The VLDB Journal (1996) 5: 215–228

The VLDB Journal
© Springer-Verlag 1996

Index nesting – an efficient approach to indexing in object-oriented databases

Beng Chin Ooi[1], **Jiawei Han**[2], **Hongjun Lu**[1], **Kian Lee Tan**[1]

[1] Department of Information Systems and Computer Science, National University of Singapore, Kent Ridge, Singapore 119260; e-mail: {ooibc,luhj,tankl}@iscs.nus.sg
[2] School of Computing Science, Simon Fraser University, British Columbia, Canada V5A 1S6; e-mail: han@cs.sfu.ca

Edited by Ron Sacks-Davis. Received December 1992 / Revised May 1994 / Accepted May 1995

Abstract. In object-oriented database systems where the concept of the superclass-subclass is supported, an instance of a subclass is also an instance of its superclass. Consequently, the access scope of a query against a class in general includes the access scope of all its subclasses, unless specified otherwise. An index to support superclass-subclass relationship efficiently must provide efficient associative retrievals of objects from a single class or from several classes in a class hierarchy. This paper presents an efficient index called the hierarchical tree (the H-tree). For each class, an H-tree is maintained, allowing efficient search on a single class. These H-trees are appropriately linked to capture the superclass-subclass relationships, thus allowing efficient retrievals of instances from a class hierarchy. Both experimental and analytical results indicate that the H-tree is an efficient indexing structure.

Key words: OODB – Indexing structures – Query retrieval

1 Introduction

Object-oriented database (OODB) systems emerged as a response to the requirements of new applications which cannot be efficiently supported by conventional database systems. One of the major concepts supported in OODB is the notion of generalization/specialization. A class in OODB can be specialized into a number of subclasses. The impact of such specialization on the semantics of object instantiation is that the access scope of a query against a class may be the instances of that class or instances of all classes in the class hierarchy rooted at that class. To support the superclass-subclass relationships efficiently, an associative search index must facilitate (1) efficient retrieval of instances from a single class, and (2) efficient retrieval of instances from classes in a class hierarchy.

In this paper, we study the implication of specialization and present an index that facilitates query retrievals based on superclass-subclass relationships. Based on the B^{+}-tree, our new associative search index is called the hierarchical tree (H-tree; Low et al. 1992). An H-tree structure is maintained for each class of a class hierarchy and these trees are nested according to their superclass-subclass relationships. When indexing an attribute, the H-tree of the root class of a class hierarchy is nested with the H-trees of all its immediate subclasses, and the H-trees of the subclasses are nested with H-trees of their respective subclasses and so forth. Indexing in this manner forms a hierarchy of index trees. Nesting H-trees supports efficient traversal of the nested H-trees (of subclasses) by enabling traversal of a nested H-tree to start at appropriate subtrees via the links maintained in its superclass's H-tree. In addition, a nested H-tree can also be accessed independent of its superclass's H-tree. Note that a queried class does not have to be the root class of the class hierarchy and therefore searching for instances within a subhierarchy of classes can start at any class as long as they are indexed on the same attribute. The nested organization provides a natural and efficient support for superclass-subclass relationships. The H-tree organization naturally lends itself to indexing in recursive query processing using semi-naive evaluations (Low et al. 1993) and indexing multiple sets (Kilger and Moerkotte 1994). We implemented H-trees and compared their performance with class hierarchy trees (CH-trees; Kim et al. 1989). Both the experimental and analytical results indicate that the H-tree is an efficient indexing structure.

The remainder of this paper is organized as follows. In Sect. 2, the problem of indexing in OODB is further discussed and related work reviewed. In Sect. 3 the data structure and nesting organization of the H-tree indexes are described. The algorithms for searching, insertion and deletion are presented in Sect. 4. Both analytical and empirical results on the performance of H-trees are presented in Sect. 5. Conclusions and future directions are presented in Sect. 6.

2 Motivation and related work

Object-oriented databases provide new kinds of data semantics, such as inheritance and superclass-subclass relationships. An instance of a subclass is also an instance of its superclass. As a result, the access scope of a query against a

The work was in part supported by NUS Research Grant RP910654

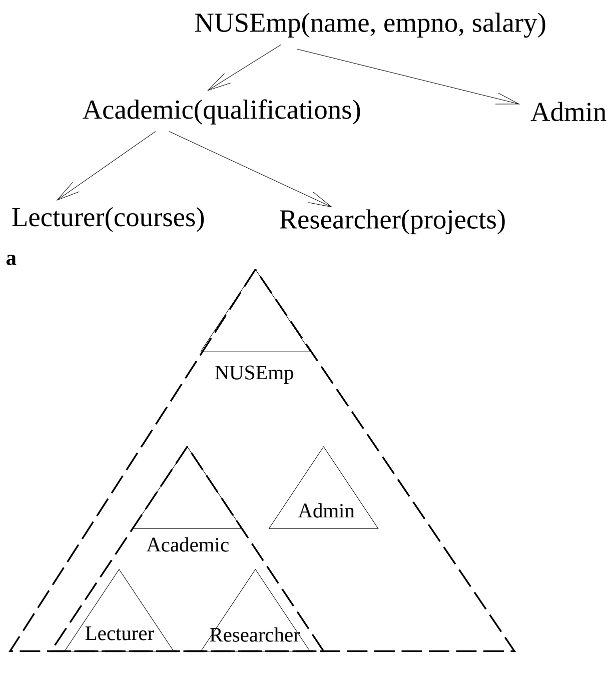


Fig. 1a,b. Access scope. **a** A class hierarchy of employees. **b** The search space of classes

class generally includes not only its instances but also those of all its subclasses. A query may also be formulated explicitly against a class and some of its subclasses. Indexes are necessary to speed up the associative search. In order to support the superclass-subclass relationship efficiently, the index must achieve two objectives. First, the index must support efficient retrieval of instances from a single class. Such a retrieval is similar to that of relational DBMS. Second, it must also support efficient retrieval of instances from a number of classes in a hierarchy of classes. Consider the class hierarchy in Fig. 1a. The NUSEmp is the root class of the class hierarchy and the superclass of Academic and Admin, and Academic in turn is the superclass of Lecturer and Researcher. Attributes in a superclass are inherited by all its subclasses. For example, the attributes *name*, *empno* and *salary* in NUSEmp are inherited by all the subclasses in the class hierarchy. We will use the term *common attributes* to refer to attributes inherited by all the classes in the class hierarchy. An associative query against class NUSEmp on one of its attributes implicitly includes its subclasses, Admin and Academic, and those of Academic, Researcher and Lecturer. Figure 1b describes the search space of each class.

Suppose we wish to index the common attribute, say *salary*. Ideally, the indexing scheme must support efficient retrieval of the following:

1. Instances of a particular class not including its subclasses. For example, a list of all academic employees who are neither lecturers nor researchers and who earn more than \$60 000 and a list all lecturers who earn more than \$40 000.
2. Instances of a class and all its subclasses. For example, a list of the employees (NUSEmp) who earn more than \$30 000.

Based on B^+-trees, Kim et al. (1989) proposed an indexing scheme called the CH-tree. To index a hierarchy of classes on a common attribute, typically one of the superclass attributes, a CH-tree maintains only one index tree for all the classes of the hierarchy. A search on a class for instances that satisfy the associative search condition is performed as if the index is maintained solely for that class. Instances of classes of no interest to the answer are discarded. As a result, the search for instances of a small number of classes may not be efficient. The structure of the CH-tree is shown in Fig. 2. While internal nodes are similar to those of B^+-trees, leaf nodes contain key values and associated directories. In a leaf node, for each key value, object identities (oids) of objects which have the same indexed value are grouped in a directory based upon objects' classes.

The performance study conducted shows that the indexing scheme of one index for all classes in a class hierarchy performs better than the indexing scheme that supports one index for each class. However, a major drawback of the CH-tree is that it does not support the superclass-subclass relationship naturally. Searching for values in a single class is treated in the same way as searching for values in a hierarchy of classes. In other words, the same searching strategy is used for retrieving values in a single class as well as in a hierarchy of classes.

In the work of Scheuermann and Ouksel (1982), a different kind of index nesting, the multi-dimensional B-tree (MDBT), was proposed for multi-attribute indexing. In an MDBT (Scheuermann and Ouksel 1982; Kriegel 1984), a B-tree is constructed for the first indexed attribute, and for each attribute value, a B-tree may be attached for indexing on the second indexed attribute and so forth. Hence, the number of B-trees can be very large, and B-trees maintained for the same attribute are not related.

Several other indexing structures were proposed (Bertina and Kim 1989; Kemper and Moerkotte 1990; Maier and Stein 1986; Valduriez et al. 1986). However, these indexes mainly deal with path indexing for nested objects in OODB.

