one node only;
```

Figure 23: 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 23 takes as input a hierarchy processed by *label-edges* and applies procedures outlined in the proofs of Lemmas 4.2 and 4.3 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 4.6 *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 1-dimensional*

query on a B^+-tree) or a 3-sided structure is built for it as per Lemma 4.3 (which means that a query on this class can be answered by performing a 3-sided query on the 3-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 4.5 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 1-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 4.3 applies here and therefore an indexing query on the class can be answered by looking at the 3-sided query structure built in *rake-and-contract*. The lemma follows. See Figure 24 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 4.7 *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 $O((n/B)\log_2 c)$.*

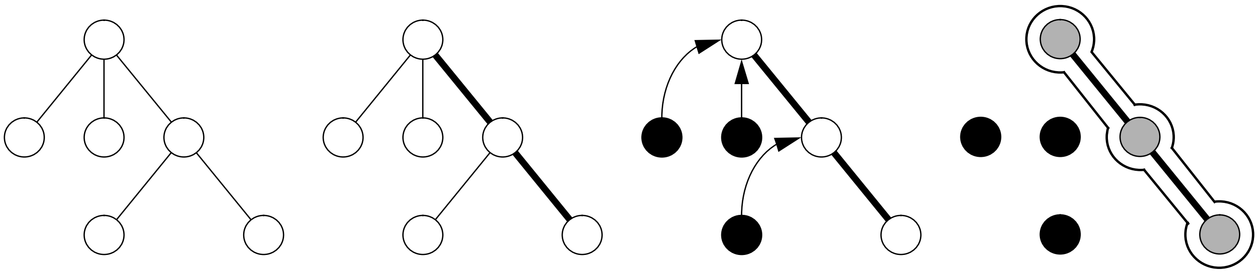


Figure 24: An example class hierarchy decomposition

5 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. Our algorithms for indexing constraints have optimal storage and query time, and log-suboptimal insert performance. Our algorithms for indexing constraints have improved space and query performance, and polylog-suboptimal insert performance (modulo amortization).

The data structures in Sections 3 and 4 should be viewed as existence proofs that, for these practical cases of 2-dimensional range searching, close to optimal I/O performance is achievable. Our new data structures have provably good performance, but are somewhat complex. One direction to focus on is whether simpler data structures can achieve the same bounds. For example, we believe that the class indexing algorithm in Section 2 is practical, even if suboptimal.

We believe that it should be possible, using standard data structure techniques, to transform our insertion bounds from amortized to worst-case (although we have not done so here). Whether they can be asymptotically improved is an open question.

The performance for the case of deletions is open. We should note that, using the techniques in this paper to dynamize the static structure of [17], it is possible to achieve the following dynamic bounds: (1) indexing constraints in $O(n/B)$ pages, dynamic query I/O time $O(\log_2 n + t/B)$ and amortized update time $O(\log_2 n + (\log_2^2 n)/B)$, and (2) indexing classes in $O((\log_2 c)(n/B))$ pages, dynamic query I/O time $O(\log_2 n + t/B)$ and amortized update time $O((\log_2 c)(\log_2 n + (\log_2^2 n)/B))$.

Progress has been made in many different directions after this paper. In the case where our goal is to handle a very large number of queries in a batch, [14] proves that 2-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.

A new technique called *path caching* was presented in [29] to convert many main-memory data structures like the priority search tree, segment tree, etc. into efficient secondary storage structures. For example, it is shown in [29] that it is possible to implement priority search trees in secondary memory so that 2-sided queries can be answered in optimal $O(\log_B n + t/B)$ disk I/O's while using a storage of $O((n/B)\log_2 \log_2 B)$. Both inserts and deletes can be made to this data structure. The amortized cost of an update is $O(\log_B n)$.

Recently, a much stronger version of the lower bound in Lemma 2.7 was proved in [36]. They show that any secondary storage data structure that can answer 2-dimensional range queries in $O(\log_B^k n + t/B)$ I/O's in the worst-case (where k is a constant), has to occupy $\Omega(\frac{n}{B}\log(\frac{n}{B})/(\log\log_B n))$ disk blocks. This offers conclusive proof that 2-dimensional range searching in secondary storage is much harder than 1-dimensional searching. This paper also contains a data structure that comes within a small term of achieving the optimal query time using optimal space.

We close with the most elegant open question: 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 $O(\log_B n)$?

References

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

[2] N. Beckmann, H. Kriegel, R. Schneider, and B. Seeger, "The R*-tree: An Efficient and Robust Access Method for Points and Rectangles," *Proc. 1990 ACM-SIGMOD Conference on Management of Data* (1990), 322–331.

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

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

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

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

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

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

[14] 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.

[15] 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.

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

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

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

[19] 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.

[20] 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.

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

[22] 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.

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

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

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

[26] 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.

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

[28] 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.

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

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

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

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

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

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

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

[36] S. Subramanian and S. Ramaswamy, "The P-range tree: A new data structure for range searching in secondary memory," *To appear in the Proceedings of the Sixth Annual ACM-SIAM Symposium on Discrete Algorithms* (1995).

[37] 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.

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