H-trees: A Dynamic Associative Search Index for OODB[1]

Chee Chin Low, Beng Chin Ooi, Hongjun Lu

Department of Information Systems and Computer Science
National University of Singapore
Kent Ridge, Singapore 0511

Abstract

The support of the superclass-subclass concept in object-oriented databases (OODB) makes an instance of a subclass also an instance of its superclass. As a result, the access scope of a query against a class in general includes the access scope of all its subclasses, unless specified otherwise. 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. Second, it must also support efficient retrieval of instances from classes in a hierarchy of classes. In this paper, we propose a new index called the H-tree that supports efficient retrieval of instances of a single class as well as retrieval of instances of a class and its subclasses. The unique feature of H-trees is that they capture the superclass-subclass relationships. A performance analysis is conducted and both experimental and analytical results indicate that the H-tree is an efficient indexing structure for OODB.

Keywords: OODB, indexing structures, query retrieval.

1. Introduction

One major difference between conventional databases and OODB is that in an OODB, a class 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. Therefore, an index for OODB must support both retrievals (excluding and including subclasses) by value efficiently.

Indexing superclass-subclass data for efficient associative search on either a single class or a hierarchy of classes is not well addressed in the literature. Based on B^+-trees, Kim et al [KKD89] proposed a scheme called the class-hierarchy tree (the CH-tree). A CH-tree maintains only one index tree for all the classes of a class hierarchy. 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 the 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. Indexing structures proposed in [BeK89, KeM90, MeS86, VKC86] mainly deals with path indexing for nested objects in OODB.

In this paper, based on the B^+-tree, we propose a new index called the H-tree. 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. The nesting 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 sub-hierarchy of classes can start at any class so long as they are indexed on the same attribute. The nested organization provides a natural and efficient support for superclass-subclass relationship. A different kind of index nesting, the multi-dimensional B-tree (MDBT), was proposed in [SuO82]. In a MDBT [SuO82], 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. Our nesting is different in that all H-trees index the same attribute and it is based on the superclass-subclass relationship.

This paper is organized as follows. Section 2 de-

[1]The work was in part supported by NUS Research Grant RP910654

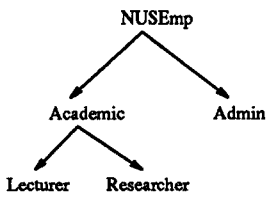
1992 ACM SIGMOD - 6/92/CA, USA

scribes the data structure and nesting organization of the H-tree indexes. The algorithms for searching, insertion and deletion are presented in Section 3. We implemented both H-trees and CH-trees [KKD89] and compared their performance. Both analytical and empirical results are presented in Section 4. Conclusions and future directions are presented in Section 5.

2. The H-Tree

2.1. Motivation

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 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 classes in a hierarchy of classes. Consider the class hierarchy in Figure 1. The NUSEmp is the root class 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, Academic and those of Academic, Researcher and Lecturer.




```
class NUSEmp
    {attributes : name, empno, salary };
class Admin subclass of NUSEmp;
class Academic subclass of NUSEmp
    { attribute : qualifications };
class Lecturer subclass of Academic
    { attribute : subject_teach };
class Researcher subclass of Academic
    { attribute : projects };
```

Figure 1: Class Hierarchy of NUS Employees

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

(1) instances of a particular class not including its subclasses.
For examples, (1) list all academic employees who are neither lecturers nor researchers and who earn more than \$50,000 and (2) list all lecturers who earn more than \$40,000.

(2) instances of a class and all its subclasses.
For example, list the employees (NUSEmp) who earn more than \$30,000.

The CH-tree proposed in [KKD89] uses the same searching strategy for both retrievals. The CH-tree indexes a hierarchy of classes on a common attribute, typically one of the superclass attributes, on a B^+-tree like structure. 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. In this paper, we propose an indexing mechanism that is designed to support retrievals of instances from a class or a hierarchy of classes. The proposed index achieves the speed of the one index one class scheme for single class retrievals and one index for all classes scheme (cf: CH-trees) for multiple classes retrievals.