The indexing mechanism that we proposed in previous work (Low et al. 1992) and further studied in this paper is designed to support retrievals of instances from a class or a hierarchy of classes. The index, called the H-tree, is based on the superclass-subclass relationship. Moreover, unlike the work of Scheuermann and Ouksel (1982), indexes are nested on the same attribute. While the CH-tree partitions the data space based on attributes, the H-tree organizes the data space into classes. As a result, it provides superior performance for range queries on a single class hierarchy, which the CH-tree is unable to provide. Consider, for example, the partitioning of the data space (i.e., the entries in the leaf nodes of the index structure) shown in Fig. 3 (Chan and Ooi 1994) for the H-tree and CH-tree. For a single-class query with a wide attribute value range (represented by the horizontal search area), it is efficiently supported by the H-tree index (Fig. 3a) as it requires only a partial sequential scan of the leaf nodes of a single-class index. However, this query is not well supported by the CH-tree (Fig. 3b) because it involves

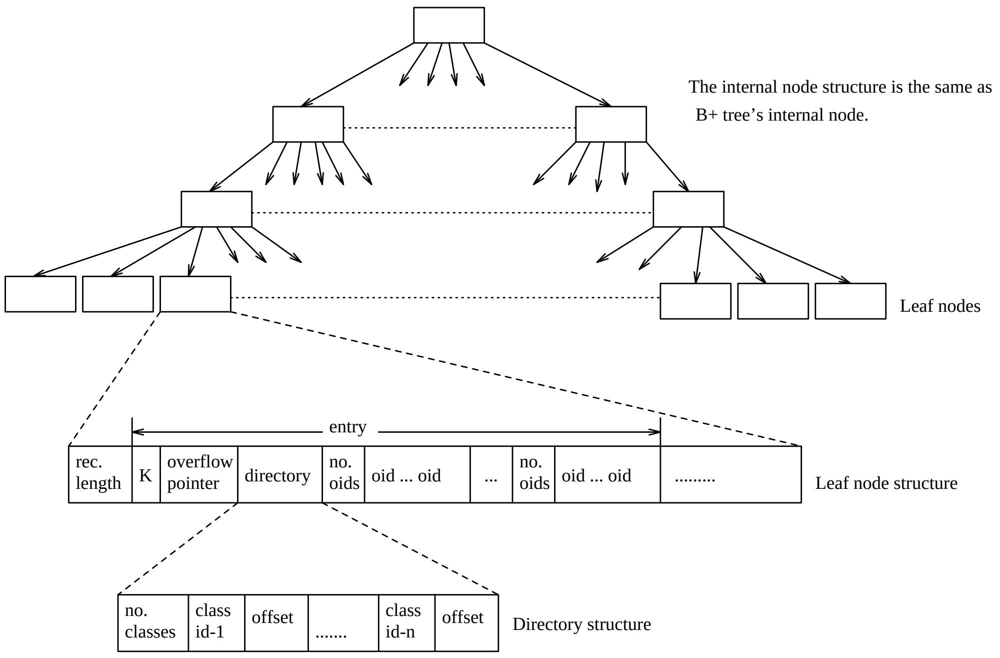


Fig. 2. The class hierarchy tree (CH-tree) structures

a long sequential scan of a single large index which is likely to access many irrelevant leaf nodes. For a class hierarchy query with a narrow attribute value range (represented by the vertical search area), it is very efficiently supported by the CH-tree (Fig. 3b) because it requires a partial sequential scan of the B^{+}-tree index. While it requires traversals of multiple single-class indexes, the H-tree is also able to deliver good performance. This is because of the nesting of classes that facilitates efficient traversal between the single-class indexes. Thus, the H-tree achieves the speed of the one-index one-class scheme for single-class retrievals and one-index all-class scheme (cf. CH-trees) for multiple-class retrievals.

3 The H-tree organization

In this section, we describe the structure of the H-tree and the nesting of H-trees. We use H_c to denote the H-tree of class c. To index the classes in Fig. 1 on a class hierarchy rooted at NUSEmp, five H-trees are created, one for each class: $H_{Lecturer}$, $H_{Researcher}$, $H_{Academic}$, H_{Admin} and H_{NUSEmp}. With the assumption that an instance is stored in only one class, each instance is indexed only once in the index of the class it is instantiated. Following the class hierarchy, $H_{Lecturer}$ and $H_{Researcher}$ are nested in $H_{Academic}$ and $H_{Academic}$ and H_{Admin} are nested in H_{NUSEmp}.

For a search on a class hierarchy rooted at Academic class, the nesting should enable us to obtain the correct answer by just performing a full search on $H_{Academic}$ and a partial search on $H_{Lecturer}$ and $H_{Researcher}$. The savings can be significant if many internal node pages can be skipped.

3.1 The data structure

The H-tree is a dynamic multi-level index that is based on the $B^{+}-$tree. However, to facilitate index nesting and to support superclass-subclass relationships, both the internal and leaf nodes of an H-tree contain more information. Figure 4 illustrates the structure of an H-tree and describes the notations used.

In an H-tree leaf node, an entry is a pair (K, P), where K is the indexed value for a fixed-length indexed key and a pair (*length, value*) in the case of variable length index values (e.g., strings). P consists of a counter and a list of oids, (number_of_oids, oid, oid, oid, . . .) whose indexed attribute value is K.

In an internal node N, apart from the usual discriminating key values, K, and child node pointers, B, we need to store pointers pointing to subtrees of nested H-trees. We use $L(n)$ to denote the pointer pointing to a subclass H-tree's subtree rooted at node n and simply L when the nested subtree node is not important to the discussion. To reduce unnecessary traversing of the nested subtree, the minimum and maximum values of the nested subtree are maintained together with the nested subtree pointer. The range values of a subtree rooted at B_i can be derived from its parent's entries, since B_i is contained in $(K_i,\ K_{i+1}]$ of the parent node. Figure 5 shows an example of a subtree (rooted at n) of $H_{subclass}$ being nested in a node (N) of $H_{superclass}$. As shown in the figure, the values in the nested subtree originated at node n must be within the values of 26 and 100. For efficiency reasons, we do not allow L pointers in a leaf node unless it is also the root.

In general, all the B^{+}-tree rules apply to H-trees. Each internal node may have up to M_i discriminating values and $M_i + 1$ branches (B). In an internal node, the K values in the subtree referenced by B_j ($j = 1, \cdots, M$) must be greater than K_{j-1} and less than or equal to K_j. A node cannot be

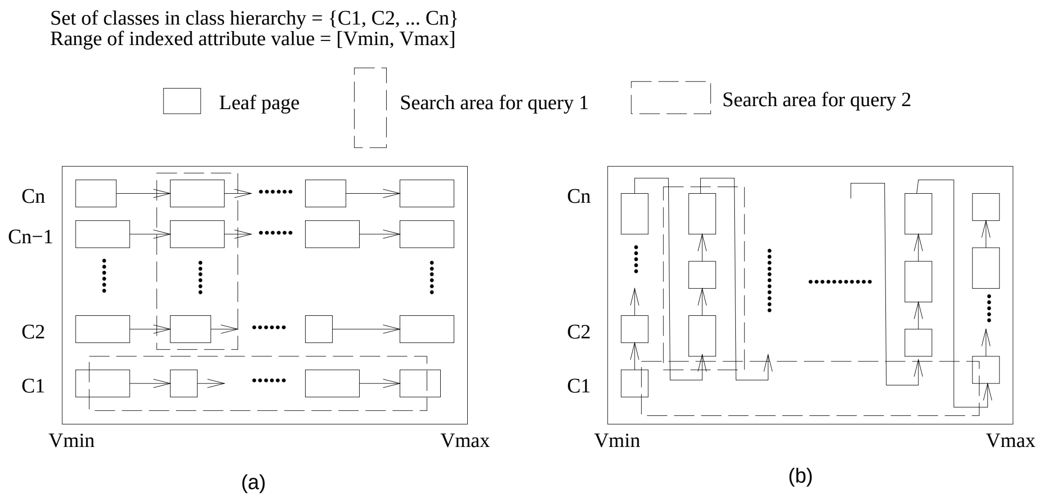
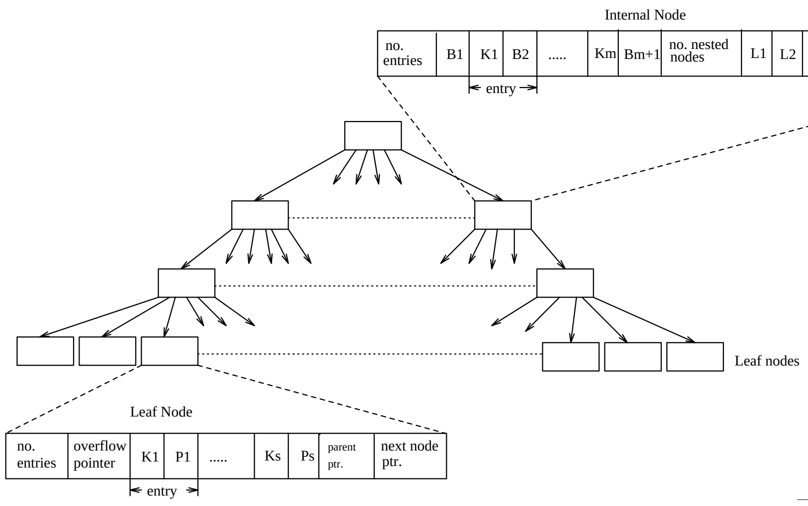

Set of classes in class hierarchy = {C1, C2, ... Cn}
Range of indexed attribute value = [Vmin, Vmax]
Leaf page
Search area for query 1
Search area for query 2
Cn
Cn−1
C2
C1
Vmin
Vmax
(a)
(b)

Fig. 3. Organization of data space in **a** an H-tree and **b** a CH-tree



Notations	
B	pointers pointing to subtrees of a node
H	index tree
K	indexed value (or branching value in internal nodes)
L	range values and pointer to node in nested tree
M_i	Maximum number of entries in an internal node
M_l	number of entries in a leaf node
P	pointers to objects or oids

Fig. 4. The H-tree structure

an empty node unless it is also the root node. Readers may refer to Comer (1979) for a complete B^+-tree description.

