OODB Indexing by Class-Division

Sridhar Ramaswamy* Paris C. Kanellakis †

Abstract

Indexing a class hierarchy, in order to efficiently search or update the objects of a class according to a (range of) value(s) of an attribute, impacts OODB performance heavily. For this indexing problem, most systems use the class hierarchy index (CH) technique of [15] implemented using B^+-trees. Other techniques, such as those of [14,18,30], can lead to improved average-case performance but involve the implementation of new data-structures, which are not as well-understood parts of database technology as B^+-trees. As a special form of external dynamic two-dimensional range searching, this OODB indexing problem is solvable within reasonable worst-case bounds [12]. Based on this insight, we have developed a technique, called indexing by class-division (CD), which we believe can be used as a practical alternative to CH. We present an optimized implementation and experimental validation of CD's average-case performance. The main advantages of the CD technique are: (1) CD is an extension of CH that provides a significant speed-up over CH for a wide spectrum of range queries – this speed-up is at least linear in the number of classes queried for uniformly distributed data and larger otherwise; and (2) CD queries, updates and concurrent use are implementable using standard B^+-tree technology. The basic idea of class-division involves a time-space tradeoff and CD requires some space and update overhead in comparison to CH. In practice, this overhead is a small factor (2 to 3) and, in worst-case, is bounded by the depth of the hierarchy and the logarithm of its size.

1 Introduction

1.1 Motivation and Basic Problems of OODB Indexing

Linguistic features of data models enhance their applicability, but are really useful only if they can be supported by efficient secondary storage access. For example, the *relational data model* [7] includes declarative programming in the form of relational calculus or algebra. Its commercial success was largely due to the fact that its elegant linguistic features can be supported by data-structures that make efficient use of secondary storage. *Object-oriented data models* offer additional linguistic features, such as class hierarchies, inherited attributes, and object types, that match application semantics better than the relational data model [1,16, 32]. We believe that efficient indexing is critical in making Object Oriented Databases (OODBs) competitive in terms of performance with relational technology. The principal motivation for our work is the development of indexing techniques to support OODBs as efficiently as B-trees [2,8] support relations. We recall that B-trees and their variants B^+-trees are the canonical examples of relational database physical support and have been an unqualified success for external dynamic one-dimensional range searching, which is the most common problem solved by indexing in relational databases.

*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: sr@cs.brown.edu.

†Current address: Dept. of Computer Science, Brown University, Box 1910, Providence, RI 02912. Research supported partly by ONR Contract N00014–94–1–1153 and by ONR Contract N00014–91–J–4052, ARPA Order 8225. Email: pck@cs.brown.edu.

The problem of OODB indexing by one attribute and by class name, called *class indexing*, where objects are organized as a forest hierarchy of classes, is a special case of external dynamic two-dimensional range searching (see below). Together with the different problem of *nested object indexing* (as in [4,11,13, 19]) it constitutes the current repertoire of OODB indexing problems. Class Indexing has been examined in [15] and more recently in [14,18,30]. These solutions are heuristic (with poor worst-case performance) and have been supported by experimental performance evaluation. Of these, the *class hierarchy index* (CH) solution of [15] is the only method used widely in practice. We believe this is the case because it was the first reasonable approach proposed and because it uses standard data-structures such as B-trees. In this paper we propose a practical alternative (and addition) to CH.

We consider databases with objects organized in a forest class hierarchy (the basic method can be extended to handle directed acyclic graph hierarchies, but the worst-case analysis works for forests). We use N for the total number of objects in a class hierarchy with c classes and K for the output size of a query on this hierarchy. We make the standard assumption that each secondary memory access transmits one page or B units of data or one disk block, and we count this as one I/O. The performance of our algorithms is measured in terms of the number of I/Os they need for querying and updating and the number of disk blocks they require for storage. The I/O bounds are expressed in terms of N, c, K and B, i.e., all constants are independent of these four parameters.

A B^+-tree on attribute A of the N-tuple relation uses $O(N/B)$ pages of secondary storage. The following operations define the problem of *external dynamic one-dimensional range searching* on relational database attribute A, with the corresponding I/O time performance bounds using the B^+-tree on A: (1) Find all tuples such that for their A attribute $a_1 \leq A \leq a_2$. If the output size is K tuples, then this *range query* takes $O(\log_B N + K/B)$ secondary memory accesses, worst-case. If $a_1 = a_2$ then we have a *point query* and if A is a key, i.e., it uniquely identifies the tuple, then this is key-based searching. (2) Inserting or deleting a given tuple takes $O(\log_B N)$ secondary memory accesses, worst-case. The average-case performance for a B^+-tree improves on the above worst-case bounds.

The problem of *external dynamic k-dimensional range searching* on relational database attributes $A_1, \ldots, A_k$ generalizes one-dimensional range searching to k attributes, with range searching on k-dimensional intervals. It is the general data structure problem underlying efficient secondary storage manipulation for many data models (see [12] for a discussion of object-oriented, spatial and constraint models). The problem of k-dimensional range searching in both main memory and secondary memory has been the subject of much research. To date, general solutions approaching the worst-case performance of B-trees for one-dimensional searching have not been found (see below for a brief overview). The basic insight is that this generality is not necessary for class indexing.

The class indexing problem has enough additional structure so that B^+*-tree technology is applicable and high performance is achievable. All that is required is a preprocessing of the class hierarchy for the selection of a particular set of* B^+*-tree indexes. This collection of indexes provides significant query performance gains at small space overhead.* Given the simplicity of the space-time tradeoff involved it is somewhat surprising that our preprocessing step, called *class-division* (CD) here, has not been used before as a practical alternative to (or in addition to) CH.

1.2 Class Indexing and External Dynamic 2D Range Searching

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

Example 1.1 Consider a database that contains information about people such as names and incomes. Let the people objects be organized in a class hierarchy which is a tree with root Person, two children of Person called Professor, Student, and a child of Professor called Assistant-Professor. (See Figure 1.) 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.

Class indexing for this hierarchy means building indexes to answer point and ranges queries and process updates such as: 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.

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 be clear from context (e.g., income). □

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 is essentially the solution that [15] calls class hierarchy index (CH). (Note that, in CH some care goes into the design of the information describing class membership for each key value). This solution has linear space usage and good update performance. However, it cannot compact a K-sized output into K/B pages because the algorithm has no control over how the objects of interest are interspersed with other objects. So its worst-case behavior is unbounded for queries against classes that are not the root of the hierarchy.

Another obvious way to solve the problem is to index the extent of each class in a hierarchy separately. This is called single class indexing (SC) in [15] and would also have good update performance and linear space usage. Moreover, when querying a class, whose full extent spans c' classes it has worst-case query I/O time $O(c' \log_B N + K/B)$. The main problem is that per class query overhead $O(\log_B N)$ is charged for each class searched. This might be reasonable when c' is very small with respect to c, but might dominate the average-case cost otherwise. Indeed, for queries directed against the root class ($c = c'$) [15] show that CH would be better than SC for any nontrivial hierarchies ($c \geq 2$) and a wide range of distributions.

The main disadvantage is that for intermediate situations (e.g., hierachy size 15, number of classes in scope of a range query being 5) both CH and SC are poor alternatives, because SC has too much overhead and CH needs to filter too much data. A naive way around this query performance problem is unrestricted use of space. Keeping a B$^+$-tree per class, indexing the full extent of each single class (FSC), would certainly be wasteful but would address the query performance issue. FSC uses $O((N/B)c)$ pages, has optimal query I/O time $O(\log_B N + K/B)$ and update I/O time $O(c \log_B N)$. The storage and update cost make this unrealistic. However, the following special structure of the problem indicates that there could be other interesting operating points in this space-time tradeoff.

Special Structure: (1) The class hierarchy is a forest and thus it can be mapped in one dimension where subtrees correspond to intervals (see below). (2) The class hierarchy is static, unlike the objects in it which are dynamic.

