The hcC-tree: An Efficient Index Structure For Object Oriented Databases

B. Sreenath
Computer Science Dept.
IIT, Bombay 400 076
India

S. Seshadri
Computer Science Dept.
IIT, Bombay 400 076
India
seshadri@cse.iitb.ernet.in

Abstract

Object oriented database systems, in contrast to traditional relational database systems, allow the scope of a query against a class to be either the class itself or all classes in the class hierarchy rooted at the class. If object oriented databases have to achieve acceptable performance levels against such queries, we need indexes that support efficient retrieval of instances from a single class as well as from all classes in a class hierarchy. In this paper, we propose a new index structure called hcC-tree (hierarchy class Chain tree) that supports both kinds of retrieval efficiently. Moreover, the update cost of the index structure is bounded by the height of the hcC-tree. We have implemented hcC-trees along with H-trees and CH-trees (two other index structures that have been proposed in the literature) and report a detailed performance analysis of the three structures. The performance study reveals that hcC-trees perform much better than the other two structures under most circumstances. The balanced behaviour of hcC-tree under all kinds of queries and in the presence of updates shows that it is a promising index structure for the future.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment.

Proceedings of the 20th VLDB Conference
Santiago, Chile, 1994

1 Introduction

In object oriented database systems, a class can be specialized into a number of subclasses which in turn can be further specialized into subclasses to form a class-hierarchy. A query against a class in object oriented database systems can have two meaningful interpretations. The first is that the target of the query is the class itself while the second is that the target of a query is the class and all its subclasses or equivalently the target of a query is the class hierarchy rooted at that class. If object oriented databases have to achieve acceptable performance levels against such queries, we need indexes that support efficient retrieval of instances from a single class as well as from all classes in a class hierarchy.

In this paper, we propose an index structure called hcC-trees (stands for hierarchy class Chain tree since the index structure stores information in two kinds of chains - hierarchy chains and class chains as we will see shortly) that supports both kinds of retrieval efficiently. The main problem in designing a good index structure to support such queries is that the two interpretations of a query against a class pose conflicting requirements on the organisation of the index. To understand this better we will first classify the types of queries that can be posed to an object oriented system. Queries based on some attribute of a class or a class hierarchy can be either point queries or range queries. Point queries basically ask for all instances of a class or a class hierarchy with a particular value for the concerned attribute. Range queries, on the other hand, ask for instances of a class or a class hierarchy whose attribute value falls in a certain range. We therefore have the following four types of queries:

- CHP queries: This type of queries refer to point queries against all classes in a class hierarchy rooted at some class. CHP queries stands for

Class Hierarchy Point queries.

- SCP queries: This type of queries refer to point queries against a single class. SCP queries stands for Single Class Point queries.
- CHR queries: This type of queries refer to range queries against all classes in a class hierarchy rooted at some class. CHR queries stands for Class Hierarchy Range queries or equivalently Class Hierarchy Range queries.
- SCR queries: This type of queries refer to range queries against a single class. SCR queries stands for Single Class Range queries.

It can now be seen that for CHP and CHR queries it would be best if the object identifiers (henceforth called oids) of objects of all the classes in the class hierarchy for a given value of the indexed attribute are clustered together (Note that we are not talking about clustering of the objects themselves but we are talking about the ease with which the matching oids for a query can be retrieved). On the other hand, for SCP and SCR queries it would be best if the oids of objects of a single class for a given value of the indexed attribute are clustered together. This conflicting requirement makes the job of designing an efficient index structure harder.

The two main index structures proposed in the literature for this problem are called CH-trees and H-trees. Class Hierarchy Trees (CH-trees) was proposed by Kim et al [KKD89] and is based on B^+-trees [Com79] and essentially maintains a single index for all classes of the class hierarchy. It clusters the oids of objects of all classes in the class hierarchy for a given value of the indexed attribute. In the same paper they explored the performance of having a separate B^+-tree for each class in the class hierarchy and showed that CH-trees perform better in most cases. H-trees was proposed in [LOL92] and the central idea here is that one B^+-tree index per class in the class hierarchy is maintained but the indexes are nested according to their subclass-superclass relationship. Essentially, the H-trees cluster the oids of objects of a single class with a given value of the indexed attribute together. We will describe the H-trees and CH-trees in more detail in Section 4. The new index structure, hcC-trees, that we propose tries to cluster the oids in both the above ways so that all queries can be answered efficiently.