3.2 Nesting of indexes

An H-tree facilitates efficient retrieval of objects in a class. For a hierarchy of two classes, two indexes $H_{superclass}$ and $H_{subclass}$ are maintained. The superclass index $H_{superclass}$ is an outer index and the subclass index $H_{subclass}$ is an inner or nested index. Linkages are maintained between some nodes of these two indexes such that a search for values in the class and its subclass requires only a full search on the superclass index and a partial search on the subclass index. When index $H_{subclass}$ is nested in $H_{superclass}$, the L pointers of the internal nodes of $H_{superclass}$ will be set to point to the nodes in $H_{subclass}$. Referring to the example in Fig. 5, a subtree (rooted at n) of $H_{subclass}$ is being nested in a node (N) of $H_{superclass}$. Node N of $H_{superclass}$ has a pointer, $L1$, that links $H_{superclass}$ and $H_{subclass}$.

We define two rules for nesting a subclass index $H_{subclass}$ in a superclass index $H_{superclass}$. These rules ensure that the data can be retrieved correctly using the index. They are:

(C1) If node n is referenced by node N, the range values of the subtree rooted at n must be within the range values of node N, except when N is also the root node of $H_{superclass}$. The root node of $H_{superclass}$ is assumed to cover the range of $H_{subclass}$.

(C2) All the leaf nodes in $H_{subclass}$ must be covered by $H_{superclass}$. This means that all the leaf nodes in $H_{subclass}$ must be reachable from $H_{superclass}$.

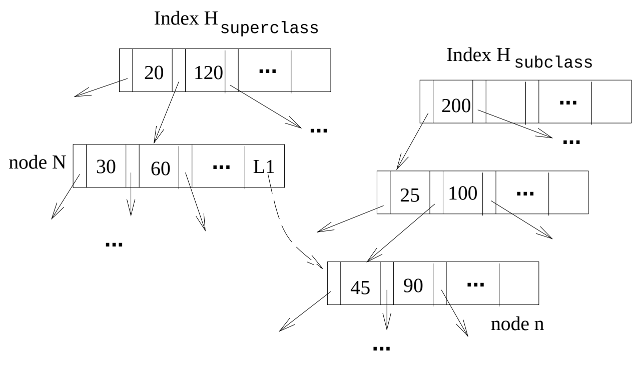


Fig. 5. Nesting part of an index

Rule C1 ensures that subtrees of subclass H-trees are defined where they are nested so that they can be reached via the correct path. Rule C2 ensures that all the nodes in $H_{subclass}$ can be queried through its superclass index $H_{superclass}$. An example of illegal nesting of indexes is shown in Fig. 6, where subtrees of $H_{subclass}$ reachable from $H_{superclass}$ are enclosed with dotted lines. In this example, n_5 is not reachable from $H_{superclass}$, and hence rule C2 is violated. Figure 7 shows a complete coverage of the leaf nodes in a nested index.

To increase the efficiency of the index, the following rules are introduced.

(E1) For each leaf node in $H_{subclass}$, there exists only one path to reach the node from $H_{superclass}$.

(E2) Suppose the immediate child nodes of n are $n_1, \cdots, n_j$. If nodes $n_1, \cdots, n_j$ are referenced from N, then node n should be referenced from N instead. This rule is to avoid unnecessary overflows.

(E3) The subtrees in $H_{subclass}$ referenced by $H_{superclass}$ should be as small as possible. For example, suppose the immediate child nodes of n are $n_1, \ldots, n_j$ and n is referenced from N. If there exists N_i, a child node of N, that can reference $n_i, \ldots, n_{i+k}$, then N_i should be set to reference $n_i, \ldots, n_{i+k}$, and N set to reference $n_1, \ldots, n_{i-1}, n_{i+k+1}, \ldots, n_j$.

Rule E1 is essential to ensure that a node in $H_{subclass}$ can only be referenced by a node in $H_{superclass}$ and hence no multiple search paths.

Rules E2 and E3 appear contradicting, but they are subtly different. Rule E3 aims to reduce the search in nested subtrees, while rule E2 reduces unnecessary overflow of L pointers in the $H_{superclass}$. A special case of rule E3 is when a node N and its child N_i can both cover the range values of node n. This is when $n_1, \ldots, n_{i-1}, n_{i+k+1}, \ldots, n_j$ is empty. In an earlier publication (Low et al. 1992), this special case of rule E3 was named rule E4.

4 Operations in H-trees

In this section, we present the outline of the algorithms for accessing (search operation) and updating (insert and delete operations) H-trees.

4.1 Searching

An H-tree can be searched under three situations:

1. Starting from the root node, the tree is searched as an index for instances of the indexed class.
2. Starting from the root node, the tree is searched as the root class of a class hierarchy and the links to (some or all of) its subclass H-trees are followed to search for instances in its subclasses.
3. Starting from an internal node via the link maintained in the superclass H-trees, the tree is searched.

The first case is a single-class search and the second and third cases are multiple-class searches. To search on a single class for instances which satisfy the search condition, the H-tree is searched like a B^+-tree by ignoring the nested tree pointers. Consider the example in Fig. 5, to search on class $H_{superclass}$ for its instances with indexed attribute value 40 we go down the subtree that is between 30 and 60, ignoring the $L1$ pointer. A multiple-class search begins the search on the H-tree of the root class and follows the L pointers to search the nested subtrees of classes of interest to the query. In this example, since the access scope includes $H_{subclass}$, index $H_{subclass}$ is searched starting at node n.

The search strategy is outlined below. We assume that the $search\_classes$ contains the subclasses whose indexes are to be searched, which is an empty set for a search on a single class. When searching a class, if none of the classes in $search\_classes$ is its subclass, its index search will be treated as a single-class search.

Algorithm Search

```
SEARCH ($cnode$, $v_1$, $v_2$)
Input:  $cnode$ – root node of tree/subtree to search.
        $v_1$ – lower bound of range search values.
        $v_2$ – upper bound of range search values.
        $v_2$ = $v_1$ for exact match search.

Output: list oids whose indexed attribute values fall within [$v_1$, $v_2$].

 1. Single-class search
    A. If $cnode$ is a leaf node, search the node and for all
       indexed values fall within [$v_1$, $v_2$], add the oids to the
       answer. Search the next leaf node and follow the chain
       till an indexed value greater than $v_2$ is encountered or
       till the last leaf node in the chain.
    B. If $cnode$ is an internal node, traverse down the first
       branch if $K_1 > v_1$, else traverse down the $i^{th}$ branch
       for the smallest $i$ where $K_{i-1} < v_1 \leq K_i$. If none of
       the discriminating values $K_i$ is greater than $v_1$, traverse
       down the right-most branch.
 2. Multiple-class search
    A. If $cnode$ is an internal node, search the subtree if its
       range intersects [$v_1$, $v_2$] and search the nested trees for
       all $L$s whose class is in $search\_class$ and range inter-
       sects [$v_1$, $v_2$], calling SEARCH ($L$, $v_1$, $v_2$). If $cnode$
       is reached via $L$ link of another class, check its node
       bitmap to see if any of its ancestor nodes contain $L$
       pointers to classes of interest. If the corresponding bit
       is set, traverse upwards and check the links.
    B. If $cnode$ is a leaf node, retrieve all data items that fall
       within [$v_1$, $v_2$].
```

