Indexing for Data Models with Constraints and Classes

Paris C. Kanellakis Sridhar Ramaswamy Darren E. Vengroff and Jeffrey S. Vitter

Department of Computer Science
Brown University
Providence, Rhode Island 02912

CS-93-21
May 1993

Indexing for Data Models with Constraints and Classes*

(Extended Abstract)

Paris C. Kanellakis† Sridhar Ramaswamy‡ Darren E. Vengroff§
Jeffrey S. Vitter¶

December 9, 1992

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 secondary storage page size, and t the size of the output of a query. Indexing by one attribute in the constraint data model (for a fairly general type of constraints) 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 which has optimal worst-case space $O(n/B)$ pages and optimal query I/O time $O(\log_B n + t/B)$ and has $O(\log_B n + (\log_B^2 n)/B)$ amortized insert I/O time. If the order of the insertions is random then the expected number of I/O operations needed to perform insertions is reduced to $O(\log_B n)$. 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 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 from $O(\log_2 c \log_B n + t/B)$ to $O(\log_B n + t/B + \log_2 B)$.

*Also appears in Proc. 1993 ACM Conference on Principles of Database Systems.

†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.

‡Address: Dept. of Computer Science, Brown University, Box 1910, Providence, RI 02912. Email: sr@cs.brown.edu. Tel: 401-863-7662. Fax: 401-863-7657. Research supported by ONR Contract N00014-91-J-4052, ARPA Order 8225.

§Address: Dept. of Computer Science, Brown University, Box 1910, Providence, RI 02912. Email: dev@cs.brown.edu. Support was provided in part by National Science Foundation research grant CCR–9007851 and by Air Force Office of Scientific Research grant number F49620–92–J–0515.

¶Address: Dept. of Computer Science, Brown University, Box 1910, Providence, RI 02912. Email: jsv@cs.brown.edu. Support was provided in part by Army Research Office grant DAAL03–91–G–0035 and by Air Force Office of Scientific Research grant number F49620–92–J–0515.

1 Introduction

Motivation and Background: The successful realization of any data model in a large scale database requires supporting its language features with efficient secondary storage manipulation. For example, the relational data model, [Cod] includes declarative programming (i.e., in the relational calculus and algebra) and expresses low data complexity queries (i.e., every fixed relational calculus query is evaluable in Logspace and Ptime in the size of the input database). Most importantly, these language features can be supported by data structures for searching and updating that make effective use of secondary storage (i.e., use I/O time logarithmic or faster in the size of input relations). B-trees and their variants B^+-trees [BaM, Com] are examples of such data structures and 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. This has been a well studied problem. (McCreight calls the 2-dimensional as opposed to the 1-dimensional case "one of the persistent puzzles of computer science" [McC].) In this paper we examine new I/O-efficient data structures for special cases of this general problem, which are important for supporting new language features, such as constraint query languages [KKR] and class hierarchies in object-oriented databases [KiL, ZdM].

We make the standard assumption that each secondary memory access transmits one page or B units of data, and we count this as one I/O. (We will use the words page and disk block interchangeably.) We also assume that $O(B^2)$ words 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 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. Our I/O bounds will be expressed in terms of n, c, t and B (i.e., all constants are independent of these four parameters); for a survey of state of the art I/O complexity see [Vit]. We first review external dynamic k-dimensional range searching, with B^+-tree performance as our point of reference whose performance we refer to as *optimal.*

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 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).

A large literature exists for in-core algorithms for 2-dimensional range searching. Basic ideas such as the range-range tree data structure and the use of fractional cascading lead to the best worst-case in-core bounds achieved to date: $O(n \log_2 n)$ space, static query time $O(\log_2 n + t)$, dynamic query time $O(\log_2 n \log_2 \log_2 n + t)$ and update time $O(\log_2 n \log_2 \log_2 n)$ (due to space limitations we refer the reader to [ChT] for a detailed survey of the topic). The ideal worst-case I/O bounds would involve making all the above logarithms 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, [OSB, SmO] examine mappings which maintain the logarithmic overheads and make the logarithms base B; however, their model does not compact the t-sized output on t/B pages.