2.2. The H-Tree Organization

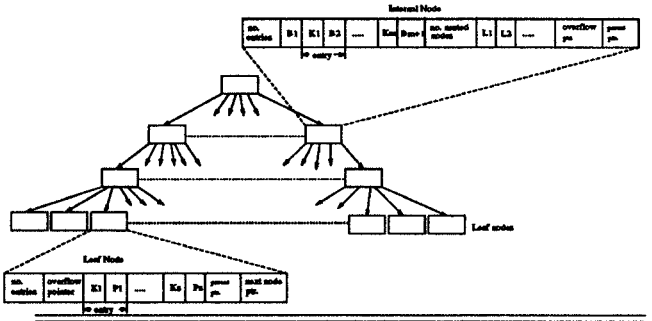
In this subsection, we describe the structure of the H-tree and the nesting of H-trees. Let H_c denote the H-tree of class c. To index the class hierarchy in Figure 1 on a common attribute, five H-trees are created, one for each class. Following the class hierarchy, $H_{Lecturer}$ and $H_{Researcher}$ 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 node pages can be skipped.

2.2.1. The Data Structure

In an H-tree leaf node, an entry is a pair (K, P), where K is the indexed value for fixed length indexed key and a pair $(length, value)$ in the case of variable length index values (eg. strings). P consists of a counter and a list of object identities (oids), (number_of_oids, oid, oid, oid, ...) whose indexed attribute value is K. Figure 2 illustrates an H-tree and the notations used.

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. For the example shown in Figure 3, the values in the nested subtree originated at node n must be within the

values of 31 and 100. For efficiency reasons, we do not allow L pointers in a leaf node unless it is also the root.



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

Figure 2: H-tree Structures

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 an empty node unless it is also the root node. Readers may refer to [Com79] for a complete B^+-tree description.

2.2.2. Nesting of Indexes

An H-tree is an indexing structure for a class where objects can be retrieved as in B^+-trees. 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 the nodes of these two indexes such that a search for values in the class and its subclass requires only to have 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 $H_{subclass}$. Figure 3 shows an example of a subtree (rooted at n) of $H_{subclass}$ being nested in a node (N) of $H_{superclass}$.

We define the following 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.

(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 $H_{superclass}$.

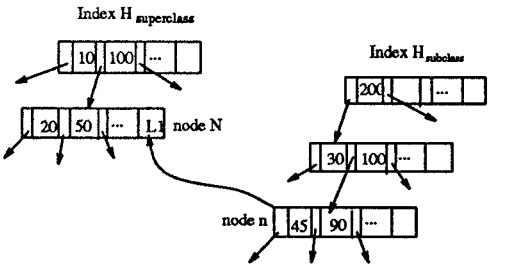


Figure 3: Nesting part of an index

Rule C1 ensures that subtrees of subclass H-trees are defined in 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 Figure 4, where subtrees of $H_{subclass}$ reachable from $H_{superclass}$ are enclosed with dotted lines. In this example, n_5 is not reachable from $H_{subclass}$, and hence rule C2 is violated. Figure 5 shows a complete coverage of the leaf nodes in a nested index.

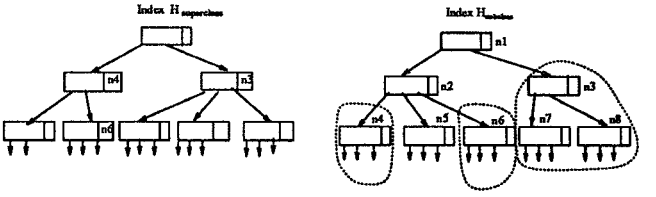
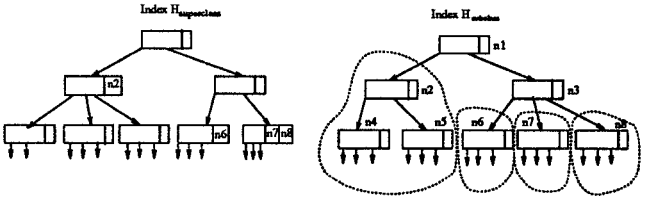


Figure 4: Incomplete nesting of index $H_{subclass}$ in $H_{superclass}$



The labels in the nodes of $H_{superclass}$ represent the the nodes in $H_{subclasses}$ the L pointers pointing to.