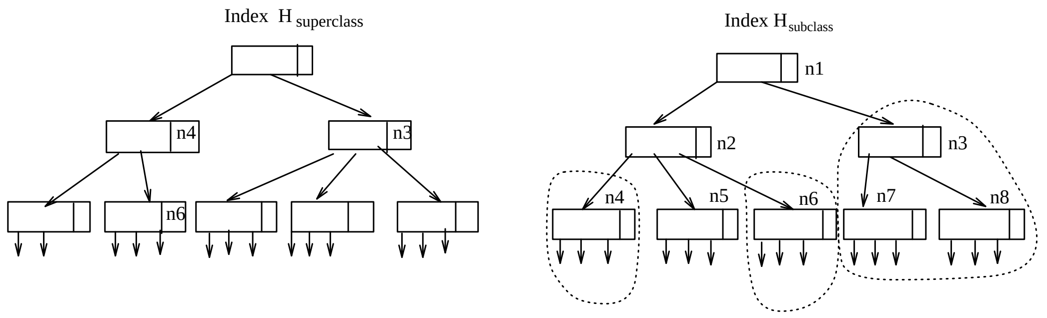


Fig. 6. Incomplete nesting of index $H_{subclass}$ in $H_{superclass}$

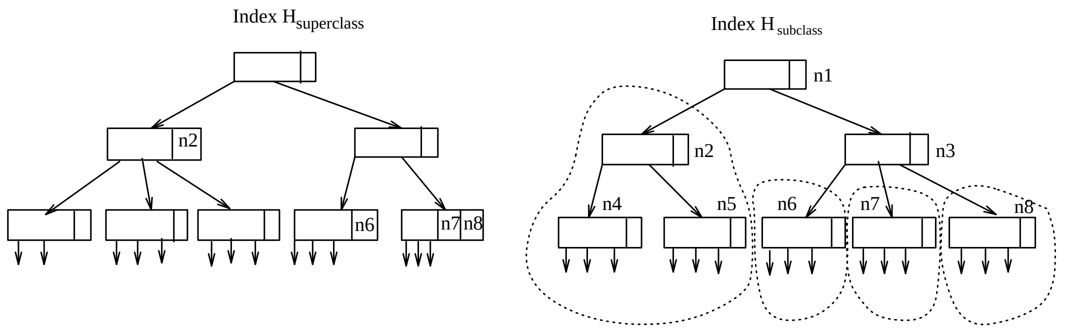
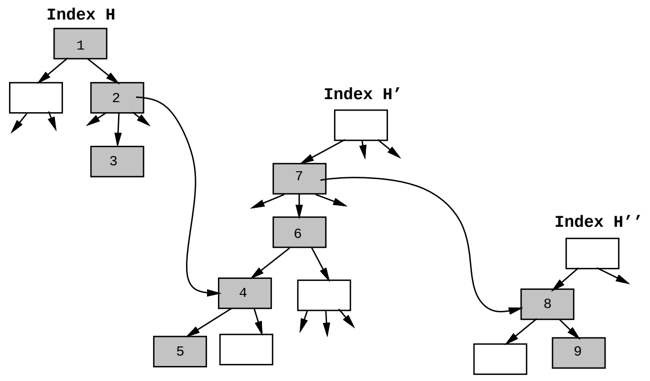


Fig. 7. Complete nesting of index $H_{subclass}$ in $H_{superclass}$



Index H'' is nested in H' and index H' is nested in H.
The numbers in the nodes are the search sequence of the nodes.

Fig. 8. Example of a search path, starting from the root of index H

In the above algorithm, the search of an H-tree starts at its root and traverses its subtrees before the nested subtrees. When searching a nested tree, not the whole tree is searched, only the subtrees nested in the nodes of the search path have to be searched. Subtrees not nested in the nodes of the search path are not searched because the search values are not within the search range.

Searching for values from a single class is similar to that in B^+-trees. To search for instances whose indexed key values fall within the search range [v_1, v_2], the algorithm searches for v_1 on the H-tree of the queried class. Once a leaf node is reached, the leaf nodes are scanned sequentially until a key value larger than the maximum search value (v_2) is encountered. For multiple class search, the L pointers of appropriate subclass H-trees are followed. When a subtree of an H-tree is searched via the L pointer in the superclass H-tree, we have to check whether there are any ancestor nodes that contain L pointers to its subclass H-trees. To facilitate such upward traversal, a bitmap, a bit for each subclass, is used to indicate the existence of L links in the ancestor nodes. For a class with eight subclasses, a byte is allocated for such purposes.

Suppose we have a class hierarchy of three levels and we have to perform a range search on the root class and all its subclasses. To simplify the explanation, we shall only consider one class at a level, with H-trees H, H', H'' respectively at the root, second, and third level. A subtree, S, in H' is searched via the L link in H. When searching for v_1 in S terminated, and H'' is not reached at all, it does not mean that subtrees of H'' within the range [v_1, v_2] are not nested in H'. It could be possible that the nesting occurs at a level higher than S. One such example is illustrated in Fig. 8. Therefore, moving up the tree from the current class nested node is essential. The node's bitmap is used to avoid unnecessary checking. An alternative solution to moving up the tree is to modify the nesting rules to nest only subtrees whose parent nodes have no L pointers. This way, when searching the subtree of a nested index, searching for nested subclass nodes is avoided as there is no L pointers in the ancestor nodes of n_L. However, searching may not be efficient as additional page accesses are incurred because of less efficient nested tree pruning. We adopt the first approach in our implementation and strictly enforce rules E3 and E4 to reduce upward search.

4.2 H-tree construction

4.2.1 Index nesting

When indexing a common attribute, the indexes are created bottom-up from the most specialized classes to the most general class; indexes for the subclasses are created before the index for the superclass. To index the class hierarchy in Fig. 1 on attribute *salary*, we first built the indexes for classes Lecturer, Researcher and Admin. The index for class Academic is then created, nesting the indexes of its subclasses, Lecturer and Researcher. The index for class NUSEmp is created last, nesting the indexes for Academic and Admin.

The nesting algorithm outlined below ensures that the rules defined in Sect. 3.2 are satisfied. $RangeValues(N)$ returns the range values of a node N.

Algorithm Nest

```
Nest ($N$, $n$)
Input:   $N$ – node of the superclass H-tree.
         $n$ – node of the subclass H-tree to be nested.

    if $N$ is not the root
        Exit if $N$ is a leaf node;
        Exit if $RangeValues(n)$ is not contained
        in $RangeValues(N)$;
    if $N$ is a leaf and the root node /* link node $n$ to $N$ */
        let $N.L_k$ be the empty $L$ pointer;
        set $N.L_k(n)$;
    else if the immediate child nodes of $N$ are leaf nodes
        let $N.L_k$ be the next empty $L$ pointer;
        set $N.L_k(n)$;
    else
        let $N_1 \ldots N_t$ ($t \leq M + 1$) be
            the immediate child nodes of node $N$;
        if there exists $N_i$ such that
        $RangeValues(n) \subseteq RangeValues(N_i)$
            call Nest ($N_i$, $n$); /* nest $n$ in $N_i$,
            a child node of $N$ : enforce a special case of E3*/
        else /* $n$ cannot be nested in node below $N$,
            try to nest the child nodes of $n$ instead. */
            let $n_1 \ldots n_s$ ($s \leq M + 1$) be
                the immediate child nodes of node $n$;
            for each $n_j$ ($j = 1 \ldots s$)
                call Nest ($N$, $n_j$); /* enforce E3 */
            if none of $n_j$ is nested in any of $N$'s child nodes,
            link node $n$ to $N$
                let $N.L_k$ be the next empty $L$ pointer;
                set $N.L_k(n)$; /* E2 */
            else /* some of $n$'s child nodes are nested
            in $N$'s child node,
                so nest the remaining unnested $n_i$
                in the current node */ for each unnested $n_i$
                    let $N.L_k$ be the next empty $L$ pointer;
                    set $N.L_k(n_i)$;
end Nest
```

To nest a subclass' H-tree in its superclass' H-tree, we traverse both trees simultaneously and try to push the L links down both trees as deeply as possible. For example, to nest an H-tree rooted at node n in its superclass' H-tree rooted at node N, we first attempt to nest the child nodes of n in the child nodes of N. If this is not possible, then only do we nest n in N.

Creating an H-tree index for a class without subclasses is similar to creating a B^+-tree. For a superclass without any instances, an empty root node is created to nest the indexes of its subclasses. In a way, range values of a superclass assume the range values of its subclasses, and the initial nested structure would be a linked list of root nodes.

4.2.2 Insertions

Inserting a new entry into an H-tree index is similar to that of the B^+-tree; as a new entry is added to a leaf node, an overflowed leaf node is split, and the split may propagate up the tree. Let oid_{new} be the object to be inserted into H, and its indexed attribute value be v_{new}. The insertion algorithm is outlined below.

Algorithm Insert

INSERT (v_{new}, oid_{new}, H)
Input: v_{new} – value of new object to insert.
oid_{new} – oid of new object to insert.
H – index to insert the new object.

1. Traverse index H to the leaf node that may contain v_{new}.
2. For secondary key indexing, the leaf node may already have v_{new}, in which case oid_{new} is added to the oid list of the indexed v_{new}. To insert the new entry into the leaf node, get the location of the new entry and shuffle the existing entries to make room for the new entry. If the leaf node does not have enough room for the new entry, call SPLIT_NODE (leaf node to be split, v_{new}, oid_{new}).