We now describe how class indexing reduces to external dynamic two-dimensional range searching, with one dimension being static. 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. 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 2 with the root and the range as parameters. At the end of the procedure, every class is associated with a range and every object has a "class" value associated with it. See Figure 1 for an example of the results of applying *label-class* to a class hierarchy. It is easy to see how querying some class over some particular attribute corresponds to two-dimensional range searching. The first dimension of this search is the class attribute and the second dimension is the attribute

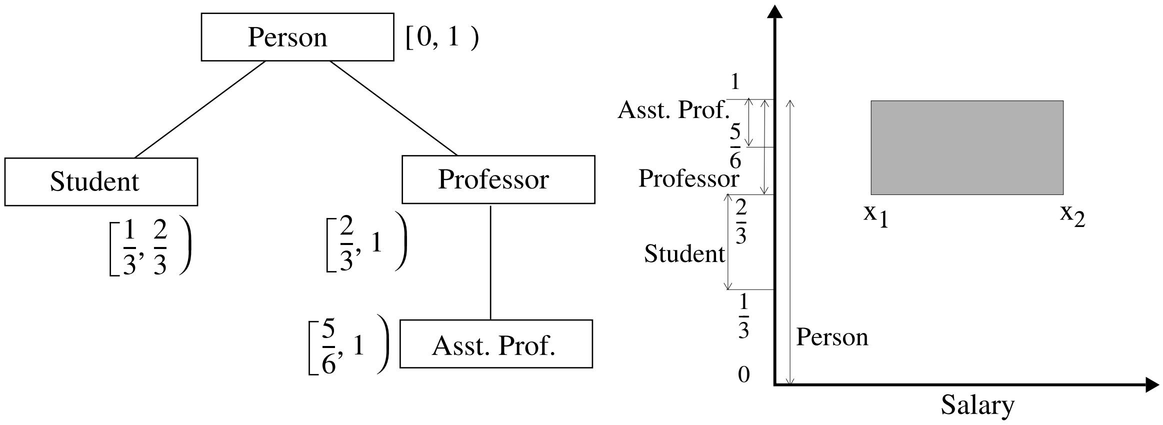

Figure 1: Using *label-class* to reduce indexing classes to two-dimensional range search

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 2: The procedure *label-class*.

over which the search is specified. The range in the class dimension is the range that we associate with the *label-class* algorithm. *The labeling involved is really a pre-order traversal of the class hierarchy.*

Since we assume that the class hierarchy is static, the above reduction is a preprocessing step. In [12], this insight was used to show that: class indexing is in dynamic query I/O time $O(\log_2 c \log_B N + K/B)$ and update I/O time $O(\log_2 c \log_B N)$, using $O((N/B) \log_2 c)$ pages.

In Section 2, we present an optimized implementation of this idea, which we call class-division (CD) and which we evaluate experimentally. In CD, the class hierarchy is preprocessed to select a family **G** of sets of classes. (For example, **G** always includes the set of all classes, which is the set of all subclasses of the root class and the scope of any query against the root). B^+-tree indexes are maintained for the unions of extents of the classes in each set g of **G**. (For example, CH is always one of the indexes maintained, as well as indexes for the leaf classes of the hirearchy). Queries against a class C are processed by accessing a small subset of these indexes appropriate for C. The idea is that if a query is against class C (i.e., its scope includes C's subclasses) the subset of indexes queried *exactly* covers C's subclasses (i.e., their contents are the full extent of C). Updates are processed by changing all replicas of each object. The special structure guarantees existence of a small number of indexes for queries and a small number of replicas for updates.

1.3 Related Work

Class Indexing: The CH method is simple (implementable by B^+-trees), has excellent update performance, space utilization and (in practice) point query performance. So, in spite of possible range query inefficiencies, it is the method of choice in most OODB systems for class indexing.

The CH method clusters objects by key. Other approaches to this problem have clustered objects also by class membership. There have been three such experimental studies [14,18,30]. All involve new data-structures.

The approach [18] is based on the H-tree data-structure. This data-structure threads many B^+-trees together to facilitate simultaneous search. This idea is known as "fractional cascading" in the data structures literature [5] and is notoriously hard to make dynamic. The H-tree scheme, is heuristic. It offers no worst-case performance guarantees for range querying. More importantly, updates are fairly complex and potentially unbounded.

The other two approaches are very recent and do share a number of features. The hcC-tree of [30] balances the CH indexes with indexes on single classes (or equivalently the class extents). This results in a doubling of space. The two kinds of indexes are integrated through the hcC-tree directory. Range searching is improved considerably. The CG-tree of [14] addresses the more general problem of indexing multiple sets. It consists of a special directory on indexes of single classes. Range searching is also improved. To combine it with the benefits of CH, a "grouping by indexed sets" extension is proposed, which is very similar to the hcC-tree. For hcC-tree and CG-tree the worst-case for range searching is similar with single class indexing (SC). However, both approaches are very new and it is unclear whether the performance gains justify a nonstandard data-structure.

Internal 2D Searching: A large literature exists for internal algorithms for two-dimensional range searching. The range tree [3] can be used to solve the problem in $O(N \log_2 N)$ space and static worst-case query time $O(\log_2 N + K)$. By using fractional cascading, we can achieve a worst-case dynamic query time $O(\log_2 N \log_2 \log_2 N + K)$ and update time $O(\log_2 N \log_2 \log_2 N)$ using the same space. We refer the reader to [6] 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 K/B. The special structure of class indexing allows such improvements and is the basic idea behind CD.

Unfortunately, the various general internal algorithms do not map to secondary storage in as smooth a fashion as balanced binary trees map to B^+-trees. For example, [22,29] examine mappings which maintain the logarithmic overheads and make the logarithms base B; however, their algorithms do not compact the K-sized output on K/B pages. Even in some cases when such mappings are possible (see [12]), the large constant factors make them unrealistic. In a sense, this is an algorithmic area where there is significant mismatch between (internal) theory and (external) practice.

External 2D Searching: 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. Examples are the grid-file [20], various quad-trees [25,26], z-orders [21] and other space filling curves, k-d-B-trees [24], hB-trees [17], cell-trees [9], and various R-trees [10,27].

These data structures were designed to have very good average-case behavior for common spatial database problems. However, they are somewhat of an overkill for the simpler problem of class indexing. Correlation between key dimensions does make many of these general techniques degrade. Many involve constant overheads that are far greater than B^+-trees and are more complex to upgrade and use concurrently. To validate this point we have experimented with R-trees.

Finally, these general methods do not provide worst-case guarantees. For a worst-case example, if we assume a uniform two-dimensional grid of points (with a point on each location with integral x and y coordinates) as input, the grid file would produce regions which are of size $O(\sqrt{B}) \times O(\sqrt{B})$. The time to report answers to a two-sided range query would then be (in the worst-case) $O(K/\sqrt{B})$ where K is the number of points in the query result. This is higher than the optimal time of $O(K/B)$ I/O's a B^+-tree would take to report K points. Most data structures in the literature fail to give optimal performance for this very simple example.

1.4 Contributions

We have developed a technique, called indexing by class-division (CD), which we believe can be used as a practical alternative to CH for the class indexing problem. This technique involves an external version of the (folklore [6]) range tree idea, with a number of preprocessing space heuristics, and is described in Section 2. A significant advantage of CD is that it is implementable with standard B^+-trees as an extension of CH. In our implementation of B^+-trees, we used a modified version of the Berkeley DB code [28] that went into the making of the POSTGRES database system [31].

Our contribution is an optimized implementation and experimental validation of CD's average-case performance. The CD method is properly viewed as an extension of CH. Namely, it consists of the index built for the root of the hierarchy (i.e., this is CH) plus some other indexes. Point queries involving two or more classes can be handled by CH. The other indexes can be used for: (1) querying extents, as opposed to full extents, (2) speeding-up range queries. The price to pay for CD is in space/update overhead. We believe that this is acceptable if it is kept low (i.e., within a factor of 2 to 3 of CH) and is balanced by large gains in range query performance.

Our main goal is to determine the speed-up effect for range queries when CD is used instead of CH, so our experiments have focused on this aspect of the problem. Note that, point queries can be handled as in CH and space/update overhead can be easily calculated to a good approximation. The range query speed-up is expressed as the ratio of I/O time by CH over the I/O time by CD, per query answered. This *query efficiency ratio* QE is the quantity measured in our experiments.

The design of our experiments is explained in Section 3. In particular, we did a first phase of experiments with a number of parameters such as page size and buffer size to determine their effect on speed-up. We also provide some experimental evidence that general purpose solutions, such as R-trees , are not competitive for this special problem. Based on these initial experiments we selected "reasonable" values for our second and main phase of experiments.

The main phase of experiments used a number of class hierarchies and three distributions of objects in classes: uniform, skewed and normal. Two additional parameters were explored: the insertion ordering of input data (sorted vs unsorted) and the size of the outputs of range queries (small vs large). The space overhead is a small factor for most reasonable class hierarchy examples and, in worst-case, it is bounded by the depth of the hierarchy and the logarithm of its size.