We have implemented all the three index structures on top of a storage manager that treats every file as a sequence of pages. We have conducted an extensive performance study of these three index structures and we show that hcC-trees perform consistently as well as or better than CH-trees and H-trees for the four types of queries above for various data distributions.

There has been recent interest in other types of indexes in object oriented database systems. Path indexes have been studied in [BK89, KM90, LLOH91] which are indexes over class composition hierarchy rather than the class hierarchy. Signature files have been studied as set access facility in OODB's recently in [IKO93].

The remainder of the paper is structured as follows: Section 2 describes the structure of hcC-trees while Section 3 describes the search and update algorithms for the hcC-tree. Section 4 compares the performance of the three index structures CH-tree, H-tree and hcC-tree. We conclude in Section 5.

2 The Structure of hcC-tree

In this section, we describe the structure of hcC-trees. A hcC-tree has three types of nodes:

1. Internal nodes
2. Leaf nodes
3. Oid nodes

The internal nodes and leaf nodes are somewhat similar to the internal nodes and leaf nodes of B^+-trees respectively. The interanl nodes form the upper levels of the tree while the leaf nodes are the last but one level of the tree. The oid nodes are at the last level (one level below the leaf nodes) and contain the actual information in the case of hcC-trees. To maintain an index on a class-hierarchy (on a common attribute), only one hcC-tree is needed (similar to CH-trees [KKD89]). We will now explain the three types of nodes in greater detail. We will assume for the sake of the ensuing discussion that the class hierarchy we are indexing consists of n classes.

2.1 Structure of Oid-Nodes

The oid-nodes are at the last level of the hcC-trees as we just mentioned. Actually, the oid-nodes are divided across $n+1$ chains of oid-nodes. Corresponding to each class in the hierarchy there is a chain of oid-nodes which is called a class chain while there is one chain of oid-nodes that corresponds to all the classes which is called the hierarchy chain. The structure of the oid-nodes of the two types of chains (class chains and hierarchy chain) differ slightly from each other. Oid-nodes of a class chain store the oid of the objects of that particular class alone. Each entry in a class-chain oid-node is of the form <key, oid list>. Figure 1 shows a typical entry in a class-chain oid-node. Each entry consists of a key value and the oids of the objects of that class which have that key value for the indexed attribute. The oid nodes of the hierarchy chain store

the oid of objects belonging to all classes. An entry in the hierarchy chain is of the form <key, set of oid list>, as illustrated in Figure 2. For a given key value, the hierarchy chain oid-node consists of n oid-lists. Each oid-list consists of oids of objects of that class which have the value k for the indexed attribute. The oid-nodes of each of the $n+1$ chains are chained together to facilitate traversal of oid-nodes from left to right.

<key,oid list>

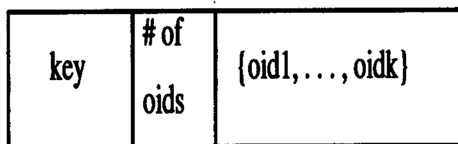

Figure 1: A typical class-chain oid-node entry

<key, set of oids>

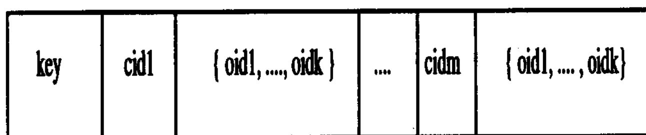

Figure 2: A typical hierarchy-chain oid-node entry

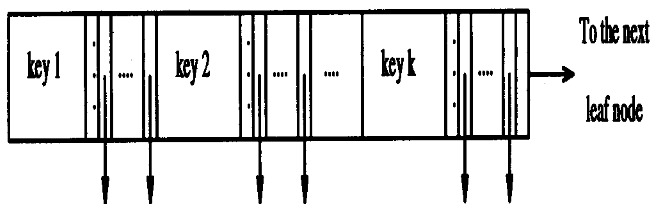

Figure 3: Leaf node

2.2 Structure Of Leaf Node

A typical hcC-tree leaf node is shown in Figure 3. Each entry in the leaf node contains a key k, a bit map and a set of pointers. The bit map consists of n bits (one bit per class) and a bit corresponding to a class in the bit map is set if and only if there is at least one object belonging to that class having the key value k. Let us assume that m bits in the bitmap are set for a given value k. Then, the set of pointers consists of $m+1$ pointers, m of these to oid-nodes of class chains and one to an oid-node of the hierarchy chain. The oid-nodes pointed to will of course contain the oids of objects whose indexed attribute has value k.