A node is split if overflow occurs. Like B^+-trees, a split may propagate upwards. In the splitting of a node n, the ranges of two resultant nodes, n_1 and n_2, are likely to be smaller than that of n. If there is $L(n)$ in the superclass H-tree, it has to be substituted with $L(n_1)$ and $L(n_2)$. The new Ls, for their smaller ranges, may be pushed further down the superclass index because of rule E4. However, violation of this rule does not affect the correctness of the H-tree operations. If there are L links maintained in node n, during the split, those that cannot be covered by n_1 and n_2 are promoted to their parent.

The node splitting algorithm is outlined below, which is designed to ensure that the H-trees obey rules defined in Sect. 3.2.

Algorithm Split Node

SPLIT_NODE ($lnode$, v_{new}, oid_{new})
Input: $lnode$ – node to split.
v_{new} – value of new object to insert.
oid_{new} – oid of new object to insert.

1. If $lnode$ is a leaf node, then use $lnode$ as the left node and create a new leaf node $lnode_{new}$ as the right node. Distribute the entries in $lnode$ plus the new entry among the left and right nodes. If there exists $L(lnode)$ in N of $H_{superclass}$, add $L(lnode_{new})$ to N. Update the parent node of $lnode$ to include the branch to the new node using the largest indexed value of the left node as the new branching value.
2. If the parent node n overflows, split the parent node.
   A. If n is the root node, create a new internal node and make it the parent node of n.
   B. Create a new internal node n_{new}. Let the middle branching value be K_{middle}. Move all the entries on the right of K_{middle} to the right node n_{new}.

C. Distribute existing L entries in n among n and n_{new} based upon their minimum and maximum values; the minimum and maximum indexed values of the subtree pointed by L must be enclosed by the minimum and maximum indexed values of the nesting node. Move the L entries which do not fit in neither n nor n_{new} to their parent node.

D. If there exists $L(n)$ in N of $H_{superclass}$, add $L(n_{new})$ to N and readjust.

E. Insert a new entry with K_{middle} as the branching value to the parent node of n. n_{new} becomes the right child of K_{middle}. Repeat step 2 if overflow occurs. This process may recur till the root node.

4.3 Deletion

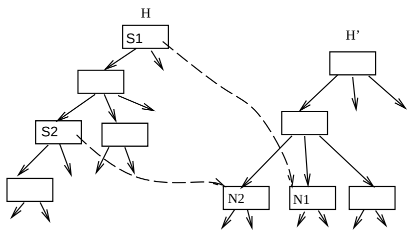
A deletion of an entry may cause a leaf node to underflow. In other words, the space utilization is less than the threshold value. The threshold value is typically half of the page capacity, which can be, however, tuned for performance purposes. When an internal node is underflowed, it is merged with either its left or right sibling. The merging requires readjustment of the links, which sometimes may result in pushing up the links in the parent index. The outline of the deletion algorithm is given below.

Algorithm Delete

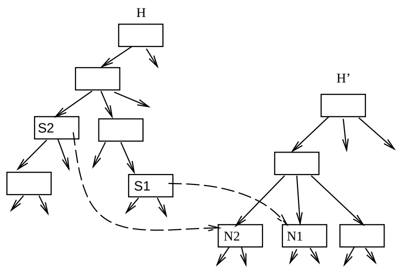
DELETE (v_{delete}, H, $oids$)

Input: v_{delete} – indexed value to delete.
$oids$ – list of objects to be deleted.
H – index to delete from.

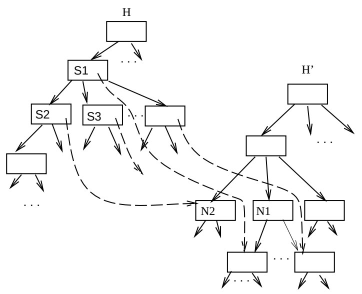
1. Traverse index H to the leaf node that contains the value v_{delete}. Let the leaf node be $cnode$. Delete the $oids$ and remove the indexed entry with $K = v_{delete}$ if its P is empty.
2. If the deletion is to remove all the indexed value v_{delete} in H and its nested indexes, search through the nested entries as in the algorithm SEARCH to delete all the indexed values with $K = v_{delete}$.
3. If $cnode$ underflows after removing the entry: Merge it with its sibling node $node_{sibling}$. Let the resultant node be $cnode$. Redistribute the entries among $cnode$ and $node_{sibling}$ if overflow occurs.
4. A. If resplit occurs, if there are $L(node_{sibling})$ and $L(cnode)$ in its superclass' H-tree, a simple readjustment is enough; check if they are required to be moved up or down.
   If there exists only one link, say $L(cnode)$, in N of $H_{superclass}$, then use Nest to re-nest all child nodes of $cnode$ in N.
   Check also if the L links in both nodes need to be readjusted.
   B. Otherwise, if there are $L(node_{sibling})$ and $L(cnode)$ in its superclass' H-tree among these two nodes, put $L(cnode)$ in the node whose range provides better coverage of the range of $cnode$. Delete old $L(node_{sibling})$ and $L(cnode)$. Check if $cnode$ is covered properly; if not, move the $L(cnode)$ up.
   If there is only one link, say $L(cnode)$, use the Nest algorithm to renest all child nodes of $cnode$ to the node that contains $L(cnode)$.
   Check the parent node if there are any links to a subclass that could be pushed down to $cnode$.
5. If a node is deleted, the corresponding entry in the parent node must be deleted. If the parent node is the root node and has one entry after the deletion, make its child the new root. If it is not the root and the node underflows, repeat step 3 with parent node as $cnode$.



a Case 1



b Case 2



c Case 3

Fig. 9a–c. Dangling nodes in deletions

Like the B^+-trees, merging of leaf nodes may propagate upward to the root node. If a node underflows, the node will be merged with its sibling (either left or right) node. A resplit is necessary if the resultant node is overflowed. This is always the case if the threshold value is more than half of the page capacity. When two nodes are merged and the resultant node is resplit, L links pointing to adjusted nodes must be readjusted. Due to their smaller ranges, the corresponding links in the superclass index pointing to two resplit nodes are more likely to be pushed down rather than being pushed up. For the same reason, the L links pointing to subclass H-trees may be pushed up to the parent node. When two nodes are merged, the resultant node has a bigger range and the L links in its parent node may be moved down.

In the Delete algorithm, the checking and relocation of L pointers can be achieved using the Nest algorithm. However, this is not efficient. In what follows, we analyze various cases where links must be rearranged. Figure 9 shows three possible cases in merging nodes N_1 and N_2 of H'. We first consider the case where resplit is not necessary and we use N_r to denote the merged node. In the first case, the link on the higher superclass node is used to link the merged node, and pushing down of the link is performed if necessary. Since the link to N_2 is in a subtree of the node S_1, the range of N_2 must also be covered by the range of S_1. In the second case, merging N_1 and N_2 results in a node with a larger range, which cannot be covered by either RangeValues(S_1) or RangeValues(S_2). The most immediate common parent

Table 1. Parameters used in analysis

Control parameters	
Term	Description
N	Total number of objects in the class hierarchy
L_{max}	Number of levels in the class hierarchy
$\mathscr{N}\mathscr{V}$	$\frac{Number\ of\ contiguous\ values\ in\ a\ range\ query}{Number\ of\ values\ in\ the\ domain} \times 100\%$
f	Fanout at each level of the class hierarchy
p	Probability of an object belonging to a level in the class hierarchy
n_e	Number of entries in a node
r	Occupancy rate

of S_1 and S_2 must be found and the link to N_r is inserted. The search for the parent is optimized using the information in the bitmap of a node; that is, a search for the immediate common parent using a node higher up in the tree. Appropriate pushing down is performed to tightly contain the range of N_r. In the third case, one of the merging nodes (N_1 in our example) has links from the superclass to its child node. The link from S_2 to N_2 is deleted, and the Nest algorithm is used to nest the children N_2 in H starting from S_2. This is sufficient since by definition, all leaf nodes of H' must be fully covered by H, implying that all child nodes of N_1 are covered by H.

In the case where resplit is necessary, the links from H to N_1 and N_2 are merely readjusted in cases 1 and 2. In the third case, the process is similar to that without split.

Both insertion and deletion algorithms are designed to ensure the effectiveness of inter H-tree linkage. In an H-tree, the links to subclass H-trees are optimized such that they are near the leaves. Furthermore, records have been partitioned into subclasses, reducing the number of records in each H-tree. With a reasonable page size (typically 2–4 KB), the realistic page height is not large.

5 Performance analysis

In the work of Low et al. (1992), we presented the best case analysis of the H-tree in terms of the storage space requirement and page accesses of range queries. The performance of the H-tree was compared with that of the CH-tree. In this section, we shall present the average case analysis of the H-tree based on an object distribution model and some experimental results. Table 1 defines the parameters we will use in our discussion and analysis.

5.1 The H-tree access complexity

The H-trees for a class hierarchy are a complex structure involving inter-relations among a hierarchy of B^+-trees. The H-trees could be viewed as a non-height balanced B^+-tree, by treating the H-tree for the root class as the main B^+-tree and the others as subtrees. The nesting level of the subclasses depends very much on the distribution of the key attribute value and the distribution of objects in the various levels in the class hierarchy.