Figure 5: Complete nesting of index $H_{subclass}$ in $H_{superclass}$

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 $n_1..n_j$

and n is referenced from N. If there exists N_i, a child node of N, that can reference n_i, then N_i should be set to reference n_i and N set to reference $n_1, .., n_{i-1}, n_{i+1}, .., n_j$.

(E4) The enclosure of the range of a nested node n by that of its nesting node N should be as tight as possible. Suppose node N and its child N_i can both cover the range values of node n, then the child node N_i should be selected to reference n.

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, however, they are subtly different. E3 aims to reduce the search in nested subtrees, while E2 reduces unnecessary overflow of L pointers in the $H_{superclass}$. Rule E4 is to push the L links down the tree as deep as possible, reducing unnecessary checking of L links. Besides, during the nesting process, searching for where to store a link stops at the node where the range of the nested subtree cannot fit in the range of any two consecutive entries.

3. Searching and Updates

3.1. 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.

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. 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. Consider the example in Figure 4, to search on class $H_{superclass}$ for its instances with indexed attribute value 40, we go down the subtree that is in between 20 and 50, ignoring the $L1$ pointer. If the access scope includes $H_{subclass}$, then index $H_{subclass}$ is searched starting at node n.

The search strategy is outlined as follows. 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.

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) If *cnode* is a leaf node, search the node, for all indexed values fall within $[v_1, v_2]$, add the oids to answer. If the largest entry in the *cnode* is less than v_2, search the next leaf node and follow the chain till an indexed value greater v_2 is encountered or till the last leaf node in the chain.

(2) If *cnode* is an internal node,

(i) if this is reached via its parent node then Recursively traverse down the tree by calling SEARCH (B_i, v_1, v_2). 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. Search the nested trees for all Ls whose class is in *search_class* and range intersects $[v_1, v_2]$; call SEARCH (L, v_1, v_2).

(ii) if this is reached via L link of another class then Search the subtree S as in 2(i). Check all the entries of ancestor nodes in the bitmap to see if ancestor nodes contain links to subclass H-trees. If the corresponding bit is set, traverse upwards and check the links. For all L entries, if its class is in *search_class* and L range intersects $[v_1, v_2]$, call SEARCH (L, v_1, v_2).

(iii) For each nested H-tree, make use of the bitmap to find the first L pointer to search for instances of relevant classes. The path to that node is traversed if it is within the search range and the L pointer is followed.

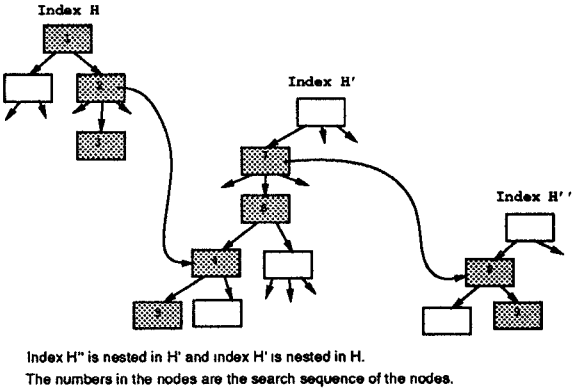


Figure 6: Example of a search path, starting from the root of index H.

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.

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 each nested H-tree, we must find the first L pointer to search for relevant instances. A bit map is used for this purpose. A bit is allocated for each node, and a full balanced tree is assumed so that the bit position of a corresponding node can be calculated. A bit is set when its corresponding node contains pointers pointing to nested H-trees. The path to that node is traversed if it is within the search range and the L pointer is followed.

When a subtree of an H-tree is searched via the L pointer in the superclass H-tree, we have to check if there are any ancestor nodes that contain L pointers to its subclass H-trees. 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. H' subtree S 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 Figure 6. Therefore, moving up the tree from the current class nested node is essential. Again, the bitmap for the nested H-tree is used to avoid unnecessary checking. An alternate 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 due to less efficient nested tree pruning. We adopt the first approach in our implementation and strictly enforce rules $E3$ and $E4$ to reduce upward search.