The practical need for good I/O support has led to the development of a large number of external data structures, which 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 (due to space limitations we refer the reader to [Sama, Samb] for a recent survey and applications). 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.

In this paper we examine algorithms with provably good worst-case I/O bounds for indexing problems in data models with constraints and classes. In order to put our contributions into perspective we point out (from the literature by using standard mappings of in-core to external data structures) 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 update I/O time $O(\log_2 n \log_B n)$. (Note that, $\log_2 n = (\log_2 B)(\log_B n)$ is I/O inefficient when compared to $\log_B n$).

Based on the special structure of the indexing problems of interest we improve on the above bounds.

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 [KKR]. This framework adapts ideas of Constraint Logic Programming or CLP, e.g., from [JaL], 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 more detailed explanation of the problem). Fortunately, it is possible to combine CQLs with existing 2-dimensional range searching data structures [KKR]. 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, which is a special case of external dynamic 2-dimensional range searching.

Dynamic interval management has been examined in the literature (see [ChT]). The best in-core bounds have been achieved using the *priority search tree* data structure of [McC], yielding $O(n)$ space, dynamic query time $O(\log_2 n + t)$ and update time $O(\log_2 n)$, which are all optimal. It is open whether dynamic interval management on secondary storage can be achieved optimally in $O(n/B)$ pages, dynamic query I/O time $O(\log_B n + t/B)$ and update time $O(\log_B n)$. Note that, various suboptimal solutions are proposed in [IKO], for a slightly more general problem, as well as a claimed optimal static solution. Unfortunately, the [IKO] 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. In this paper we provide an optimal static solution for external dynamic interval management. This static solution is quite involved, but it achieves the optimal space and query time bounds with small constants. We also semi-dynamize this solution with very good amortized insert time. More specifically:

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. In Section 3, we propose a new data structure for this problem. Our data structure has optimal worst-case space $O(n/B)$ pages and optimal query I/O time $O(\log_B n + t/B)$ and has $O(\log_B n + (\log_B^2 n)/B)$ amortized insert I/O time. If the order of the insertions is random then the expected number of I/O operations needed to perform insertions is reduced to $O(\log_B n)$.

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 [MaS], it constitutes the basis for indexing in object-oriented databases. Indexing classes has been examined in [KKD] and more recently in [LOL], but the solutions offered there are largely heuristic with poor worst-case performance.

In Section 2.2, we reduce indexing classes to a special case of external dynamic 2-dimensional range searching. We also assume that the class-subclass relationship is static, although objects can be inserted or deleted from classes. Under this reasonable assumption for c-class hierarchies 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 with only one copy of each object in secondary storage. In Section 4, analyzing the hierarchy and using techniques from the constraint indexing problem, we improve query I/O time to $O(\log_B n \ + \ t/B \ + \ \log_2 B)$ using space $O((n/B)\log_2 c)$ pages. Amortized update I/O time for the semi-dynamic problem (with inserts only) is $O(\log_2 c((\log_B^2 n)/B + \log_B n + \log_2 B))$.

We close in Section 5 with open problems and a discussion of how to dynamize deletions in addition to insertions.

2 The Problems and Base Algorithms

2.1 Indexing Constraints

To illustrate indexing constraints in CQLs consider the domain of rational numbers and the language whose syntax consists of the theory of rational order with constants + the relational calculus. The semantics of a program in this language is based on the theory of rational order with constants, by interpreting relational atoms as shorthands for formulas of the theory. An input generalized database (e.g., a generalized relation of arity k) is a quantifier-free DNF formula, over k variables, of the theory. For each input generalized database, the queries can be evaluated in closed form, bottom-up, and efficiently in the input size.

For example, let the database contain a set of planar rectangles with rational coordinates. We wish to compute all pairs of distinct intersecting rectangles. Let $R(z, x, y)$ be a ternary generalized relation; we interpret $R(z, x, y)$ to mean that (x, y) is a point in the rectangle with name z. A rectangle can 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 $\{(n_1, n_2) \mid n_1 \neq n_2 \wedge (\exists x, y)(R(n_1, x, y) \wedge R(n_2, x, y)\}$. 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 etc. This simplicity of expression can be combined with efficient evaluation techniques, even if quantification is over the infinite domain of rationals.