To perform a search for data distributed across multiple classes, we have to traverse an H-tree from the root and

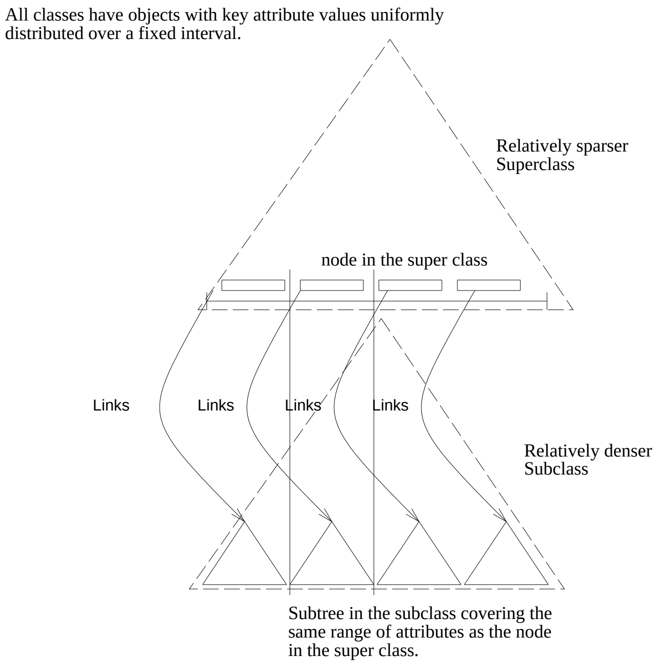

All classes have objects with key attribute values uniformly distributed over a fixed interval.
Relatively sparser Superclass
node in the super class
Links
Links
Links
Links
Relatively denser Subclass
Subtree in the subclass covering the same range of attributes as the node in the super class.

Fig. 10. Best case ‘nesting’ of a subclass into its superclass

the others through the links via the first H-tree. In order to calculate the mean access time, we express it in terms of the number of link and node traversals. To make the average case analysis possible, we make the following assumptions on the distribution of data.

1. The distribution of the key attribute values in all the classes are uniformly distributed over a fixed interval.
2. The distribution of the key attribute values is independent of the designation of objects into the classes.
3. The number of objects at each level of the class hierarchy follows a suitable geometric progression.

The first two assumptions are important as they form the basis to relate the height of a nested subtree to the relative sizes of the classes, without which analysis would be almost impossible. By these two assumptions, we observe that during a ‘nesting’ of a subclass into its superclass, each last level internal node in the superclass covers an equal portion of the subclass. That is, there is a link from each last level internal node in the superclass to the root of a subtree in the subclass (Fig. 10). Given a large enough number of objects in a class, the height of these subtrees do not differ significantly. By assuming a reasonable distribution of objects with adjustable parameters, we allow the analytical result to be adjusted according to specific applications.

On the aspect of the data structure, the size of each node in each H-tree is assumed to be the same, and an average occupancy rate is assumed for internal nodes of H-trees. To simplify the analysis, we also assume that the fanout of a class, the number of subclasses, is the same for all classes except those at the leaf level which have zero fanout.

In OODB, objects with the same properties and structure are grouped into one class. These objects may further be classified or grouped according to some more specified properties. Objects that cannot find a suitable subclass stay with the current class where they are structurally best described. In practice, subclasses are defined to better define the roles of objects and, as such, classes with subclasses have less objects than those classes at the leaf level of the class hierarchy. Therefore, when a class is specialized to a number of subclasses, instances migrate from the existing class to its subclasses which better describe their characteristics. Hence, more instances are distributed at the lower level. To better reflect such distribution, we can define the probability of instances being distributed at different levels, with the lowest level having the highest probability. We fix the probability of an instance being distributed at the lowest level to be p and the probability of an instance getting stored at level i to be:

$$P(i) = \begin{cases} (1-p)^{L_{max}-i}, & \text{if } i = 1 \\ p(1-p)^{L_{max}-i}, & \text{otherwise} \end{cases}$$

Here, the level of the root is taken as 1. With the above probability, we now define the distribution of objects into the ith level of the class hierarchy to be:

$$NP(i)(= Np^k(1-p)^{L_{max}-i})$$

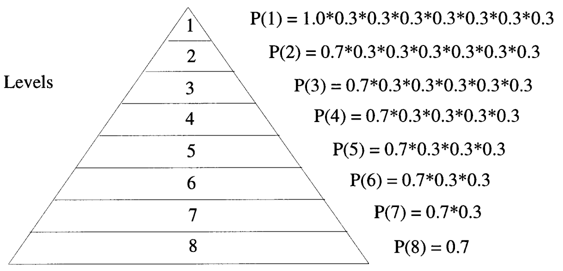
where k is 0 for $i = 1$, and 1 otherwise.

Intuitively, this method of distribution allows us to vary the way objects are distributed into the different levels of the class hierarchy in a simple way with just one parameter (p). To illustrate this, Fig. 11 shows the probabilities of objects belonging to the various levels of a hierarchy with $p = 0.7$. In this example, classes in level 8 will have 70% of the objects and the next higher level (level 7) will have 21% of the objects. This is logical since as a class is specialized into multiple subclasses, the objects are distributed to these subclasses. Only objects that do not fit the description of subclasses are stored in the original class. Depending on the class-hierarchy fanout and distribution probability p, a class which is nearer to the root may have fewer number of instances. Consider the example in Fig. 1, once Lecturer and Researcher have been defined, it is most likely that classes that better describe academic positions will be defined. If this is such a case, most of the objects in Academic class will be distributed to its subclasses. For such a data distribution with high p, the H-trees at the lower levels of the class hierarchy are bigger than the H-trees at the higher level. With the assumption that the distribution of attribute values within the classes is uniform and over the same fixed interval, the height of the subtree to be nested is logarithmically in proportion to the size of the subtree. The height of the subtree in a subclass nested in its superclass is therefore inversely related to the ratio of the size of the superclass to the subclass.

Let the number of objects in the superclass and subclass be N_u and N_l respectively. Then the number of leaf nodes in the superclass

$$= \frac{N_u}{n_e \times r} .$$

The H-tree stores the links (L pointers) only at the internal nodes, and therefore the nodes at the level just before the leaf level provides the most efficient subclass coverage. The number of nodes at the level before the leaf level



Levels
1
2
3
4
5
6
7
8
P(1) = 1.0*0.3*0.3*0.3*0.3*0.3*0.3*0.3
P(2) = 0.7*0.3*0.3*0.3*0.3*0.3*0.3
P(3) = 0.7*0.3*0.3*0.3*0.3*0.3
P(4) = 0.7*0.3*0.3*0.3*0.3
P(5) = 0.7*0.3*0.3*0.3
P(6) = 0.7*0.3*0.3
P(7) = 0.7*0.3
P(8) = 0.7

Fig. 11. Probabilities of objects belonging to different levels of the class hierarchy with $p = 0.7$

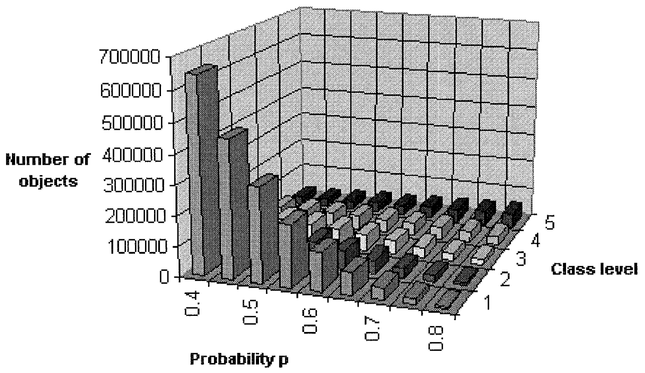


Fig. 12. A 3D histogram of the distribution of objects into each of the classes at different levels of the hierarchy with different values of p fixing $L_{max} = 5$

$$= \frac{N_u}{(n_e \times r)^2} ,$$

and the number of objects in the subclass within the interval covered by each of these nodes

$$= \frac{N_l \times (n_e \times r)^2}{N_u} .$$

The number of nodes at the base of the subtree that contains $\frac{N_l \times (n_e \times r)^2}{N_u}$ objects

$$= \frac{N_l \times n_e \times r}{N_u} .$$

The height of the subtree with $\frac{N_l \times n_e \times r}{N_u}$ leaf nodes

$$= \lfloor log_{n_e \times r} \frac{N_l \times n_e \times r}{N_u} \rfloor + 1 .$$

The cost of traversal from the root of the tree at the top of the class hierarchy to the ith level class, $Cost(i)$, is calculated as below.