3.2. H-tree Construction

3.2.1. 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 Figure 1 on attribute *salary*, we first build 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 Academia and Admin.

The nesting algorithm outlined below ensures that the rules defined in Section 2.2.2 are satisfied. The Range(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 Range(n) is not contained in Range(N);
  if N is a leaf and the root node
    let N.L_k be the empty L pointer;
    set N.L_k(n); /* link node n to N */
  else if N immediate child nodes are leaf nodes
    let N.L_k be the next empty L pointer;
    set N.L_k(n);
  else
    let N_1 ... N_t (t ≤ M + 1) be the immediate
     child nodes of node N;
    if ∃ N_i s.t. Range(n) ⊆ Range(N_i)
      call Nest (N_i, n); /* enforce E4*/
    else /* n cannot be nested in node below N,
      try to nest the child nodes of n instead. */
      let n_1 ... n_s (s ≤ M + 1) be the immediate
       child nodes of node n;
      for each n_j (j = 1 ... s)
        call Nest (N, n_j); /* enforce E3 */
      if none of n_j is nested in any of N's
       child nodes
        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's H-tree in its superclass's H-tree, we traverse both trees simultaneously, and try to push the L links down both trees as deep as possible. For example, to nest an H-tree rooted at node n in its superclass's 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 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.

3.2.2. 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}.

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 be to 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 Section 2.2.2.

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:

(i) If n is the root node, create a new internal node and make it the parent node of n.

(ii) 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}.

(iii) 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.

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

(v) 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.

3.3. Deletion

A deletion of an entry may cause a leaf node to underflow; 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) (i) If resplit occurs,
if there are $L(node_{sibling})$ and $L(cnode)$ in its superclass's 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 $cnode$ in N.
Check also if the L links in both nodes need to be readjusted.

(ii) Otherwise,
if there are $L(node_{sibling})$ and $L(cnode)$ in its superclass's 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 re-nest 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 $cnode$.

Similar to B^+-trees, merging of leaf nodes may propagate upward to the root node. If the node underflows, the node will be merged with its sibling (either left or right) node. The resultant node is resplit if overflow occurs. L links pointing to adjusted nodes must be readjusted. In a resplit, 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, the L links in its parent node may be moved down. In the above algorithm, the checking and relocation of L pointers can be achieved using the Nest algorithm. Although the L links sometimes need to be relocated, they are however usually moved a level up or down.

4. Performance Analysis

The CH-tree has been shown to be more efficient than the approach of supporting one index for one class which is the approach H-trees adopted. Therefore, it is appropriate that we compare the performance of the H-tree against that of the CH-tree.

4.1. Analytical Analysis

In this section, we derive the best case cost model to estimate the performance of the H-trees and CH-trees. Table 1 presents the parameters used in the cost analysis for the best case.

Control Parameters	
Labels	Descriptions
$\mathcal{D}$	no. of unique values in the domain of the indexed attribute.
$\mathcal{D}_i$	no. of unique values in the index of class i.
$\mathcal{N}$	total no. of objects the class hierarchy.
$\mathcal{N}_i$	no. of object in class i.
f	Average branches of an internal node.
$\mathcal{TC}$	Total no. of classes in the class hierarchy.
$\mathcal{NC}$	no. of classes that contain objects with an indexed value.
$\mathcal{NCS}$	no. of classes in the search space.
$\mathcal{NV}$	$\frac{\text{no. of contiguous values in a range query}}{\text{no. of values in the domain}} \times 100\%$
size(X)	Size of the variable X.

Derived Parameters	
Labels	Descriptions
$\mathcal{NE}$	no. of entries in a leaf node.
$\mathcal{NO}_{vc}$	no. of objects per indexed value per class
$\mathcal{NL}_i$	no. of leaf nodes in index i.
$\mathcal{NI}_i$	no. of internal nodes in index i.
$\mathcal{NN}_i$	no. of nodes (leaf and internal) in index i.