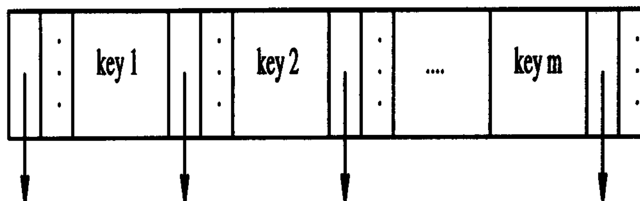

Figure 4: Internal node

2.3 Structure of Internal Node

An internal node of an hcC-tree is illustrated in Figure 4. The internal node structure is similar to the structure of a B$^+$-tree internal node in that there are m keys and $m+1$ pointers to nodes at a lower level. However, there is some additional information stored in the internal nodes of hcC-trees. Associated with each of the $m+1$ intervals (determined by the m keys) of the internal node, a bitmap of n bits is stored. A bit (in the bitmap) corresponding to a class for an interval is 0 if and only if there exists no object belonging to that class which has a value for the indexed attribute that lies within the interval. The fanout for an internal node is between d and $2d$, similar to B$^+$-trees, where d is the order of an internal node.

We will now look at an example hcC-tree of Figure 5 which is partially populated. The figure depicts a root node which is the only internal node in this tree. There are two leaf nodes at the next level and the rest are oid-nodes. In the figure we assume that the class hierarchy contains two classes. The root node has only one key and the bitmap for the second interval of the root node is {10} which means that in this interval there are some objects belonging to class 1 and no objects belonging to class 2. In the first leaf node (with keys 10 and 20), for the key 10 the first pointer is pointing to an oid node which has the oids of the objects belonging to class 1 and which have the value 10 for the indexed attribute and the second pointer is pointing to the oid node which has objects belonging to class 2 and with the value 10 for the indexed attribute. These are pointers to class chains for classes 1 and 2 respectively. The next pointer for the key 10 points to the oid node which stores the oids of objects of both the classes 1 and 2 with a key value of 10. This is the pointer to the hierarchy chain. Since there are no objects belonging to class 2 with the key value 20, its bit is set to 0 and there is no pointer corresponding to it in the first leaf node. In general, many entries in

a leaf node and in fact many leaf nodes can point to the same oid node. In the figure, for the keys 10 and 20, the first pointer points to the same oid node.

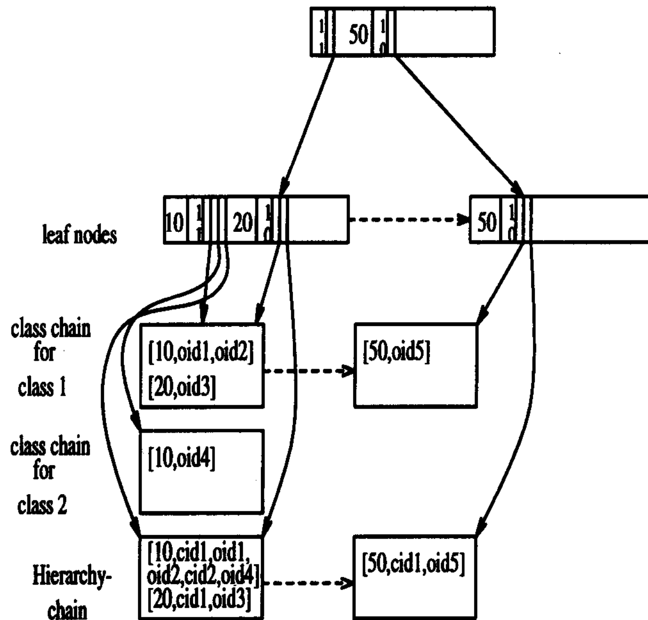

Figure 5: hcC-tree

3 Search and Updates

3.1 Searching

Searching an hcC-tree for a particular value is similar to that of B+-trees. We outline the search algorithm in this section.

Algorithm Search

Search (cid_{search}, v_1, v_2)

Input: cid_{search} - list of classes that are to be searched.
v_1 - lower bound of the range of search values.
v_2 - upper bound of the range search values.
$v_1 = v_2$ for point queries.
Output: list of oids whose indexed attribute values fall within $[v_1, v_2]$.