$Cost(i) =$ height of first tree
+ height of subsequent subtrees in the lower levels of the class hierarchy.

$$= \lfloor log_{n_e \times r} \frac{N(1-p)^{L_{max}-1}}{n_e \times r} \rfloor + 1$$
$$+ \Sigma_{j=2}^{i} (\lfloor log_{n_e \times r} \frac{N_j \times n_e \times r}{N_{j-1}} \rfloor + 1) .$$

In the above cost, N_j and N_{j-1} are respectively the number of objects in a subclass at level j and a superclass at level $j-1$ ($j > 1$). The expected cost over L_{max} levels is

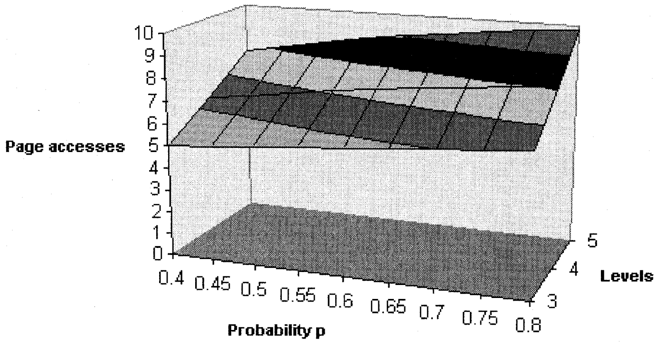


Fig. 13. 3D performance graph varying p and L_{max}

$= \Sigma_{i=1}^{L_{max}}(P(i)\times Cost(i))$.

We note that the H-tree provides a good mean access time if the distribution of the key attribute values are independent and uniform over all the classes, and that the size of the superclass is similar to that of the subclass. We compute the expected cost on parameters, N = 5 000 000, n_e = 150, r = 0.67 and f = 3. The distribution of objects into various classes at different levels is shown in Fig. 12.

As we can see from Fig. 13, the performance of the H-tree depends on the way the objects are distributed into the various classes in the different levels of the class hierarchy and the depth of the hierarchy. The three-dimensional performance surface graph indicates that a gradual increase in the number of objects in the classes at each level causes a gradual increase in the number of nodes being traversed. Databases with more objects in the lower levels of the class hierarchy tend to distribute their objects into smaller classes because of the fanout factor of the class hierarchy. With higher p, more objects are distributed to classes at the lower level of the class hierarchy. However, due to the fanout of the class hierarchy, the increase in height of the H-trees at these levels is not very great. In contrast, if the distribution is skewed heavily toward classes at the top of the hierarchy, the H-trees behave like a single class B^+-tree. When the size of a class and its subclass is comparable, the links from the class will link to smaller subtrees in the subclass. Thus, the cost saving comes from such a relationship. This is especially pronounced when this gradual increase in class sizes is initiated from a small root class.

In the original scheme, the superclass contains only objects which do not belong to any of the specialized subclasses. Although this idea is intuitive and natural, it does not maximize the advantage of nesting a subclass as most of the links will tend to be pointed to subtrees of a relatively large height. A good alternative is to invert the local structure by ‘promoting’ a subclass in the role of the parent class during storage. The choice of the subclass to be ‘promoted’ should be based on the relative size of the subclass to the rest of the peers. Alternatively, a relatively sparse superclass may be padded by dummy attribute values of a fraction of objects inserted in its immediate subclasses. Its efficiency is yet to be studied.

Table 2. Static parameters

Static parameters values			
Labels	Values used in exp.	Labels	Values used in exp.
Size(Page)	4096	Size(Node)	Size(Page)
Size(K)	4	Size(PageAddr)	12
Size(B)	Size(PageAddr)	Size(Oid)	12
Size(ClassId)	4	Size(Counter)	2
Size(Offset)	2		

5.2 Experiments

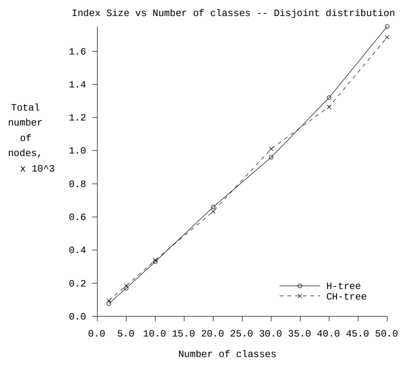
Both H-trees and CH-trees were implemented in C. For the H-tree, two-thirds of an internal node is allocated for the entries (K and B), and one-third is allocated for the nested tree pointers (L) and backward links to the nodes of superclasses. The space is fully utilized for entries in the CH-tree, and hence a CH-tree internal node contains about one-third more entries than an H-tree node. Table 2 describes the values of the parameters used.

The distribution of the key values across the classes of a class hierarchy has a significant impact on the performance of attribute-based indexes. For instance, if the key values of an indexed attribute are confined to instances of a single class, then an index like the H-tree which supports a single-class retrieval will perform better than an index like the CH-tree where instances of all classes in a class hierarchy are indexed in the same index. The distribution of indexed values can take one of the following forms:

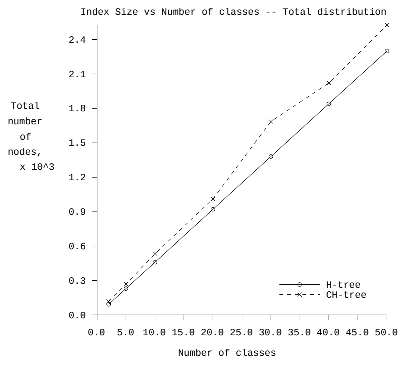
1. Disjoint – the domain of indexed values of each class in the class hierarchy is disjoint, i.e., an indexed value is contained in only one class.
2. Total inclusive – the domain of indexed values is the same for all classes in the class hierarchy. The probability of an indexed value appearing in any class is the same.
3. Partial inclusive – the domains of any two classes are partially overlapping, i.e., only some of the classes have objects with a particular indexed value.

The first two distributions are extreme cases and respectively represent the best and worst cases for an indexing technique.

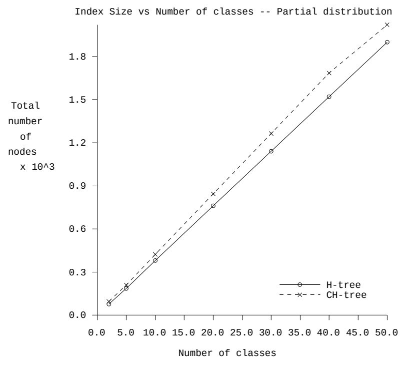
Like other databases, indexing of data objects within a class hierarchy is affected by the distribution of data. In OODB, objects are distributed into classes that best define their properties. When a new class is created, new objects are inserted into that class, or objects migrate from existing classes to it. In the first scenario, a new specialized class is created for new data that cannot fit the description of existing classes. Such a case is similar to inserting new records to the CH-tree and an H-tree. To see the effect of insertion of new objects into a new class, we fix the number of instances per class at 10 000 instances and we gradually vary the number of classes from 2 to 50, one at a time, and fix the domain size at 10 000 values. As a result, the total number of instances increases from 20 000 (2 classes) to 500 000 (50 classes). The number of levels of classes in the class hierarchy varies from 2 to 3. Figure 14 shows the storage requirements of both H-trees and CH-trees. For all three distributions, both indexes are fairly competitive in storage requirements. The number of classes with objects containing certain attribute



a



b



c

Fig. 14a–c. Storage requirement with addition of classes and objects. **a** Disjoint distribution. **b** Total inclusion distribution. **c** Partial inclusion distribution

values is the smallest for disjoint distribution, and the highest for total inclusive distribution. The CH-tree requires more space as the number of classes per attribute value increases. This is due to the increase in the directory size in its leaf nodes. While the CH-tree storage requirement is affected by the directory size kept in the leaf nodes, the H-tree storage requirement is affected by the number of classes involved and the height of each index. The storage requirement of H-trees is not highly affected by distribution of indexed values.

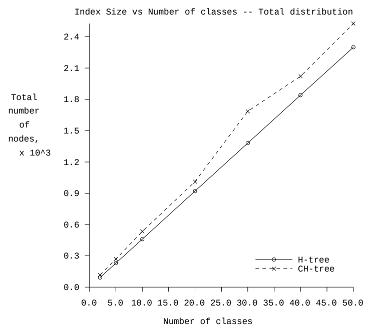
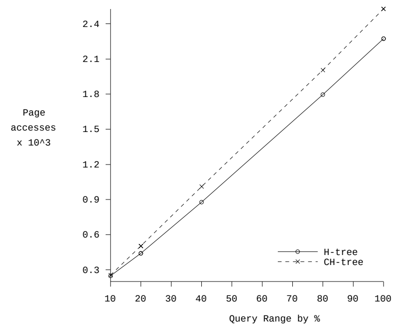


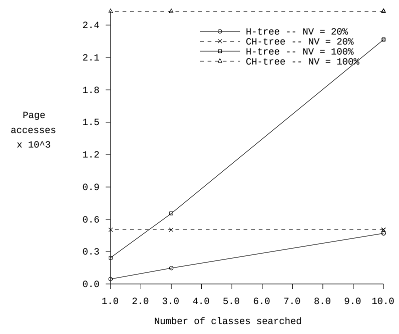
Fig. 15. The effect of object migration on storage requirement

In the second scenario, a more specialized subclass is created to better reflect the general characteristics of a subgroup of existing data. The support of such kind of incremental modeling requires an index to be able to adapt dynamically to migration of data. When a new subclass is created, data from existing classes within the same class hierarchy may migrate to it. Addition of a new class will cause the increase of the number of leaf nodes of the CH-tree since more class identities have to be stored. For the H-tree indexing scheme, a new H-tree must be created and more internal nodes will be used for indexing. In Fig. 15 we present the storage requirement for the total inclusive distribution with object migration. For the CH-tree, as the number of classes increases, the directory size in the leaf nodes also increases. The fact that an increase in the number of classes decreases the number of oids per class only causes a slight increase in storage space requirement. For the H-tree, when the number of classes increases, the storage space required increases more sharply due to an increase in the number of root nodes and smaller H-trees.