Table 1: Parameters and Notations

H-tree storage space requirement:

$$\text{size(LEntry)} = size(K) + size(Counter) + (\mathcal{NO}_{vc} \times size(Oid))$$

$$\mathcal{NE} = \left\lfloor \frac{size(Node) - size(Counter)}{size(LEntry)} \right\rfloor$$

$$\mathcal{NL}_i = \left\lceil \frac{\mathcal{D}_i}{\mathcal{NE}} \right\rceil$$

$$\mathcal{NI}_i = \left\lceil \frac{\mathcal{NL}_i}{f} \right\rceil + \left\lceil \frac{\lceil \frac{\mathcal{NL}_i}{f} \rceil}{f} \right\rceil + \ldots + 1$$

$$\mathcal{NN}_i = \mathcal{NL}_i + \mathcal{NI}_i$$

$\mathcal{NN}_i$ is the lower bound estimation of the number of nodes in the i^{th} H-tree. Therefore, when indexing on a common attribute of a class hierarchy, the total number of nodes ($\mathcal{NN}$) is the sum of the number of nodes of all the indexes:

$$\mathcal{NN} = \sum_{for\ all\ indexes} \mathcal{NN}_i$$

H-tree range query cost (page access):

$$= \begin{cases} height(H_i) + \lceil \frac{\mathcal{NV}}{\mathcal{NE}} \rceil & (1) \\ \sum_{i=1}^{\mathcal{NCS}} \left(height(H_i) + \left\lceil \frac{\frac{\mathcal{D}_i}{\mathcal{D}} \times \mathcal{NV}}{\mathcal{NE}} \right\rceil \right) & (2) \end{cases}$$

where $height(H_i)$ is the number of levels in the H-tree of class i; second equation includes the search of subclasses, whereas the first one does not.

Now, let us analyze the CH-tree proposed in [KKD89]. In a CH-tree, a leaf node contains key values and associated directories; for each key value, oids of objects which have the same indexed value are grouped in a directory based upon objects' classes. The cost model for the CH-tree is given below, which is quite similar to that given in [KKD89].

CH-tree storage space requirement:

$$\text{size(Dir)} = size(Counter) + (\mathcal{NC} \times (size(ClassId) + size(Offset)))$$

$$\text{size(LEntry)} = size(K) + size(Dir) + (\mathcal{NC} \times (size(Counter) + (\mathcal{NO}_{vc} \times size(Oid))))$$

$$\mathcal{NE} = \left\lfloor \frac{size(Node) - size(Counter)}{size(LEntry)} \right\rfloor$$

$$\mathcal{NL} = \left\lceil \frac{\mathcal{D}}{\mathcal{NE}} \right\rceil$$

$$\mathcal{NI} = \left\lceil \frac{\mathcal{NL}}{f} \right\rceil + \left\lceil \frac{\lceil \frac{\mathcal{NL}}{f} \rceil}{f} \right\rceil + \ldots + 1$$

$$\mathcal{NN} = \mathcal{NL} + \mathcal{NI}$$

where $size(Dir)$ is the size of the directory in a leaf node entry. Notice that no subscript is used because only one index is maintained for a hierarchy of classes.

CH-tree range query cost (page access):

$$= height(CH - tree) + \left\lceil \frac{\mathcal{NV}}{\mathcal{NE}} \right\rceil$$

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 may perform better than an index like the CH-tree where instances of all classes in a class hierarchy are indexed in the same index. As in [KKD89], we consider three indexed value distributions :

(1) disjoint – the domain of indexed values of each class in the class hierarchy is disjoint, $\mathcal{D} = \sum_{i=all\ the\ classes} \mathcal{D}_i$ and $\mathcal{NC} = 1$,

(2) total inclusive – the domain of indexed values is the same for all classes in the class hierarchy, $\mathcal{D} = \mathcal{D}_i$ and $\mathcal{NC} = \mathcal{TC}$, and

(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, $\mathcal{D}_i = \mathcal{D} \times \frac{\mathcal{NC}}{\mathcal{TC}}$, where $1 < \mathcal{NC} < \mathcal{TC}$. We only consider $\mathcal{NC} = \frac{\mathcal{TC}}{2}$ in our simulations and experiments.