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.

In many cases, the projection of any generalized tuple on x is one interval $a \leqslant x \leqslant a'$. This is true for the above example, for CQLs with any dense linear order, for relational calculus with linear inequalities over the reals, and in general when a generalized tuple represents a convex set. Under such assumptions, which we call *convex* CQLs, there is a good solution for our indexing problem.

(1) 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*.

(2) 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.

(3) Inserting or deleting a given generalized tuple is performed by computing its projection and inserting or deleting intervals from a set of intervals.

We have reduced the problem to dynamic interval management. This is a well-known problem with many elegant in-core solutions from computational geometry. In this paper we examine solutions for this problem with good I/O performance. Using simple techniques from computational geometry (we reduce external interval intersection to external stabbing queries and we map intervals into points in 2d space) and *diagonal range queries* (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) we can show the following.

Lemma 2.1 *Indexing constraints for convex CQLs reduces to external dynamic 2-dimensional range searching with diagonal corner queries and updates. This reduction can be done using $O(n/B)$ pages, dynamic query I/O time $O(\log_B n + t/B)$ and update I/O time $O(\log_B n)$.*

2.2 Indexing Classes

To illustrate the problem of indexing classes, consider an object-oriented database (e.g., containing information about people such as names and incomes). The *objects* in the database (e.g., the people) 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 this class hierarchy be a tree with root Person, two children of Person called Professor, Student, and a child of Professor called Assistant-Professor. One 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 are all people, where as, the extent of Person are people who are not in the Professor, Assistant-Professor, and Student extents.

The problem of *indexing classes* means being able to do *external dynamic 1-dimensional range searching on some attribute of the objects, but for the full extent of each class in the hierarchy*. For example, find all people in (the full extent of) class Assistant-Professor with income between \$50K and \$60K, or find all people in (the full extent of) class Person with income between \$100K and \$200K, or 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. One way of indexing classes is to create a single B^+-tree for all objects (i.e., index the collection of all objects) and then filter out the objects in the class of interest. This solution cannot compact a t-sized output into t/B pages. 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, dynamic 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 it is easy to see that indexing classes is a special case of external dynamic 2-dimensional range searching on some attribute of the objects. One can use the range-range idea with classes as the primary dimension and the object attribute as a secondary dimension and show the following:

Lemma 2.2 *Indexing classes reduces to external dynamic 2-dimensional range searching with one dimension being static. This reduction can be done using $O(n/B)$ pages, dynamic query I/O time $O(\log_B n + t/B)$ and update time $O(\log_B n)$.*

Lemma 2.3 *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.*

The problem of indexing classes, despite its structure, is nontrivial. First, observe that if the class hierarchy is a path then the range queries involved are 3-sided. Priority search trees were developed for in-core processing of such queries. Second, if the class hierarchy is a balanced tree of depth $\log_2 c$ one can show that the solution cannot be as good as the 1-dimensional B^+-tree solution.

Theorem 2.4 *Consider the fully static problem of indexing classes, where the hierarchy is a balanced binary tree of depth* $\log_2 c$*. It is not possible to achieve static query I/O time* $O(\log_B n + t/B)$ *with only one copy of each object on secondary storage.*

3 An Algorithm for External Dynamic Interval Management

By Lemma 2.1, if we want to solve the external dynamic interval maintenance problem it suffices to consider external dynamic 2-D range searching with diagonal corner queries.

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

For conceptual simplicity, we first consider the static case, in which we must answer queries, but data points will neither be inserted nor deleted. The semi-dynamic case will be considered in Section 3.2. The data structure we will use is called a *metablock tree*.

At the outermost level, a metablock tree 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 1.