The distribution of data over classes in the class hierarchy affects the size and height of indexes and hence their performance. For the disjoint distribution, a class can be identified for a given range and therefore a small portion of the database needs to be searched. Such a query is biased towards indexes that support an index for each individual class, as the number of respective indexes need to search can be rather small. Searching one H-tree is much more efficient than searching the CH-tree, since an H-tree is comparatively much smaller than the CH-tree. Unless the data is very skewed to a particular class or the query range is very large, the H-trees are expected to perform better than the CH-trees. For the total inclusive distribition, both H-trees and the CH-tree require the most amount of storage space among the three distributions. To study the performance of both indexes, a database of 500 000 ($\mathcal{N}$) instances following the total inclusive distribution is created. The instances are randomly generated over the domain of [1, 10 000] and distributed to 10 classes of a class hierarchy that is 3 levels high. The H-tree indexes use 2842 pages, whereas the CH-tree requires 2550 pages. We consider different query ranges and different number of classes being queried. While only one retrieval is performed for a full search range ($\mathcal{NV}$=100%), up to 1000 retrievals are performed for the other search ranges ($\mathcal{NV} < 100\%$). For ranges with more than one retrieval, an



a



b

Fig. 16a,b. Query efficiency. **a** The effect of query ranges. **b** The effect of the number of queried classes

average is taken. For the 100% search range ([1, 10 000]), although a direct sequential scan of leaf nodes is possible, we search the indexes using the searching routine. The results are summarized in Fig. 16.

In the first experiment (Fig. 16a), the percentage of queried values $[v_1, v_2]$ over the domain ranges varies from 10% (e.g., [1, 1000], [1001, 2000], etc.) to 100% ([1, 10 000]) and the queries are for all 10 classes. The results show that the CH-tree performance degrades faster than that of the H-tree as the queried range increases. More importantly, the results show that the 10 H-trees give an effect of a single index without storing the class identities in their leaf nodes.

In the second experiment (Fig. 16b), the number of classes involved in the range queries varies from 1 to 10, with $\mathscr{NV}$ fixed at 20% and 100%. As expected, the H-tree is a much more efficient indexing structure when the number of classes involved is small. With the increase in the number of classes, more links have to be followed and more internal nodes have to be searched. The performance of the CH-tree is independent of the number of classes being queried, since for a given range, all data, regardless of their classes falling within the range, must be retrieved.

Both empirical results exhibit the same behavior to that obtained using the best case cost models reported by Low et al. (1992). In these experiments, no overflowed internal nodes of H-trees have been recorded. This is largely due to the large amount of node space we reserved for the L pointers, which could be tuned to further improve the efficiency of H-trees. The results indicate that the H-tree is an efficient indexing structure for associative search in OODB. The H-tree can be used in conjunction with other indexes to provide support on path indexing.

Sreenath and Seshadri (1994) studied the performance of the H-tree and the CH-tree. In this study, one of the experiments varies the number of classes being queried from 1 to 10, and the range of range queries from 1% to 40%. For complete overlap distribution of data (cf. Table 1 of Sreenath and Seshadri (1994)), the H-tree was shown to be more efficient than the CH-tree for queries referencing up to 6 classes and on all ranges. As the number of classes gets larger, the performance of the two indexes becomes comparable. For point queries over all classes, the H-tree did not perform well in comparison to the CH-tree. This is expected since the CH-tree traverses a single path from the root to a leaf node and accesses a small number of leaf pages, whereas for the H-tree we need to follow all links to individual H-trees. The saving from reading additional leaf pages, as in the case of range queries, is now offset by the number of links the H-tree has to follow. For point queries, a hash table would be the most efficient indexing structure. A performance study on the CH-tree and H-tree was also conducted by Niu et al. (1994), and the results exhibit fairly similar behavior to that reported by Sreenath and Seshadri (1994).

6 Conclusions

In an OODB, retrieval of objects based on their key values and on their classes is very common. Such search can also be used to facilitate retrieval of component (or aggregate) objects based on certain key values of the component (or aggregate) objects. Support for efficient associative search is important. The H-tree provides a natural support for OODB where objects are stored in their proper classes.

In this paper, we have presented the structure of the H-tree and a case study of its deletion algorithm. The H-tree supports efficient associative search on instances of a single class, and instances of a class and some or all of its subclasses. Two access methods that can be efficiently implemented using H-trees are: (1) scanning for all instances of a class and its subclasses and (2) scanning for instances of a class and some select subclasses. By not following the path to the nested index of irrelevant classes, the condition where the instances are not members of those classes is naturally satisfied.

Experiments on the H-tree and CH-tree were conducted to evaluate the performance of the H-tree and CH-tree. The results show that the H-tree is an efficient structure both in terms of storage requirements and query efficiency. To provide the average case analysis of the H-tree, we presented a model for data distribution within a class hierarchy. The average cost model shows that the structure of the H-trees is suitable for applications where more objects are distributed to classes at the leaf level of the class hierarchy. In general, the study so far indicates that index nesting is an efficient approach to indexing multiple sets of data of the same data type. The index nesting concept has been studied by Kilger and Moerkotte (1994) as an index for indexing set-tuples, and by Low et al. (1993) as an index for facilitating query processing in deductive databases. Like other indexes, further fine-tuning of the H-tree is required to overcome its weaknesses and to further exploit its strengths. With the use of multiple indexes, the degree of concurrency should improve and it is of interest to study such an improvement.

Acknowledgement. We would like to thank Chee Chin Low, Thong Wei Koh, and Chee Yong Chan for their contribution in implementation and analysis.

References

Bertino E, Kim W (1989) Indexing techniques for queries on nested objects. IEEE Trans Knowl Data Eng 1:196–214

Carey M, DeWitt D, Richardson J, Shekita E (1986) Object and file management in the EXODUS extensible database system. In: Proc Intl Conf on Very Large Data Bases, Kyoto, Japan, pp 91–100

Chan CY, Ooi BC (1995) Chi-tree: a new class hierarchy index for object-oriented databases. (Technical Report) National University of Singapore, Singapore

Comer D (1979) The ubiquitous B-tree. ACM Comput Surv 11:121–137

Kemper A, Moerkotte G (1990) Access support in object bases. In: Proc ACM Intl Conf on Management of Data, Atlantic City, NJ, pp 364–374

Kilger C, Moerkotte G (1994) Indexing multiple sets. In: Proc Intl Conf on Very Large Data Bases, Santiago, Chile, pp 180–191

Kim et al. 1989kkd:kim Kim W, Kim KC, Dale A (1989) Indexing techniques for object-oriented database. In: Kim W, Lochovsky FH (eds) Object-oriented concepts, databases, and applications. Addison-Wesley, Reading, Mass, pp 371–394

Knuth D (1973) The art of computer programming, vol 3. Sorting and searching. Addison-Wesley, Reading, Mass

Kriegel HP (1984) Performance comparison of index structures for multi-key retrieval. In: ACM Proc Intl Conf on Management of Data, Boston, Mass, pp 186–196

Low CC, Ooi BC, Lu H (1992) H-trees: a dynamic associative search index for OODB. In: ACM Proc Intl Conf on Management of Data, San Diego, Calif, pp 134–143

Low CC, Lu H, Ooi BC, Han J (1993) Efficient access methods in deductive and object-oriented databases. In: Proc Conf on Deductive and Object-Oriented Databases, Munich, pp 68–84

Maier D, Stein J (1986) Indexing in an object-oriented DBMS. In: IEEE Proc Intl Workshop on Object-Oriented Database Systems, Pacific Grove, Calif, pp 171–182

Niu Y, Ozsu MT, Szafron D (1994) An evaluation of indexing techniques for object base management systems. (Technical report) Department of Computer Science, University of Alberta, Alberta

Scheuermann P, Ouksel M (1982) Multidimensional B-trees for associative searching in database systems. Inf 7:123–137

Sreenath B, Seshadri S (1994) The hcC-tree: an efficient index structure for object oriented database. In: Proc Intl Conf on Very Large Data Bases, Santiago, Chile, pp 203–213

Valduriez P, Khoshafian S, Copeland G (1986) Implementation of techniques of complex objects. In: Proc Intl Conf on Very Large Data Bases, Kyoto, Japan, pp 101–110