The results are described in Section 4; see [23] for more details. They offer strong "proof" that we can successfully trade a small amount of storage for good, guaranteed query times. For storage overheads that were generally under a factor of 3, CD offered range query speed-ups with respect to CH at least linear in the number of classes queried for uniformly distributed data and larger otherwise. (The uniform case is the one that most favored CH, but even there CD was advantageous.) More specifically, for a range query on the root class of the hierarchy, CH and CD are roughly equivalent. But for the same range query on a different class c, the CH cost remains constant and the CD cost decreases in proportion to the number of subclasses of c. For example, typical performance improvements were by QE ratios of 8 and for some cases 50. We conclude, with possible extensions of the CD approach, in Section 5.

2 The Class-Division Algorithm

We preprocess a given forest hierarchy H as follows. We use the procedure basic-class-division and the two heuristics (shown in Figure 3) to create a family $\mathbf{G}$ of sets of classes.

After this preprocessing B^+-tree indexes are maintained for the unions of extents of the classes in each member of $\mathbf{G}$. If a range query is against class C a subset of indexes is queried, which *exactly* covers C's subclasses and which involves at most a small number q of indexes. A class can appear in at most a small number r of members of $\mathbf{G}$, so an object (or just its object identity) can have at most r replicas. Updates are processed by changing all replicas.

More formally, preprocessing solves the following combinatorial problem, which we name *class-division of H according to maximal replication factor r and maximal query factor q.*
Input: Forest class hierarchy H with c classes, and positive integers r, q.
Ouput: A family $\mathbf{G}$, whose members are sets of classes from H, such that
(1) No class appears in more than r members of $\mathbf{G}$.
(2) For C any class in H and C^* its set of subclasses in H including C itself, there is a subfamily of $\mathbf{G}$, with at most q members, that exactly covers C^* (the union of at most q members of $\mathbf{G}$ is C^*).

Clearly class-division is possible for $q = c$ and $r = 1$ and $q = 1$ and $r = c$. Interestingly, from the proof of (Lemma 2.3 of [12]) we have the following space-time tradeoff:

Proposition 2.1 For any forest class hierarchy H with c classes, it is possible to do class-divison of H according to $r = \lceil \log_2 c \rceil + 1$ and $q = 2\lceil \log_2 c \rceil$.

Proof: We know that every class query can be represented as a two-dimensional range query, the two dimensions being the attribute range-searched and the class attribute of Figure 2. What basic-class-division essentially does is to build a binary search tree on the class attribute dimension. Its preorder sort is used to make contiguous all classes in a subtree of the hierarchy. It is easy to show, by a geometric cut argument, 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 on the class attribute dimension. That is, one can answer class indexing queries on any class by looking at no more than $2\lceil \log_2 c \rceil$ indexes (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 has more than $\lceil \log_2 c \rceil + 1$ replicas.□

The Heuristics: The basic-class-division procedure provides the guarantees of Proposition 2.1. It is an existence proof for reasonable r, q. In fact, for small hierarchies, solutions to class-division with even smaller r, q can be sought using exhaustive searches.

In practice, we found that solutions to basic-class-division can be greatly improved using the two heuristics of Figure 3. (a) Only some of the internal nodes of the binary tree built by basic-class-division matter. The others can be pruned away. (b) *Raking* bushy subtrees and *contracting* long skinny paths of a tree hierarchy are well-known heuristics for preprocessing trees. We apply rake to eliminate leaves of a tree and contract to eliminate paths, all of whose intermediate nodes have degree 2. An advantage of rake-contract over basic-class-division is that clustering of more than two classes at a time can be controlled, which is useful when a class has many children in the hierarchy tree. *Rake-contract also bounds the replication factor by the depth of the hierarchy.*

We illustrate basic-class-division and these heuristics in Figure 4. We consider the hierarchy H of 7 classes, numbered by preorder in Figure 4 part (a).

The result of basic-class-division with space pruning is in Figure 4 part (b). We use the notation 12 for $1 \cup 2$, i.e., this would correspond to an index on the union of the extents (not the full extents) of classes 1 and 2. Space pruning indicates that indexes need to be keep only for 4,5,6,7 (to answer queries on these classes), for 34 (to answer queries on class 3, by using indexes 34 and 5), for 2 and 56 (to answer queries on class 2, by using indexes 2, 34 and 56), and for 1234567 (to answer queries on class 1–this is equivalent to CH). The replication factor is three and the query factor is three.

The result of rake on the hierarchy is indicated in Figure 4 part (c). The result of the rake-contract heuristic is in Figure 4 part (d). Note that, in this case rake-contract did result in an improvement. As before, we need to be keep indexes for 4,5,6,7 (to answer queries on these classes), for 345 (to answer queries

procedure *basic-class-division* (H,$\mathbf{G}$)

Sort the class hierarchy H in **preorder**.
Let result of sort be $C_1, C_2, \ldots, C_c$.
Let $C_1^1, C_2^1, \ldots, C_c^1$ be the 1-level family of sets of classes
with $C_i^1 = \{C_i\}$ for $1 \le i \le c$.
for i from 1 to $\lceil \log_2 c \rceil + 1$ **do**
Create a family of sets of classes $C_1^i, C_2^i, \ldots$ at i level
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 sets in family, let $C_{l_i}^i$, the last set at round i be
$C_{l_{i+1}}^{i+1}$, the last collection at round $i+1$.
endfor
Family of sets of classes constructed at all levels forms $\mathbf{G}$.
The levels give this family a binary tree structure.

endproc

Prune-Space Heuristic: (H,$\mathbf{G}$)
Construct $\mathbf{G}$ by basic-class-division.
For each class C in hierarchy H, cover the set of C's subclasses in H
with maximal subtrees in $\mathbf{G}$'s binary tree structure, so the cover is exact.
Remove any sets in $\mathbf{G}$ not used in covers.
Let $\mathbf{G}$ be the resulting family of sets of classes.

Rake-Contract Heuristic: (H,$\mathbf{G}$)
Construct $\mathbf{G}$ by basic-class-division with space pruning.
Build a family of sets of classes $\mathbf{J}$ by using rake and contract on H.
Fully replicate on raking leaves.
Use basic-class-division with space pruning for contracting paths.
. Alternate rake and contract steps until the hierarchy is fully processed.
If $\mathbf{J}$ improves replication or query factor then replace $\mathbf{G}$ by $\mathbf{J}$.
Let $\mathbf{G}$ be the resulting family of sets of classes.

Figure 3: The preprocessing procedure *class-division*.

on class 3), for 26 (to answer queries on class 2, by using indexes 26 and 345), and for 1234567 (to answer queries on class 1). The replication factor is again three and the query factor is improved to two.

CD as an extension of CH: (1) $\mathbf{G}$ always includes the set of all classes, so CH is one of the indexes maintained, with leaf structures as in [15] . In our implementation of CD, all *point queries involving two or more class extents* are handled by CH. Point queries typically cost the height of an index, which for practical purposes will always be a small constant, e.g., 2 to 3 I/Os.
(2) $\mathbf{G}$ always includes indexes for the leaf classes of the hierarchy. These will be used for *point and range queries for leaf classes*, i.e., where extents and full extents are the same. Note that, the [30] data-structure has provision for querying both full extents and extents. The equivalent task here would involve adding indexes for the extents of all classes.
(3) In CD, unlike CH, *range queries of any class C* are handled in CD using the small number of indexes covering C's full extent. This is the main difference and the subject of our experiments.
(4) In CD *updates* are handled by processing all replicas. Replication numbers, space usage and update performance can all be estimated fairly closely by direct calculation.
(5) CD is implementable using standard B$^+$-trees. Although we do not examine this issue here, *concurrent*