1. Case type of query
/* The type can be figured out from cid_{search}, v_1, and v_2 */

   SCP: Search to the leaf node that contains the value v_1. Search the class-chain oid node of this class for the value v_1 and return matching oids. The search may occasionally end early as along the search path, the bit corresponding to this class may be zero.

   CHP: Search to the leaf node that contains the value v_1. Search the hierarchy node for the value v_1 and return the matching oids belonging to classes of cid_{search}. The search may end early if it is detected that all classes do not have oids corresponding to v_1 somewhere along the search path.

   SCR: O_{node}=Search-Oid-Node(v_1, v_2, cid_{search}). Search the oid nodes starting from O_{node} in the oid chain for this class for matching oids.

   CHR: If the number of classes in cid_{search} is equal to the number of classes in the class hierarchy then use hierarchy chain to retrieve matching oids. Otherwise, use the individual class chains.

Search-Oid-node (v_1, v_2, cid)

Input: $[v_1 - v_2]$ - range being searched
cid - class-id of the object.
Output: Returns the oid-node of class cid from where the chain should be scanned for this range

1. Use the bitmaps to get to the right oid node. This will look at exactly one node in each level of the tree and is therefore as efficient as a normal B+-tree search.

3.2 Insertions

The insert algorithm is outlined below:

Algorithm Insert

Insert (v, oid, cid)

Input: v - key value of new object to insert.
oid - oid of new object to be inserted.
cid - class-id of the new object.

1. L_i= Leaf node in whose interval v falls.
2. If v not in L_i
   Add v to L_i
   Set bit corresponding to cid and allocate space for two pointers
   Handle splits like normal B+-tree split
   Let L_i be the leaf node that contains v finally
3. If v in L_i and bit of cid for v is zero
   Set bit corresponding to cid and allocate space for one more pointer
   Handle splits like normal B+-tree split
   Let L_i be the leaf node that contains v finally
4. $O_{inserted}$ = Class-Chain-Oid-Insert(v,oid, cid).
5. Update L_i to reflect that the oid-node for v (of the class chain corresponding to cid) is $O_{inserted}$
6. $O_{inserted}$ = Hier-Chain-Oid-Insert(v,oid, cid).

7. Update L_i to reflect that the hierarchy chain oid-node for v is $O_{inserted}$

The insert algorithm has three main phases. In the first phase it finds the leaf node in whose interval v falls and then adds v to the leaf (if it is not already there) or adds a pointer for v corresponding to cid in this leaf node (if the pointer is not already present). All splits are handled like normal B^+-trees. The bitmaps are updated (if need be) as the algorithm descends down from the root towards L_i and can be similarly updated while going back towards root in case of a split. After the first phase there will never be a need to split a leaf node or an internal node as all the required space has been allocated. In the second phase it adds the oid to the class chain while in the third phase it adds the oid to the hierarchy chain. We will describe Class-Chain-Oid-Insert only. The algorithm for Hier-Chain-Oid-Insert is similar.

Class-Chain-Oid-Insert (*v, oid , cid*)

Input: v - key value of the object.
oid - oid to be inserted
cid - class-id of the new object.
Output: Returns the class-chain oid-node where oid is inserted

1. O_i = Find-Oid-Node(v,cid)
2. $O_{inserted}$ = Oid-Insert(v, oid, O_i)
3. return $O_{inserted}$

Class-Chain-Oid-Insert first finds an oid node O_i into which it can insert this oid and then inserts it into O_i. This insertion may cause O_i to split and the oid-node where this oid is finally inserted is $O_{inserted}$.

Find-Oid-Node (*v,cid*)

Input: v - key value of the object.
cid - class-id of the new object.
Output: Returns the oid-node where we may insert the oid.

1. If the class cid has no object at all, then create a new oid node and return it.
2. If the class cid has no object with value smaller than or equal to v for the indexed attribute, let v_2 be the smallest value larger than v for which the class cid has an object with its indexed attribute value v_2. Return the oid node which contains the value v_2.
3. If the class cid has no object with value v for the indexed attribute, let v_1 be the largest value smaller than v for which the class cid has an object with its indexed attribute value v_1. Return the oid node which contains the value v_1.
4. Return the oid node (of the class chain corresponding to cid) in which v is found.

Oid-Insert (*o_{node},v,oid*)

Input: o_{node} - the oid node in which *oid* may be inserted.
v - the key value.
oid - the object to be inserted.
Output: The oid node in which the oid is finally inserted.