The first two distributions are extreme cases and respectively represent the best and worst cases for the indexing technique. Here we present the analytical results based on the cost models derived above, using page size of 4K bytes, page identifier and Oid of 12 bytes, ClassId of 4 bytes, Counter and Offset of 2 bytes. In the simulations, two third of an internal node of an H-tree 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.

We study storage requirement under two situations:

(1) As a new specialized class is created, some instances of existing classes are migrated to this new class.

(2) As a new specialized class is created, new instances are created for it.

In the first storage requirement study, we vary the number of classes in the class hierarchy from 2 to 50, while fix the domain size and total number of instances respectively at 10,000 and 500,000. The number of levels of the class hierarchy varies from 2 to 3. Figure 7 shows the storage requirements of H-trees and CH-trees, illustrating the effect on the total index sizes as more specialized subclasses are created. For the H-tree, when the number of classes increases, the storage space required increases due to an increase in the number of root nodes and smaller H-trees. The CH-tree performs well in the cases where values of the indexed attribute of different classes are disjoint, in which case the class directories maintained in the leaf nodes are not affected by the number of classes. For the other two distributions, 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.

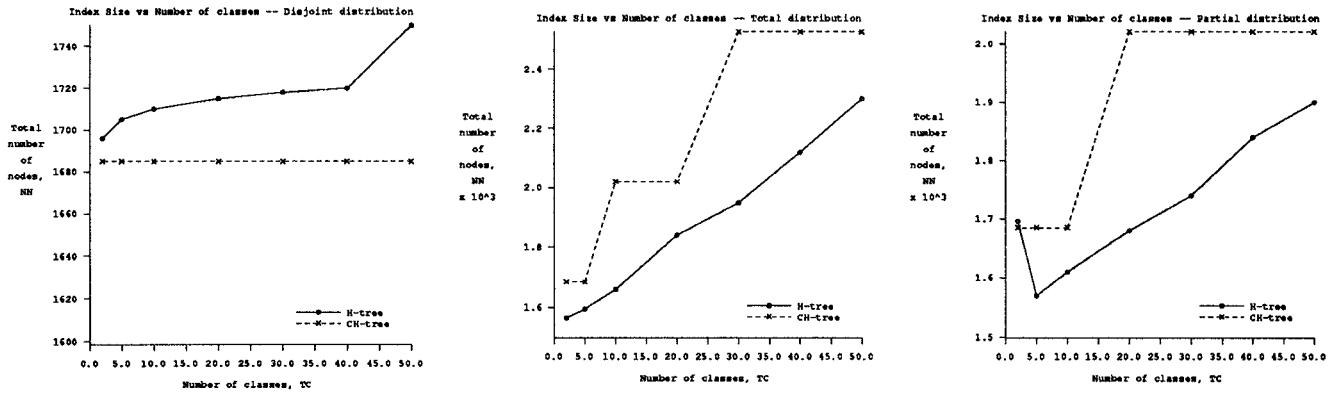
In the second simulation, we fix the number of instances per class at 10,000 instances, and we gradually vary the number of classes from 2 to 50, one at a time, and fix the domain size at 10,000 values. As a result, the total number of instances increases from 20,000 (2 classes) to 500,000 (50 classes). For all three distributions, both indexes are fairly competitive in storage requirement. Due to space constraint, the results are not shown in this paper. We observe from these two experiments that the CH-tree storage requirement is to some extent affected by the directory size kept in the leaf nodes, while the H-tree storage requirement is affected by the number of classes involved and the height of each index.

In the query efficiency study, we examine two cases: different query ranges, and different number of classes being queried. In the first case, we let the number of values in the query range $[v_1, v_2]$, $\mathcal{NV}$, vary from 10% to 100% of the domain size of $\mathcal{D} = 10,000$, while set the number of classes $\mathcal{TC} = 10$ and the number of instances $\mathcal{N} = 500,000$. The result shown in Figure 8(i) illustrates that the H-tree performs well even when the query range is very large. In the second case, we set the query ranges at 20% and 100% of the domain size, and vary the number of query classes/subclasses from 1 to $\mathcal{TC}$ classes. In this case, $\mathcal{TC}$, the number of classes in the class hierarchy is set to 10, with 3 levels of classes in the class hierarchy. The result illustrated in Figure 8(ii) shows that the performance of H-tree improves significantly compared to the CH-tree, especially in cases where the number of classes involved in a query is small. Queries targeted at more specialized classes are common in OODB and such direct support is important to the performance of the system. The analytical results show that the H-tree is an efficient indexing structure.