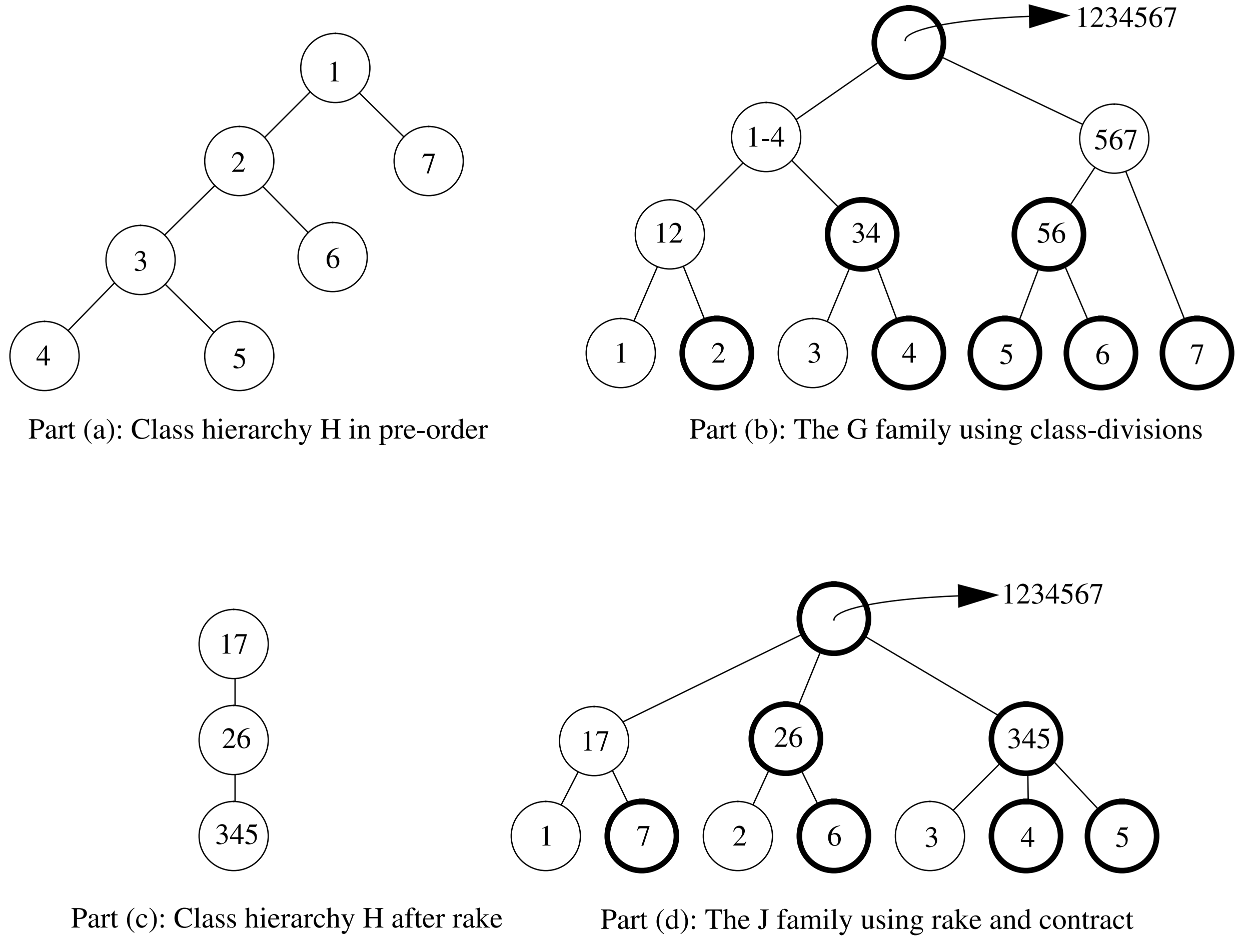

Figure 4: Illustrating class-division.

use of the indexes can be performed using standard locking techniques for B^+-trees. Consistency of the small number of replicas can be maintained using small-sized transactions.

3 The Experiments

3.1 Methodology

The experiments were conducted on Sparc 10 workstations running SunOS 4.1.3. A modified version of the Berkeley DB code [28] that went into the making of the POSTGRES database system [31] was used to implement B^+-trees. The code builds B^+-trees on disk using the OS-level file system. The Berkeley DB code runs at the user level, does its own buffer management, and exercises no control over the virtual memory and paging activity of the operating system. Therefore, in order to count I/Os, we restricted ourselves to measuring the explicit read requests issued by the program. In other words, implicit reads generated by the operating system pager did not figure in the statistics collected.

The example class hierarchies in the experiments were selected to be a mixture of hierarchies that were difficult to index and typical examples from the literature. (For example, some of them came from [1].)

The class-division algorithm was used to create collections that were then indexed using B^+-trees. When populating class hierarchies with objects several simplifying assumptions were made:

1. It was assumed that the objects were small. (The objects used in our experiments were always less than 100 bytes long.) This is realistic in many primary index situations and also if object identities are used for secondary indexes. Also, large objects, typically of size greater than the page size of the underlying secondary storage, pose difficult problems that are orthogonal to the ones being considered here.

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

3. In experiments involving uniform distributions, it was assumed that the keys of the objects were uniformly distributed over a range. Queries were also assumed to be uniformly distributed over the range of the keys as well as over the different classes in the class hierarchy.

4. In experiments involving skewed distributions, objects in a particular class had keys uniformly distributed over a range, but these ranges were different for different classes. Some of these ranges overlapped, but the overlap was always limited to 50% of the range. Queries were uniformly distributed over the union of the ranges.

5. In experiments involving normal distributions, objects in a particular class had keys distributed in a gaussian distribution around a mean. The means of the different classes were staggered so that there was overlap between the various classes. The starting point of the queries were uniformly distributed over the entire range but their width was made a normally distributed variable. Because of the normal distribution of the key values and the width of the queries, there was no easy way to tell if a query was going to be "small" or "large". The graphs for these experiments will not show this distinction.

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

- data size. This is the number of objects that belong to the individual extent of a class.
- ordering of input data. The Berkeley DB code for B^+-trees has a hook that optimizes the space utilization of the B^+-trees produced when the input data is given in sorted order. Both sorted and unsorted input data are realistic situations, since occasional reorganizations in a database can be considered similar to presenting sorted input to the B-tree algorithm.
- buffer size. Typical values range from 100 kilobytes to 5 megabytes.
- page size (or disk block size). Typical values for the page size range from 512 bytes to 64 kilobytes.
- query size. Queries can be either restricted to retrieve only a small portion of the database (typically $< 10\%$) or totally unrestricted, possibly retrieving the entire database.
- number of queries. This is the number of queries the algorithms are asked to answer in one experiment.

The experiments were performed as follows. First, a class hierarchy was selected for the experiment and values were chosen for each of the variables that could be controlled. Input data was then generated and the B^+-trees required by the CH and the CD techniques were created. A query file was then produced. (We usually generated a set of 2000 queries.) One process was started to answer queries using CH and after it finished, another was started to answer queries using CD. These processes had identical parameters. Queries from the query file were answered one by one and the number of disk I/Os requested for each query was recorded in a file. After this concluded, the output files were processed to compare the number of I/Os each method took for the queries. Since all the test conditions were identical for the two processes, this was a fair comparison. The ratio of the I/Os for CH and for CD was computed for each query answered. This ratio, called the *query efficiency ratio (QE ratio)*, was the most important statistic derived from the experiments.

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

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

The QE ratio depends on the experimental conditions under which queries are answered. The other parameters can be estimated quite closely by looking at the indexes needed for CD on the target class hierarchy. Consequently, the QE ratio is the statistic we present graphically and the other parameters are computed and presented whenever we present a class hierarchy.

We conducted two preliminary sets of experiments to select some parameters and check whether a general purpose 2D method could be used instead of CD.

3.2 Selecting Page and Buffer Size

In the first experiments, hierarchy H1 in Figure 5 was used to study the effect of buffer pool size and page size on the QE ratio. A variety of values were assigned to data size, buffer pool size, and page size.

Further, both sorted and unsorted inputs as well as small and large queries were considered (for a total of 2400 test runs, answering $48,000$ queries and performing many gigabytes of disk activity). In summary, the QE ratio is not greatly affected by the small/large nature of the queries or the sorted/unsorted nature of the input. It can be observed, however, that the QE ratio is somewhat larger for large queries than for small queries. Similarly, it is a little larger for unsorted input than sorted input.

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

Example 3.1 Consider hierarchy H1 in Figure 5. The class-division algorithm proceeds by first indexing c_1. c_1 and c_2 are combined to give C_2, which is then indexed. c_3 is indexed next. Finally, all the individual extents are combined to give C_0 (which is the same as C_4), which is also indexed.