1. If space for inserting v and the oid
   v_{large}=max(largest value in O_{node},v);
   $O_{toinsert}$=oid node immediately after O_{node} in the chain.
   If no space for v_{large} and its oids in $O_{toinsert}$
      $O_{toinsert}$=Get a new oid node.
      Hook up $O_{toinsert}$ into the class chain
   Insert v_{large} and its oids in $O_{toinsert}$;
   If ($v_{large} \neq v$)
      Delete v_{large} from o_{node}.
      Insert v in O_{node}.
      L_f=leaf node in whose interval v_{large} falls.
      Update L_f to reflect that the oid node for v_{large} for *cid* is $o_{toinsert}$.
      Return o_{node}.
   else Return $o_{toinsert}$.
   else Insert v in o_{node}.
   Return o_{node}.

The first phase of the insert is similar to the insert algorithm of the B^+-trees. The cost is therefore the same except that occasionally updating bitmaps may cause some nodes to be written back to disk. The second and third phase are additional in hcC-trees. These phases in most cases would not cost anything as they will look at the same nodes looked at during the first phase. Occasionally, in the second phase a maximum of two extra traversals of the tree may be needed (one for Find-Oid-Node and the other for Oid-Insert). In the third phase, finding the hierarchy chain oid node in which the oid can be inserted will not cost anything as this can be ascertained from L_i in algorithm Insert. However, in the third phase an extra traversal may be needed while inserting the oid into the hierarchy chain oid-node. Thus, a maximum of three extra traversals may be needed over B^+-trees but in most cases there is no extra overhead. Therefore, even in the worst case, the cost of the insert algorithm is bounded by the height of the tree.

3.3 Deletion

The delete algorithm is described below:

Algorithm Delete

Delete (*v,oid,cid*)

Input: *v* - indexed value to delete.
oid - the object to be deleted.
cid - class to which *oid* belongs to.

1. L_i=leaf node that contains the value *v*.
2. O_1=oid node for the class *cid* for the value *v*.
3. O_{node}=Delete-Oid(v,O_1,cid);
Similarly delete oid from the Hierarchy chain also.
4. If $O_{node} \neq$ Null Return;
else
Set the bit for the class *cid* for *v* to zero;
Delete pointer corresponding to class *cid*
If all the bits are zero for *v*, delete it completely from the leaf node.
If underflow occurs handle like that of a normal B^+-tree and update the bitmaps of the nodes affected by deletion.

Delete-Oid (*v, oid,* O_1)

Input: *v* - indexed value to delete.
oid - the object-id to be deleted.
O_1 - the oid node from which *oid* is to be deleted.
Output: Returns Null if there are no more objects belonging to *v* in O_1 after the deletion of *oid*.

1. Delete *oid* from the oid list of *v* in O_1.
2. If there are no more objects for *v*
Delete *v* itself from O_1.
If there are no more entries in O_1, delete O_1 from the oid chain and adjust the pointers.
Return Null.
else
Return O_1.

4 Performance Analysis

In this section, we report on the implementation details of the three index structures and the results of the experiments we conducted to study their performance. Before going into the details of the implementation we describe the structure of CH-trees and H-trees briefly.

CH-tree Structure

The CH-tree maintains only one index for all the classes in a class-hierarchy. The non-leaf nodes of the CH-tree are the same as that of B^+-trees and only the leaf nodes are different . The leaf nodes store for each key value the oids of the objects of all the classes in the Hierarchy which have that value for the indexed attribute. In other words, the leaf node structure of CH-trees is similar to the hierarchy chain oid-node structure of the hcC-trees. Searching for values in CH-trees on a single-class is treated in the same way as searching for values on a hierarchy of classes. The searching and update procedures in CH-trees are similar to that of B^+-trees.

H-tree Structure

To index a class-hierarchy on a common attribute, one H-tree is maintained for each class of the hierarchy and these trees are nested according to their superclass-subclass relationships, to form a hierarchy of index trees. The H-tree for each class is similar to that of B^+-trees. The H-tree of the root class of the hierarchy is nested with the H-trees of all its immediate subclasses, and the H-trees of the subclasses are nested with the H-trees of their respective subclasses and so on. The H-trees are similar to single-class indexing in that a separate tree is maintained for each class but they are different in that the different trees are not independent due to the nesting of trees. The nesting facilitates jumping into the middle of a subclass tree without going through its root node. At the same time H-trees also facilitate the independent access of any of the subclass H-trees. The leaf nodes of H-trees are similar to that of B^+-trees. However, in an internal node N of a H-tree, apart from the usual discriminating key values and child node pointers(B), pointers(L) pointing to subtrees of nested H-trees are stored. The L-pointers also consist of the minimum and maximum values of the subtree of nested H-trees. The rules for nesting H-trees are described in [LOL92]. 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. For search on multiple classes or the entire class-hierarchy, the search begins on the H-tree of the root class, and follows the pointers to search the nested subtrees of classes of interest. The algorithm is described in [LOL92].