Now let us consider how we can store a set of k data points in blocks of size B. One very simple scheme would be 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 could put the data points into *vertically oriented* blocks by discriminating on the x coordinates. These techniques are illustrated in Figure 2. Each metablock in our tree is divided into both horizontally and vertically oriented blocks. This means that each data point is represented more than once, but our 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. Next, each metablock M contains pointers to B blocks that represent, in horizontal orientation,

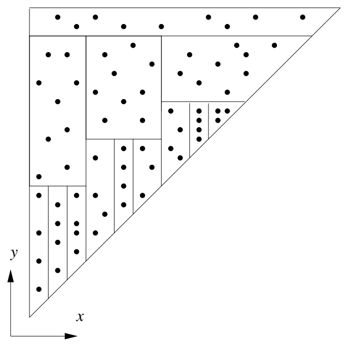

Figure 1: A metablock tree for $B = 3$ and $n = 70$. All data points lie above the line $y = x$. Each region represents a metablock. The root is at the top. Note that each non-leaf metablock contains $B^2 = 9$ data points.

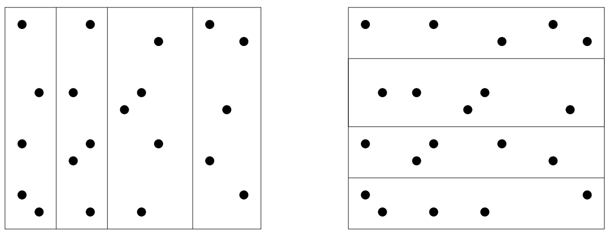

Figure 2: Vertically and horizontally oriented blockings of data points. In this illustration, $B = 5$. Each thin rectangle represents a block.

the set $TS(M)$. $TS(M)$ is the set obtained by taking the B^2 data points with the largest y coordinates from among the up to $B^3 - B^2$ data points represented by siblings of M that lie to its left. This is illustrated in Figure 3. Note that each metablock already requires $O(B)$ blocks of storage space, so storing $TS(M)$ for each metablock does nothing to asymptotic space usage of the metablock tree.

The final bit of organization left is used only for those metablocks that can possibly contain the corner of a query. These are the leaf metablocks, the root metablocks, and all metablocks that lie along the path from the root to the rightmost leaf. 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 $O(t/B + 1)$ I/O operations where t is the number of data points in S that lie within the query region.*

Proof: (sketch) We divide S into a vertically oriented blocking of k/B blocks. Let us first restrict ourselves to queries whose corners are in the set C of points at which right boundaries of the block regions intersect the line $y = x$. We choose a subset $C^* \subset C$ of these points and use one or more blocks to explicitly represent the answer to each query that happens to have a corner $c \in C^*$. This is illustrated in Figure 4.

We handle a query whose corner c is in C^* by simply reporting the explicit result that

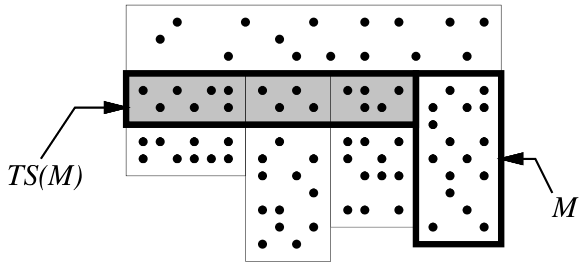

Figure 3: A metablock M and the set $TS(M)$. Note that $TS(M)$ spans all of M's left siblings in the metablock tree. Though it is not explicitly shown here, $TS(M)$ will be represented as a set of B horizontally oriented blocks. In this illustration, $B = 4$.

we stored for it. For a query whose corner c is in the set $C \setminus C^*$, we iteratively examine blocks in the vertically oriented blocking of S, starting at the block just to the left of c and moving to the left until we reach a vertical block bounded on the right by a point $c^* \in C^*$. We then simply examine the blocks that explicitly store the answer to the query cornered at c^*.

The trick that makes the data structure work is that C^* is chosen so as to satisfy the following conditions:

- The total number of blocks needed to store the outputs of all the queries cornered at points in C^* is $O(k/B)$, so the space usage is optimal.
- For any point $c \in C \setminus C^*$, the number t of points in the output to the query cornered at c is at least half the number of points retrieved in the blocks, up to an additive term of $O(B)$. This means that the query time is $O(t/B + 1)$.