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

The total storage used for the indexes is:

$$(3c_1 + 2c_2 + 2c_3 + c_4)/S$$

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

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

Figure 6 presents graphs that plot the QE ratio, averaged over all queries, against page size and buffer pool size. In the experiments generating these graphs, the size of each individual class extent was 20,000 objects, and 2000 queries uniformly distributed over the range of the key and the four classes in hierarchy H1 were used. Here we present only a sample of the results (see [23] for the details). The experiments we did differed in the nature of the queries asked (small or large) and the nature of the input used to construct the B$^+$-trees (sorted or unsorted input).

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

The plots in Figure 6 also show that the QE ratio tends to increase with buffer pool size. The reason for this is that, once a significant portion of the database fits into memory, the CD algorithm performs extremely well when queries fall nearby, and this is bound to happen when a large number of queries are asked. The CH method, on the other hand, cannot take advantage of the large buffer pool size because it has no control over how data from different classes are interspersed together. The QE ratio shows little or no change with increasing page size, except when the page size becomes so large that no more than a few pages can fit into the buffer pool. At this point, the QE ratio drops slightly but quickly increases again as the size of the buffer pool increases.

In the main set of experiments, which were performed on larger class hierarchies "moderate" values were chosen for data size, buffer pool size and page size and the QE ratio was computed while varying the other parameters. The page size was set to 4096 bytes, and the buffer pool size was set to 500 kilobytes.

The top part of Figure 7 is a typical graphical presentation of speed-up (in this case for hierarchy H2) that we use in the main experiments. On the left we plot the QE ratio on a per class basis. (That is, the ratio is averaged over each class separately rather than over the entire hierarchy.) On the right we present the averages over all the classes with the same number of subclasses. This allows us to plot QE by number of classes queried. In these experiments, the size of each individual class extent was set to 10,000 objects. We generated a set of 2000 queries for each run and conducted the experiments. We plot four separate lines on each graph to get an understanding of the effect of sorted input and query size on the QE ratio.

3.3 R-Trees and Class-Division

In a second preliminary experiment we compared the use of R-trees to that of CD. Our comparison was on hierarchy H2. R-trees work by dividing space into rectangular areas. They perform well if they are able to generate rectangles that do not intersect each other much. In the class indexing problem, one of the dimensions is of very low cardinality compared to the other. These experiments confirmed the intuition that it is very difficult to produce good partitions in this case. (We used Guttman's [10] quadratic split algorithm to process node splits.)

The speed-up ratios in Figure 7 indicate that CD outperforms R-trees by a factor of over 30, even when the class hierarchy size is 6. Surprisingly, R-tree performance is much worse for small queries when compared

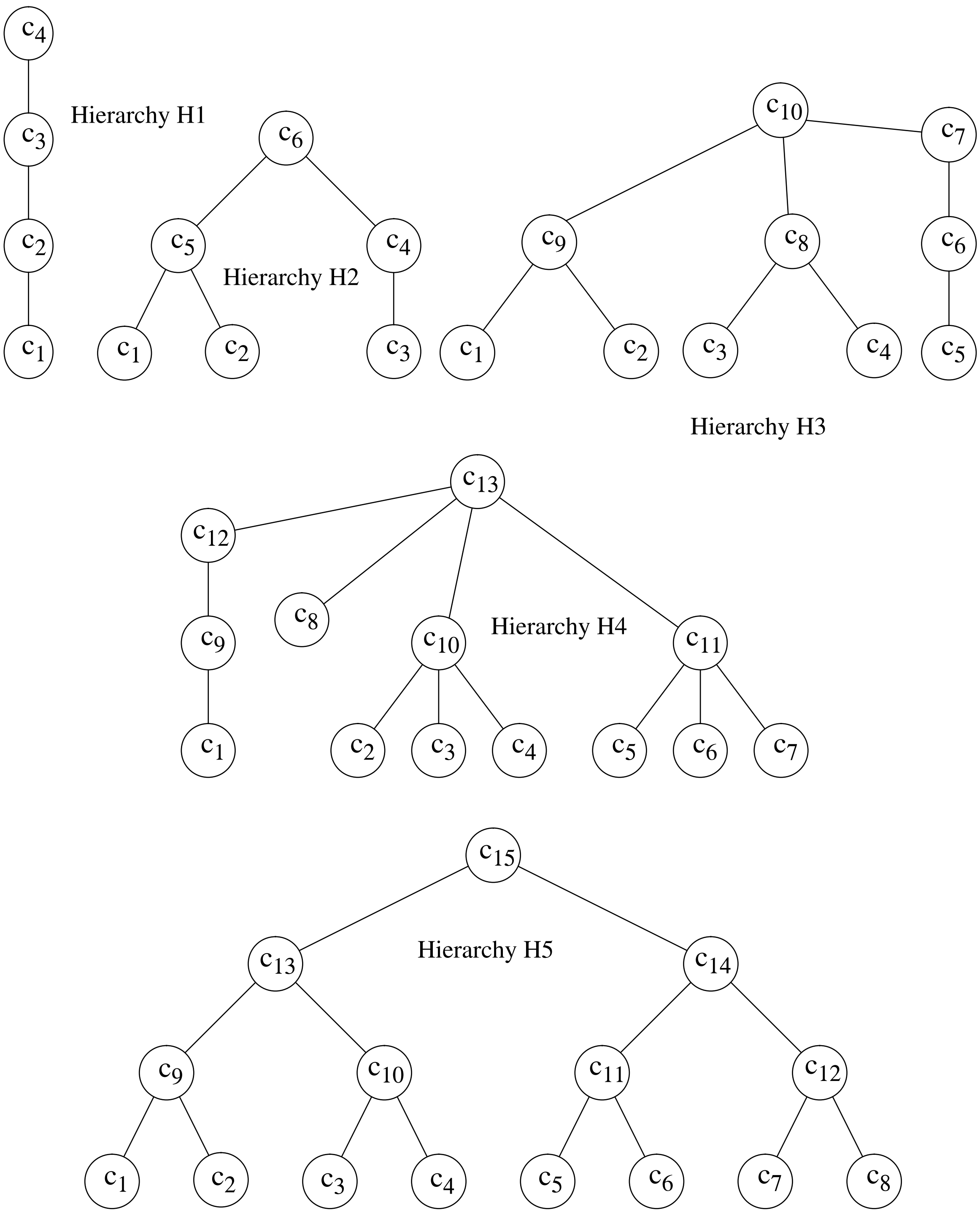

Figure 5: The hierarchies used in the experiments. A 63 class complete binary tree hierarchy used in the experiments, called H6, is not shown here.