4.2. Experiments

We implemented the CH-tree and H-tree on the EXODUS [CDR86] storage manager. We forced each node to be a small object, an object that occupies less than a page, to avoid generating of large object indexes. We also implemented the indexes in C++. The result reported in this section is based on those implemented on the EXODUS storage manager, using the same set of parameters used for analytical study. Each index node is treated as an object by the EXODUS storage manager and the size of the node is the same as the size of a physical page. As in the analytical study, we allocate two third of an internal node for the entries (K and B), and one third for the nested tree pointers (L) and backward links. For the CH-tree, the space is fully used for entries. Due to the space constraint, we would only report the query efficiency experiments.

A database of 500,000 ($\mathcal{N}$) instances is created and distributed to 10 classes ($\mathcal{TC} = 10$) of a class hierarchy of up to 3 level high. The indexed attribute values are constrained to the range of [1, 10000] ($\mathcal{D} = 10,000$), and the domains of classes totally overlap ($\mathcal{NC} = 10$). As in the previous simulations, different query ranges and different number of classes being queried are considered.



(i) (ii) (iii)

Figure 7: Index Size vs Number of classes with N = 500,000; (i) disjoint distribution, (ii) total inclusion distribution and, (iii) partial inclusion distribution.

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 average is taken. For the 100% search range ([1, 10000]), although a direct sequential scan of leaf nodes is possible, we search the indexes using the searching routine. The results are summarized in in Figure 9. In the first experiment (graph (i)), 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, 10000]). The results show that the CH-tree performance degrades faster than that of the H-tree as the queried range increases. In the second experiment, the number of classes involved in the range queries varies from 1 to 10 classes, with $\mathcal{NV}$ fixed at 20% and 100%. The results show that the H-tree is a much more efficient indexing structure when the number of classes involved is small. Both empirical results exhibit the same behavior to that obtained using the cost models. In these experiments (in fact in what we have performed so far), 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 would be tuned in future.

5. Conclusions

We have proposed in this paper a new hierarchical indexing structure called the H-tree. 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. Both analytical and empirical performance analysis indicate that the H-tree, when compared to the CH-tree, is a more efficient indexing structure for retrievals on a single class and on a hierarchy of classes. Two access methods that can be efficiently implemented using H-trees are: i) scanning for all instances of a class and its subclasses and, ii) 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. In [LLO91], we used H-trees to compute least-fix point in recursive query processing, by using one H-tree at one iteration of the semi-naive evaluation. The presented index nesting concept can be easily modified for other indexing techniques such as B-trees and m-way trees.

Apart from the distribution of data across classes, two other factors that may influence the performance of H-trees are the number of classes and the levels of class hierarchies. More experiments need to conducted to test the effectiveness of H-trees with different distributions and class hierarchies. We have taken the update cost into consideration during the design of the H-tree. However, we have yet to justify the effectiveness of the efficiency rules in reducing the update cost. Note also that H-trees provide a higher degree of concurrency than a single index scheme. These are two of the issues that will be further studied.

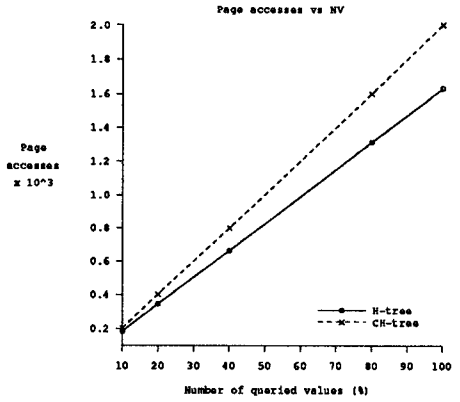
Acknowledgment

We would like to thank Chan Chee Yong for suggestions on improving the presentation and efficiency of algorithms.