Experiments

We implemented the three index structures in C on top of a Storage Manager that treats a file as a collection of pages. For all the three index structures, each individual index structure is a file and a node corresponds to a page of the storage manager. The page size of the storage manager was fixed at 4K bytes. The performance metric in all our experiments was the number of page accesses.

The relative performance of the index structures depends on a lot of factors. One important factor is the distribution of key values across the classes of a class

hierarchy. We consider the following indexed attribute distributions:

1. complete overlap – All classes in the class hierarchy have objects corresponding to all key values appearing in the leaf nodes of the indexes.
2. partial overlap - the domains of any two classes are partially overlapping, i.e. only some of the classes have objects with a particular indexed value. We consider only 50% overlap, i.e., half of the classes in the hierarchy will have objects with a particular indexed value.

For each of the above data distributions, we study the performance of the index structures for the four types of queries identified earlier.

Another important factor on which the relative performance of the index structure depends is the number of classes in a class hierarchy. We will first look at the effect of the data distribution and then the effect of varying the number of classes on the relative performance of the index structures.

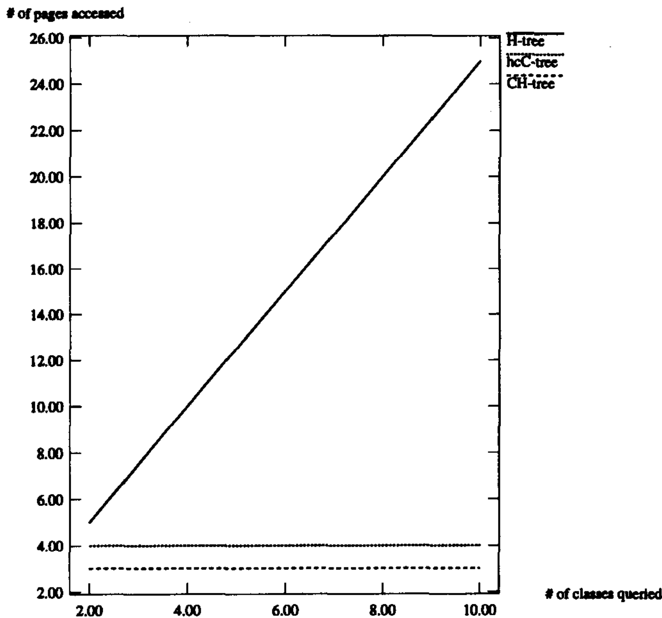

Figure 6: CHP query, Complete Overlap, 10 classes

4.1 Effect of Data Distribution

4.1.1 Complete Overlap

In this section we study the performance of the index structures for the complete overlap case. We created a database of 1.5 million instances. There were 10 classes in the class hierarchy and each class had 150,000 instances. The class hierarchy was 4 level deep with each class having two or less than two immediate subclasses. The performance for the four types of queries under this setting are described below:

SCP Queries

The number of page accesses for all indexes is the height of the index in this case. The height of CH-trees and H-trees are 3 for the given configuration while it is 4 for hcC-trees.