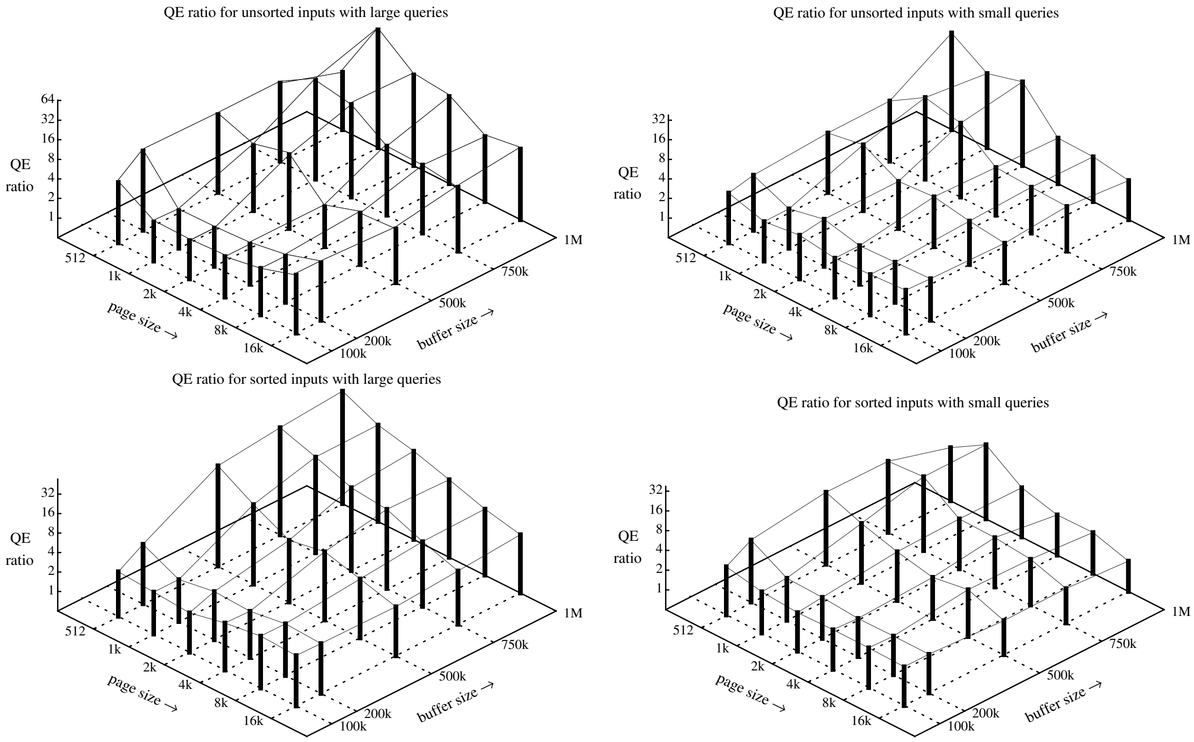

Figure 6: Four plots of QE ratio against page size and buffer pool size. In these experiments, on hierarchy H1, each class had 20,000 objects. The four plots differ in the nature of the input used to construct the btrees (sorted input gives better storage utilization), and on the nature of the queries asked (small queries retrieved less than 10% of the database, while large queries were unrestricted).

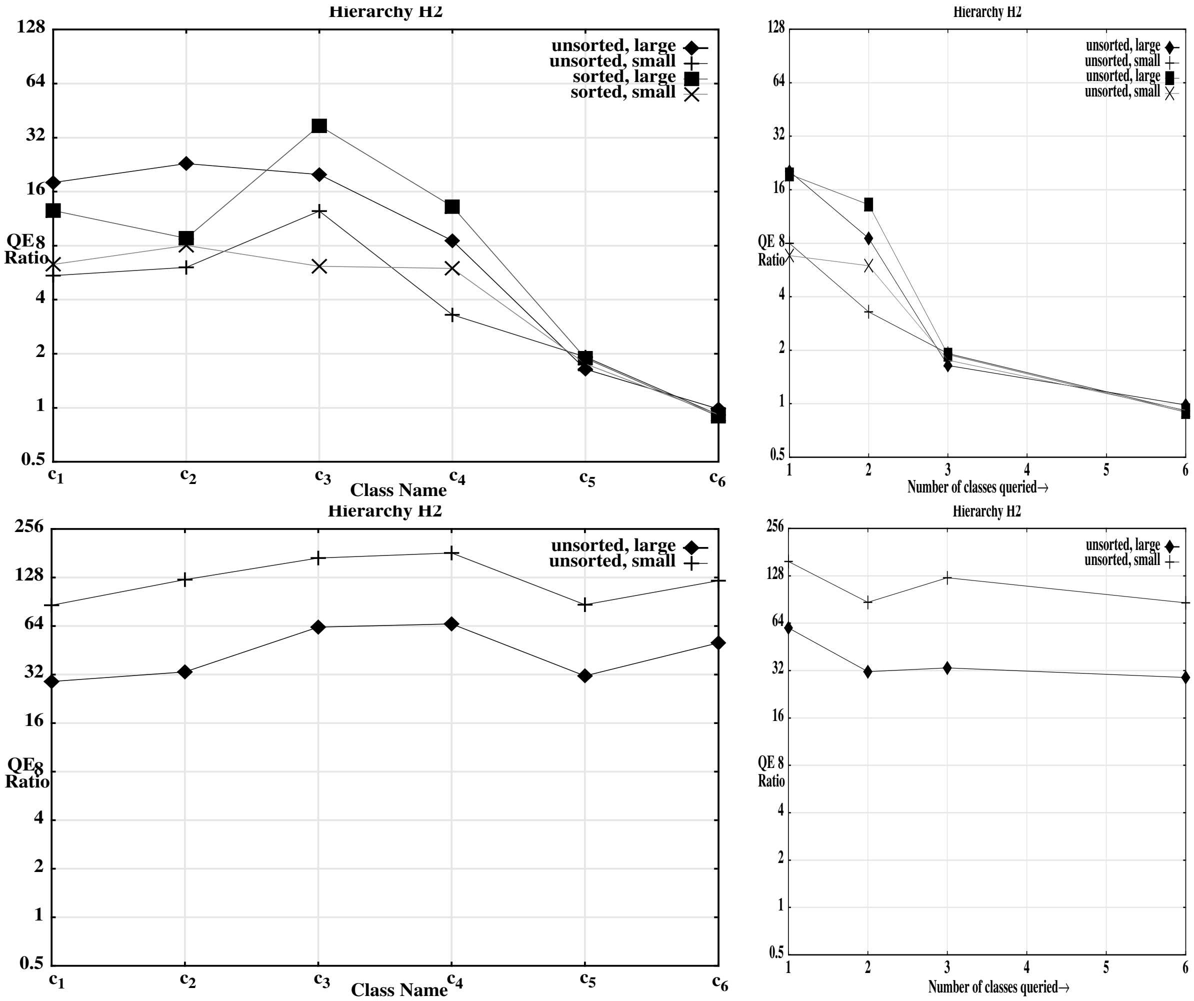

Figure 7: Plots comparing the performance of CH and R-trees against CD. The top half of the figure compares the performance of CH against CD, and the bottom half compares the performance of R-trees against CD. In these experiments, conducted on hierarchy H2, each class had 10,000 objects, and 2000 uniformly distributed queries were used.

to large queries. A closer analysis revealed that R-trees retrieved a significant portion of the database for even very small queries, which is a good indication that its partitions overlapped very heavily. The CD, using B^+-trees looked at very few pages to answer small queries. (The number of wasted I/O's in a search performed by CD is limited by the maximal query factor for the hierarchy being indexed. This factor was never more than 2 in our examples. Subsequently, CD never performed more than 4 wasted I/O's for any query.)

Example 3.2 Hierarchy H2 (in Figure 5) is a small hierarchy of 6 classes. c_1, c_2, c_3 and c_4 are indexed first. This is followed by the creation and indexing of C_5 and C_6. Under this scheme, every class except c_4 has its complete extent indexed. Queries on c_4 are answered using the indexes on c_3 and c_4.

The storage overhead factor is

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

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

4 Experimental Evaluation of QE Ratio

We now present some of the results of our main experiments. For this abstract, we use three typical hierarchies of size in the range 10-15. We also present results from a 63 class hierarchy, with unfavorable storage overhead for CD.

We can see from these examples that with relatively small overheads in space, it is possible to index class hierarchies so that queries on them can be answered extremely efficiently.

Example 4.1 Hierarchy H3 (in Figure 5) combines a "long, skinny" component with a "bushy" one. $c_1, c_2, \ldots, c_6$ are indexed separately, and after this C_7, C_8, C_9 and C_0 (C_{10}) are created in the usual manner and indexed. By doing this, every class other than c_6 has its full extent indexed. Queries on c_6 are thus answered by looking at the indexes on c_5 and c_6.

The storage overhead factor is

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

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

Example 4.2 Hierarchy H4 (in Figure 5) is similar to H3 in its combination of "skinny" and "bushy" hierarchies. As in the previous example, $c_1, c_2, \ldots, c_8$ are indexed separately. This is followed by the creation and indexing of C_{10}, C_{11}, C_{12} and C_0 (C_{13}). All classes except c_9 have their full extents indexed as a separate collection. Queries on c_9 are answered using the indexes on c_1 and c_9.

The storage overhead factor is

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

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

Example 4.3 Hierarchy H5 (in Figure 5) is a complete binary tree of height 3. $c_1, c_2, \ldots, c_8$ are indexed first using their individual extents. C_9 is obtained by combining c_1, c_2 and c_9, C_{10} by combining c_3, c_4 and c_{10}, C_{11} by combining c_5, c_6 and c_{11}, and C_{12} by combining c_7, c_8 and c_{12}. C_{13} is obtained by combining C_9, C_{10} and c_{13}, and C_{14} by combining C_{11}, C_{12} and c_{14}. Finally, C_0 (C_{15}) is obtained by combining C_{13}, C_{14} and c_{15}. $C_9, C_{10}, \ldots, C_{15}$ are then indexed.