The details of the construction and the full proof are deferred to the full paper, as is the remainder of the proof, which consists of showing that we can make a minor modification to our data structure to handle queries whose corners are on the line $y = x$ but not in the set C. □

We can now prove the following theorem:

Theorem 3.2 *If a set of n data points (x, y) in the half plane $y \geqslant x$ 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, which is optimal. The size of the data structure is $O(n/B)$ blocks of size B each.*

Proof: (sketch) The space usage is $O(n/B)$ from Lemma 3.1 and the construction of the tree given above. 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. The four types are illustrated in Figure 5.

To complete the proof we simply look at the contributions of each type of metablock to I/O efficiency and to t. A type I metablock returns B^2 data points and uses $O(B)$ I/O operations. There are at most $O(\log_B n)$ type II nodes, each of which can be queried, using

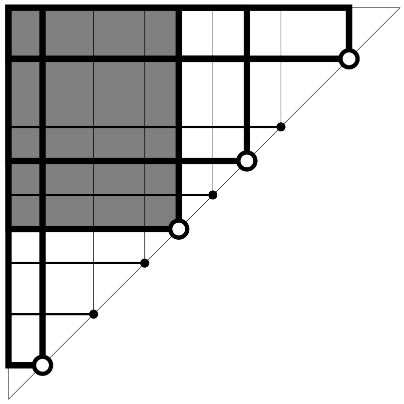

Figure 4: Illustration of Lemma 3.1. The marked points lying along the diagonal line $y = x$ are the points in the set C. Those that are small and dark are points in $C \setminus C^*$. The larger open points are in the set C^*. The dark lines represent the boundaries of queries whose corners are at points $c \in C^*$. One such query is shaded to demonstrate what they look like.

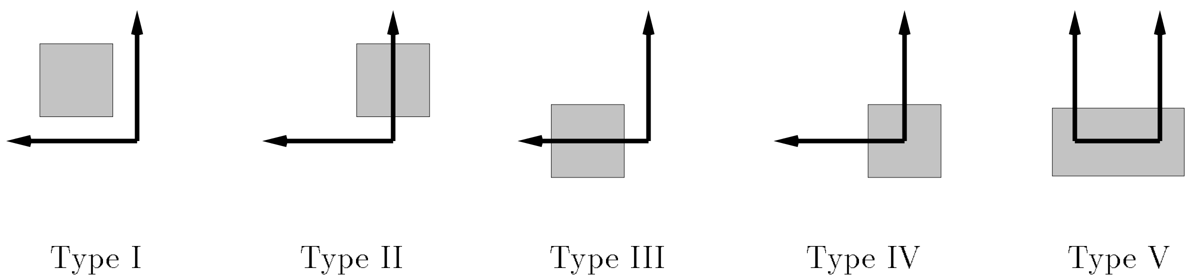

Figure 5: The five types of metablocks. The shaded rectangles represent metablock boundaries and the thick lines are the boundaries of queries. The first four types appear in processing two sided queries as described in Section 3. In addition to these four types, type V metablocks can occur when we process three sided queries as described in Section 4.

its vertically oriented blocks, so as to visit at most one block that is not completely full. Only one type IV node can exist, and if it contains k data points within the query region then by Lemma 3.1 they can be found using only $O(t/B)$ I/O operations. The set of all type III children of a type I node can be queried, using their horizontally oriented blocks, so as to examine at most $O(B)$ blocks that are not entirely full. Since we used $O(B)$ I/O operations for the output from the type I block, the extra type III I/O operations can be absorbed into the type I I/O. Finally, a large number of type III nodes can be children of a singe type II node M. These can be efficiently queried by examining $TS(M)$. Filling in the details, we get the desired optimal time bound of $O(\log_B n + t/B)$ I/O operations. □

3.2 Dynamization of Insertions

The data structure described in the previous section can be made semi-dynamic in the sense that it will support the insertion of points and remain efficient. The bounds and methodology are given by the following theorem:

Theorem 3.3 *An $O(n/B)$ block metablock tree of n data points can be made semi-dynamic, so that insertion of a data point can be performed in $O(\log_B n + (\log_B^2 n)/B)$ amortized I/O operations and diagonal corner queries can be performed in optimal $O(\log_B n + t/B)$ I/O operations.*

Proof: (sketch) Because the details are quite complex, we defer a complete proof of this theorem to the full version of the paper. The arguments center on the following four ideas:

1. We allow horizontally and vertically oriented blocks of data points to fill to twice their capacity before we take any action. Because it takes B insertions for a block that ordinarily holds B data points to get this full, we know that once it does we can perform $O(B)$ I/O operations and have their cost amortized over the insertions.