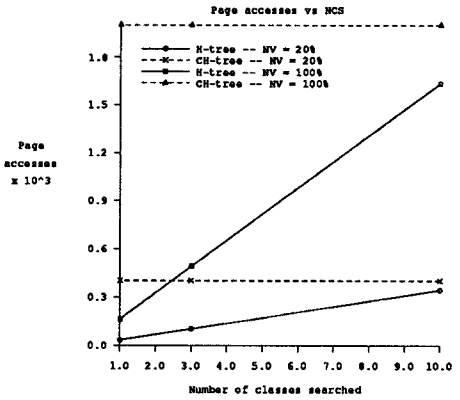
References

[BeK89] E. Bertino, W. Kim: Indexing techniques for queries on nested objects. *IEEE Trans. Knowledge and Data Engineering*, 1(2), 196-214 (1989).

[CDR86] M. Carey, D. DeWitt, J. Richardson, E. Shekita: Object and file management in the EXODUS extensible database system. *Proc. Int. Conf. on Very Large Data Bases*, Kyoto, Japan (1986).

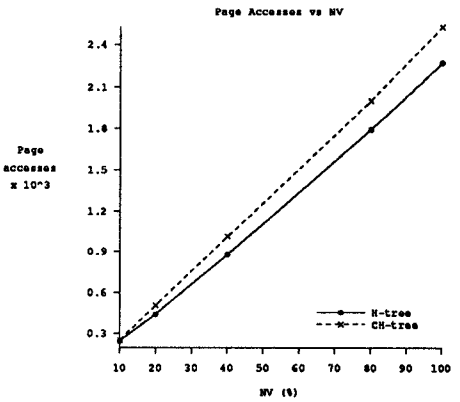


(i)

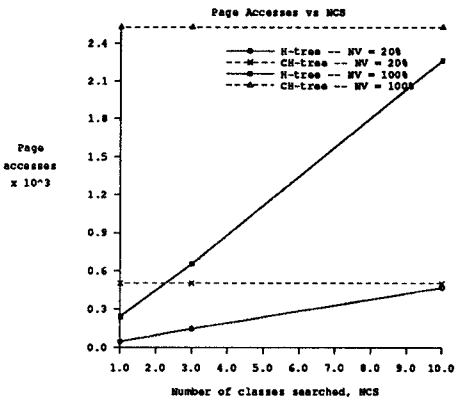


(ii)

Figure 8: Number of page accesses for range queries; (i) the number of values in the range queries varies from 10% to 100% of the domain values, (ii) the number of classes searched varies from 1 to 10 classes.



(i)



(ii)

Figure 9: Experiment Results: Number of page accesses for range queries; (i) the number of values in the range queries varies from 10% to 100% of the domain size, (ii) the number of classes in the search space varies from 1 to 10 classes.

[Com79] D. Comer: The ubiquitous B-tree. *ACM Computing Surveys*, 11(2), 121-137 (1979).

[KeM90] A. Kemper, G. Moerkotte: Access support in object bases. *Proc. ACM Intl. Conf. on Management of Data*, 364-374 (1990).

[KKD89] W. Kim, K. C. Kim, A. Dale: Indexing techniques for object-oriented database. In *W. Kim and F. H. Lochovsky (ed): Object-oriented Concepts, Databases, and Applications*, Addison-Wesley, 371-394 (1989).

[LLO91] C. C. Low, H. Lu, B. C. Ooi, J. Han: Efficient Access Methods in Deductive and Object-Oriented Databases. *Proc. Intl. Conf. on Deductive and Object-Oriented Databases*, Lecture Notes in Computer Science 566, 68-84 (1991).

[MeS86] D. Maier, J. Stein: Indexing in an object-oriented DBMS. *IEEE Proc. Intl. Workshop on Object-Oriented Database Systems*, Pacific Grove, G.A., 171-182 (1986).

[SuO82] P. Scheuermann, M. Ouksel: *Multidimensional B-trees for associative searching in database systems.* Information Systems, 7(2), 123-137 (1982).

[VKC86] P. Valduriez, S. Khoshafian, G. Copeland: Implementation of techniques of complex objects. *Proc. Intl. Conf. on Very Large Data Bases*, Kyoto, Japan, 101-110 (1986).