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

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

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

Figure 8 present typical speed-up results for the uniform distribution. On the left we plot the QE ratio on a per class basis. (That is, the ratio is averaged over each class separately rather than over the entire hierarchy.) On the right we present the averages over all the classes with the same number of subclasses. This allows us to plot QE by number of classes queried. In these experiments, the size of each individual class extent was set to $10,000$ objects. We generated a set of 2000 queries for each run and conducted the experiments. We plot four separate lines on each graph to get an understanding of the effect of sorted input and query size on the QE ratio.

For the uniform distribution we also present (the right part of the plot) for a hierarchy of 63 classes H6 that is a complete binary tree of depth 6. It is indexed by creating an index on the complete extent of every interior node. For this hierarchy, the maximal replication factor was 6, the maximal query factor 1 and the storage overhead factor 5.1.

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

In Figure 10 and Figure 11 we present the analogous statistics gathered from skewed and normal distributions respectively. Clearly, these cases favor CD even more than the uniform cases.

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

5 Conclusions and Open Problems

We have developed a practical technique, called indexing by class-division (CD), which can be used to significantly improve the class hierarchy index (CH) method of [15]. We present an optimized implementation and experimental validation of CD's average-case performance. The results of our experiments show the superior performance of CD over CH for range searching, given a small space tradeoff.

There are a number of possible extensions of the investigation presented here. We would like to highlight the following three. (1) The preprocessing of the hierarchy could use a priori estimates of class extent sizes. This could lead to a tuning of CD. (2) Concurrency control and recovery issues are important in any realistic setting. A better understanding is necessary of CD's interaction with the lock manager. (3) Our technique can be extended to handle restricted forms of multiple inheritance where a class can inherit from only a small number of superclasses. In this case, a class indexing query is reduced to a small number of range queries. It is not clear that the method will be as efficient as CD. (4) Both CH and CD assume that the hierarchy is static. The design of leaf records depends on this in CH, as well as CD's preprocessing. Adapting to limited hierarchy changes is important and an interesting issue for further study.

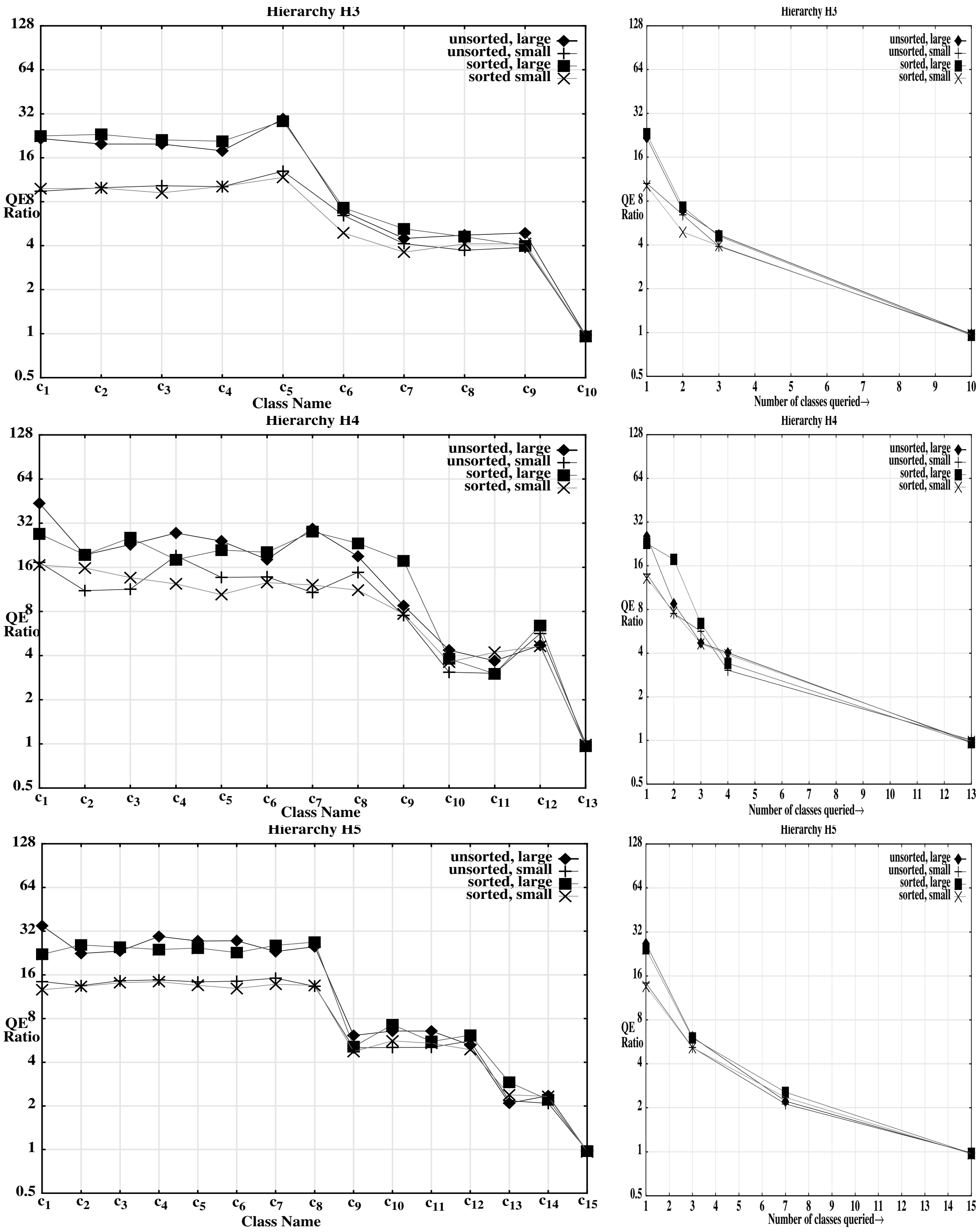

Figure 8: Plots of QE Ratio for class hierarchies H3, H4 and H5 for the uniform distribution of objects. The plots on the left plot QE ratio on a per-class basis, while the plots on the right plot QE ratio against the size of the queried classes. In these experiments, each class was populated with 10,000 objects distributed uniformly in the key domain.

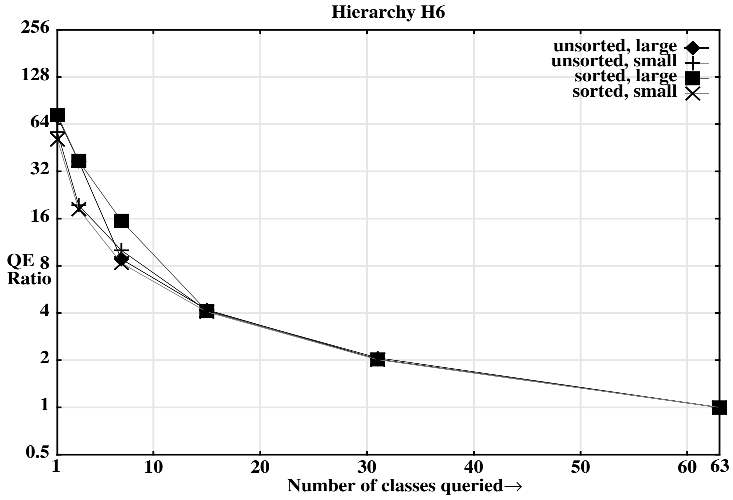

Figure 9: Plot of QE ratio against the size of the queried class for hierarchy H6, a complete binary tree with 63 nodes. In the experiments for this class, each class was populated with 10,000 objects distributed uniformly in the key domain.