2. For the corner structures described in Lemma 3.1 we maintain an additional block recording points that have been added to the metablock but not yet incorporated into the corner structure. This block is called an *update list* for the corner structure. Once the update list becomes full (after B insertions), we have done enough insertions to amortize the cost of reading in the corner structure, reorganizing it, and writing it back to secondary memory. When queries are performed on a corner structure that has pending updates, we can afford to examine the one additional block in which contains the update list.

3. Points inserted into the metablock tree may have to be inserted into up to $B - 1$ TS structures. Because we cannot possibly afford the cost of doing this, we cache all updates made to the children of a given metablock into an additional corner structure, which we call the *parent update list*, which itself handles insertions through an update list as described in 2. We allow this structure to contain up to B^2 points before we update the TS structures of all of the children of the metablock with which it is associated. Whenever a query requires us to examine a TS structure, we also examine the parent update list associated with that TS structure, which, by Lemma 3.1 can be done efficiently.

4. Whenever a subtree of the metablock tree becomes imbalanced we can split one of the children of its root to restore balance. This is allowed to continue until a metablock M has $2B$ children instead of the normal B, at which point enough insertions have been performed to pay for rebalancing the subtree rooted at M.

$\square$

If we make the additional assumption, which is popular in the analysis of randomized algorithms in computational geometry, that the n data points to be inserted in the data structure are arbitrary, and can be picked by an adversary, but that the order in which they are inserted is equally likely to be any of the $n!$ possible orderings, we can eliminate the extra $\log_B n$ factor associated with rebalancing. This gives us the following result:

Theorem 3.4 *If n data points are inserted into a metablock tree in random order, the total expected time to process the insertions is $O(n \log_B n)$.*

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 algorithm used $O(n/B \; (\log_2 c))$ storage. Here c is the size of the class hierarchy, n is the size of the problem and B is the disk block size.

We now consider two extremes of the problem and show that they both have efficient solutions. We call a class hierarchy *degenerate* when it consists of a tree where every node has only one child. We give efficient solutions to the class indexing problem when the hierarchy is degenerate and when the hierarchy has constant depth. Combining these techniques, we give an efficient solution to the whole problem.

Lemma 4.1 *Consider an instance of the class indexing problem where k is the maximum depth of the class hierarchy and n is the size of the problem instance. We can index the class hierarchy so that the worst case query time is $O(\log_B n \; + \; t/B)$, the worst case update time is $O(k \; \log_B n)$, and the scheme uses $O((n/B) \; k)$ storage. This is optimal when k is constant.*

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

Lemma 4.2 *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 $O(\log_B n + \log_2 B + t/B)$, the amortized insert time is $O((\log_B^2 n)/B + \log_B n + \log_2 B)$ per operation, and the storage space required is $O(n/B)$.*

Proof: (sketch) In Section 2, we reduced the class indexing problem to 2D range searching in secondary memory. When the hierarchy is degenerate, the fourth side is always $y = 0$ and we only have to deal with 3-sided range queries.

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; and (3) both corners may lie on metablocks which are children of the same metablock.

Unfortunately, we do not know of any algorithm with a performance bound like that of Theorem 3.2. The method of [IKO] solves an instance of this problem of size m with a worst case query time of $O(\log_2 m + t/B)$. Their scheme uses $O(m/B)$ storage. We use their scheme to deal with type IV and type V metablocks and build auxiliary structures for each metablock. Since their scheme uses optimal storage, our asymptotic storage complexity does not change. Their structure can be amortized in the same way as the metablock tree. The bounds in the lemma then follow from our analysis of the metablock tree in Section 3. $\square$

```
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;
```

Algorithm 1: An algorithm for labeling a tree with thick and thin edges.