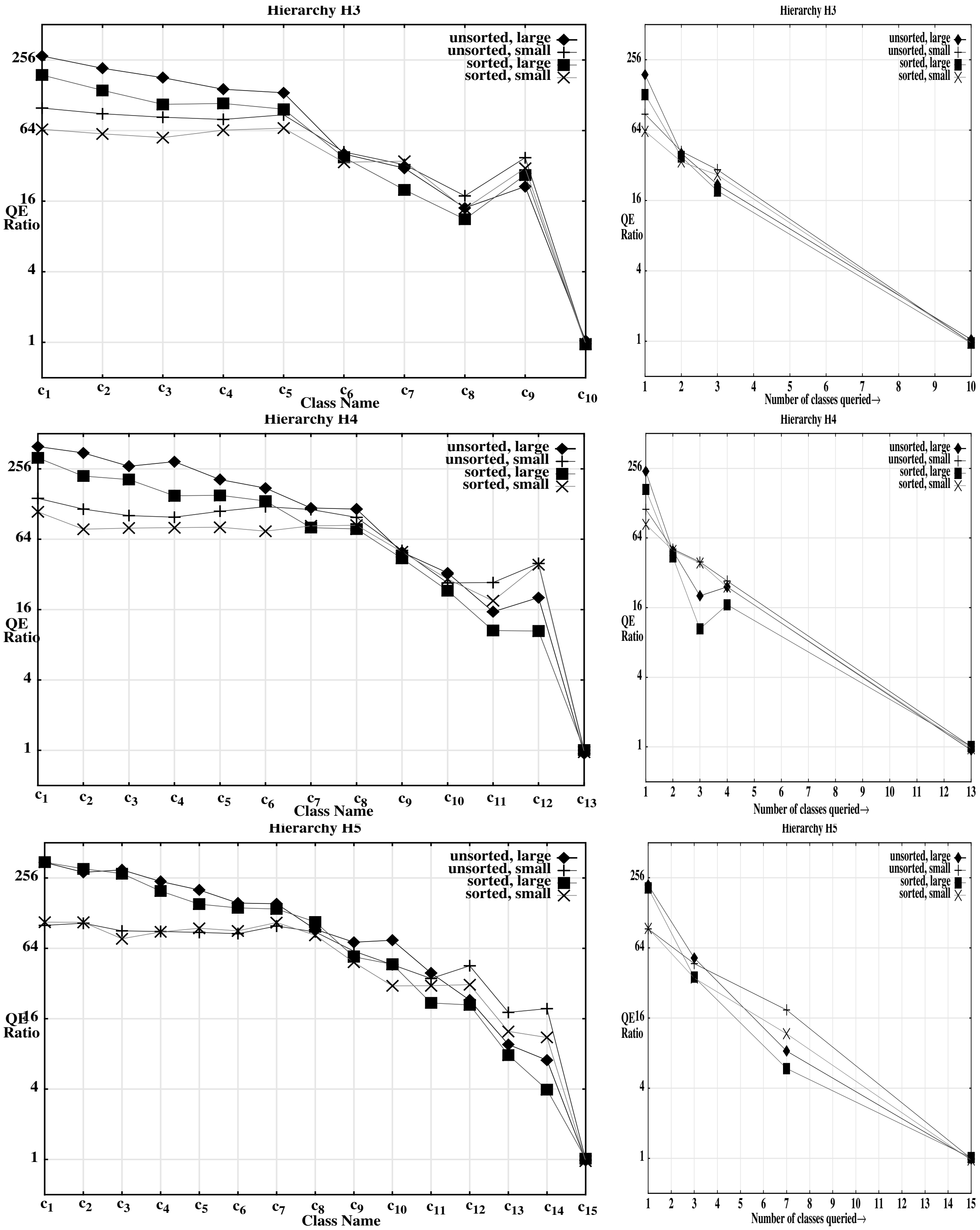

Figure 10: Plots of QE Ratio for class hierarchies H3, H4 and H5 for the skewed distribution of object keys. The plots on the left plot QE ratio on a per-class basis, while the plots on the right plot QE ratio against the size of the queried classes. In these experiments, each class was populated with 10,000 objects. The key values were uniformly distributed over a range, but ranges were different for different classes and partially overlapped with each other.

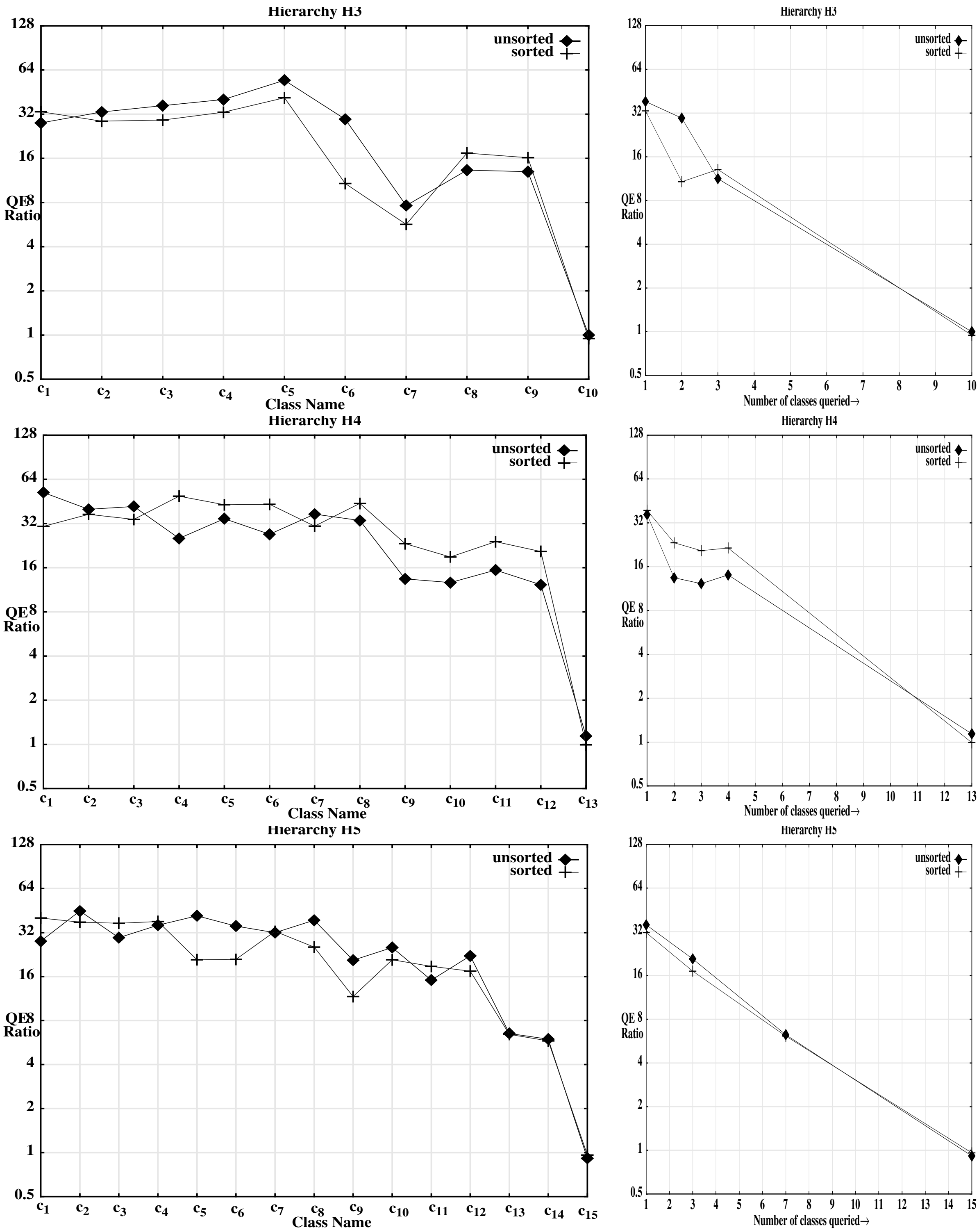

Figure 11: Plots of QE Ratio for class hierarchies H3, H4 and H5 for the normal distribution of object keys. The plots on the left plot QE ratio on a per-class basis, while the plots on the right plot QE ratio against the size of the queried classes. In these experiments, each class was populated with 10,000 objects following a staggered normal distribution in the key domain.

References

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

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

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

[4] E. Bertino and W. Kim, "Indexing techniques for queries on nested objects," *IEEE Transactions on Knowledge and Data Engineering* 1(2) (1989).

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

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

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

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

[9] 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.

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

[11] Y. Ishikawa, H. Kitagawa, and N. Ohbo, "Evaluation of signature files as set access facilities on oodbs," *Proc. of the ACM SIGMOD* (1993).

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

[13] A. Kemper and G. Moerkette, "Access support in Object Bases," *Proc. of the ACM SIGMOD* (1990).

[14] C. Kilger and G. Moerkotte, "Indexing Multiple Sets," *Proc. 1994 VLDB Conference*.

[15] 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.

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

[17] 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.

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

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

[20] 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.

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

[22] 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.

[23] S. Ramaswamy, "Indexing for Data Models with Classes and Constraints," Department of Computer Science, Brown University, Ph.D. Thesis, 1994.

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

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

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

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

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

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

[30] B. Sreenath and S. Seshadri, "The hcC-tree: An Efficient Index Structure For Object Oriented Databases," *Proc. 1994 VLDB Conference*.

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

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