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 [SlT]. The following lemma is easily proved using induction.

Lemma 4.3 *Let Algorithm 1 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 $\log_2 c$.*

We are now ready to prove the key lemma. Algorithm 2 takes as input a hierarchy processed by Algorithm 1 and applies procedures outlined in the proofs of Lemmas 4.1 and 4.2 appropriately to parts of the hierarchy. Initially, we associate a unique collection to each class. This collection will contain the extent of the class.

Lemma 4.4 *Let an instance of the class indexing problem have class hierarchy size c, problem size n. Let the disk block size be B. Let us index the collections as per Algorithm 2. Then we have*

(1) No extent of any class is duplicated more than $\log_2 c$ times; and

(2) Every class in the input class hierarchy gets indexed in the sense that either an explicit index is built for its full-extent or a search structure is built for it as per Lemma 4.2.

Proof: We copy the extent of a class as many times as there are thin edges from it to the root of the hierarchy. Part (1) follows immediately follows from Lemma 4.3.

It is easy to see one of the two for loops in Algorithm 2 runs at least once unless the hierarchy has size one. This implies that every iteration of the repeat loop reduces the size of the hierarchy, which implies that the algorithm will terminate.

Part (2) follows easily. Notice that a node gets deleted from the hierarchy only when an explicit index is built for the items belonging to it or when a structure is built for it according to Lemma 4.2. This implies that every class in the input hierarchy is indexed in one of the two ways. See Figure 6 for an example of how a hierarchy is processed. □

We put everything together in the following theorem:

```
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 4.2);
         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;
```

Algorithm 2: The rake and contract algorithm for reducing an arbitrary hierarchy tree.

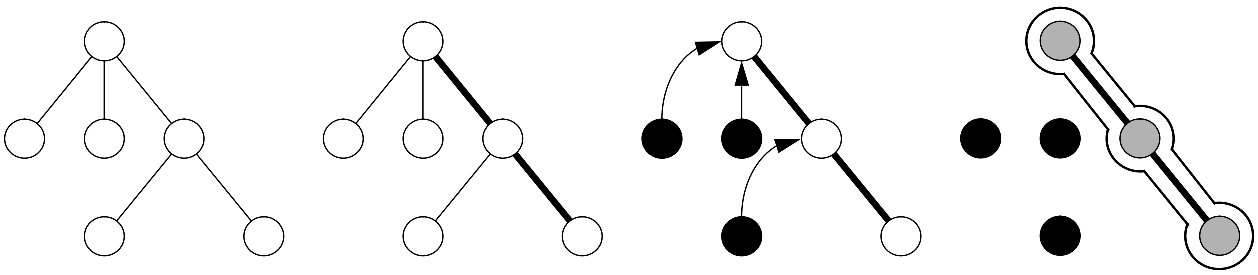

Figure 6: An example class hierarchy decomposition

Theorem 4.5 *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* $O(\log_2 c((\log_B^2 n)/B + \log_B n + \log_2 B))$ *per operation, and the storage space required is* $O((n/B)\log_2 c)$*.*

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.

Our new data structures have provably good performance, but are complex. Whether simpler solutions exist, even for the static case, is open. From the point of view of implementation 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.

In addition, we believe that it is possible, using standard data structure techniques, to transform our insertion bounds from amortized to worst-case (although we have not checked all the details yet). Whether they can be asymptotically improved is an open question.

The performance for the case of deletes is open. We should note that, using the techniques in this paper to dynamize the static structure of [IKO] 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 time $O(\log_2 c((\log_2^2 n)/B + \log_2 n))$.

We close with the most elegant open question: can dynamic interval management on secondary storage be achieved optimally in $O(n/B)$ pages, dynamic query I/O time $O(\log_B n + t/B)$ and worst case update time $O(\log_B n)$?

References

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

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

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

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

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

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

[KKR] P. C. Kanellakis, G. M. Kuper, and P. Z. Revesz, "Constraint Query Languages," *Proc. 9th ACM PODS* (1990), 299–313.

[KKD] 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.

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

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

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

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

[OSB] 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.

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

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

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

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

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

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