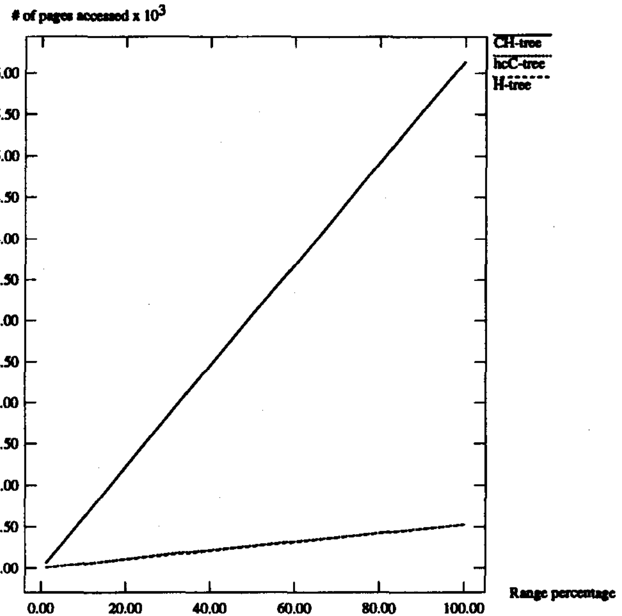

Figure 7: SCR query, Complete Overlap, 10 classes

CHP Queries

In this experiment, we varied the class against which the class hierarchy point query was targeted and hence the number of classes that were actually queried. Figure 6 shows the results of CHP queries as we vary the number of classes against which the query was targeted. It is clearly seen that CH-trees and hcC-trees are immune to the number of classes being queried while the number of page accesses for H-trees grows linearly with the number of classes.

SCR Queries

In this experiment, we varied the range of the SCR query. Figure 7 shows the results of SCR queries as the range is varied from 1% to 100% of the domain of

the indexed attribute. hcC-trees and H-trees are very close in their performance while CH-trees perform very poorly because a significant portion of their leaf nodes contain irrelevant information for SCR queries.

CHR Queries

We could vary the range of the query as well as the number of classes against which the query is targeted for CHR queries. We varied the number of classes from 1 to 10 and the range from 1% to 40%. Table 1 shows the results of this experiment. Once again hcC-trees perform the best while CH-trees are good only if the entire class hierarchy is queried. The H-trees do not perform as well as hcC-trees because for each class they have to traverse the internal nodes of the H-tree corresponding to that class at least partially.

Discussion of Complete Overlap results

In all cases, hcC-trees are the best or close to the best. CH-trees are good only if the target of a query includes all classes in the class hierarchy. H-trees are good only if the target of a query is a single class. hcC-trees are good under all circumstances which is a desirable characteristic given that one can not expect queries to be against a single class or all classes all the time.

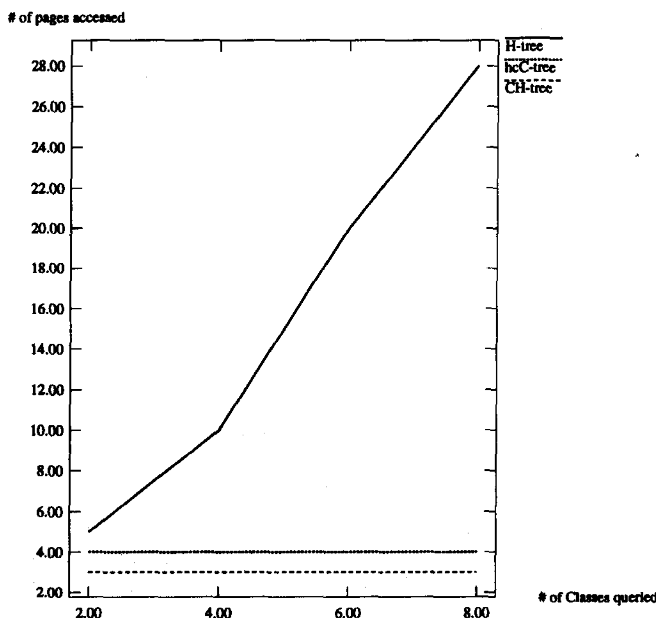

Figure 8: CHP query, Partial Overlap, 8 classes

4.1.2 Partial Overlap

In this section, we study the performance of the index structures for the partial overlap case. In this experiment, we created a database of 1.2 million instances. There were 8 classes in the database. For each key in between 1 and 300,000, four classes were chosen randomly to have objects with their indexed attribute equal to the chosen key value. Therefore, each class has approximately 150,000 instances. The class hierarchy was 3 level deep with each class having two or less than two immediate subclasses. The performance for the four types of queries under this setting are described below:

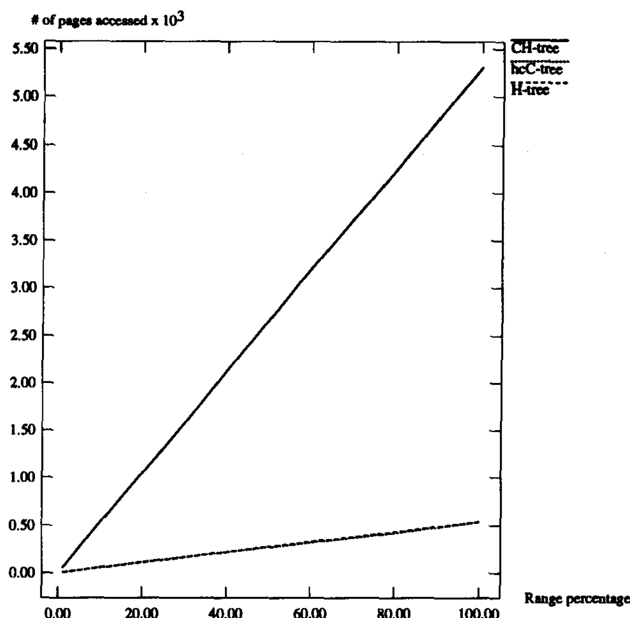

Figure 9: SCR query, Partial Overlap, 8 classes

SCP Queries

The results in this case were similar to complete overlap for CH-trees and H-trees, i.e., the number of page accesses was equal to the height of the tree which was three. In the case of hcC-trees, though the height of the tree was 4, the average number of page accesses (over 100 random searches) was 3.5. The reason is that occasionally the search in a hcC-tree can terminate early, if it can be determined at the internal nodes or the leaf nodes (using the bitmap) that the search would be unsuccessful for this point query.

# Classes queried	Index Used	1 %	2 %	5 %	10 %	20 %	40 %
	H-tree	9	15	31	57	116	231
1	CH-tree	63	119	300	606	1228	2459
	hcC-tree	9	15	32	58	115	230
	H-tree	18	30	69	122	235	468
2	CH-tree	63	119	300	606	1228	2459
	hcC-tree	15	27	64	116	230	460
	H-tree	35	59	137	245	475	940
4	CH-tree	63	119	300	606	1228	2459
	hcC-tree	27	51	128	232	460	920
	H-tree	52	88	205	368	708	1406
6	CH-tree	63	119	300	606	1228	2459
	hcC-tree	39	75	192	348	690	1380
	H-tree	86	146	347	611	1180	2332
10	CH-tree	63	119	300	606	1228	2459
	hcC-tree	63	123	320	580	1150	2300

Table 1: CHR query,Complete overlap, 10 classes

CHP Queries

In this experiment we varied the class against which the class hierarchy query was targeted and hence the number of classes that the query was actually targeted. Figure 8 shows the results of CHP queries, as we vary the number of classes against which the query was targeted. As in the case of complete overlap, CH-trees and hcC-trees are not sensitive to the number of classes being queried. The H-trees, however perform worse than the complete overlap case. The reason is as follows: Recall that the H-tree reserves one-third of the space in a node for L-pointers (pointers to subclass nodes and information about the maximum and the minimum values of the subclass node). It turns out that for the partial overlap distribution, this space is not enough for L-pointers and hence we need to have overflow nodes for these L-pointers. This, in turn, increases the number of page accesses for the CHP queries, since the L-pointers have to be searched to go from a superclass to a subclass.

SCR Queries

The results of this experiment does not differ much from the complete overlap case and is therefore omitted.

CHR Queries

In this experiment we varied the range of the query as well as the number of classes against which the query is targeted. We only report results of an experiment in which the range is fixed at 1% while the number of classes targeted was varied from 1 to 10. Figure 10 shows the results of this experiment. Once again hcC-trees perform the best while CH-trees are good only if the entire class hierarchy is queried. The H-tree performance in this case is marginally worse than that of complete overlap. The reason is as follows: Consider three classes A, B and C where C is a subclass of B which in turn is a subclass of A. During a class hierarchy search on A, the H-tree for class B would be entered at some node (say N). In order to get the matching oids of class B all that is needed is to traverse to the corresponding leaf and retrieve them, but in order to get to the H-tree of class C, the subtree rooted at N and the ancestors of N may have to be searched for L-pointers. As suggested in [LOL92] we have bitmaps per class to say whether or not a node has a L-pointer but still quite a few nodes may have to be scanned. This leads to further degradation in performance. The reason this was not as much an issue in the complete overlap case is that the trees for all classes are identical and the nesting is likely to be perfect.

Discussion on Partial Overlap

It is seen that H-trees seem to suffer in the partial overlap case because of improper nesting. CH-trees would also perform relatively badly if the ranges of the individual classes were disjoint but a single class query was looking for oids of objects in a range in which the queried class has no matching oids. CH-trees would search the entire queried range and then report that no matching oids are present while H-trees

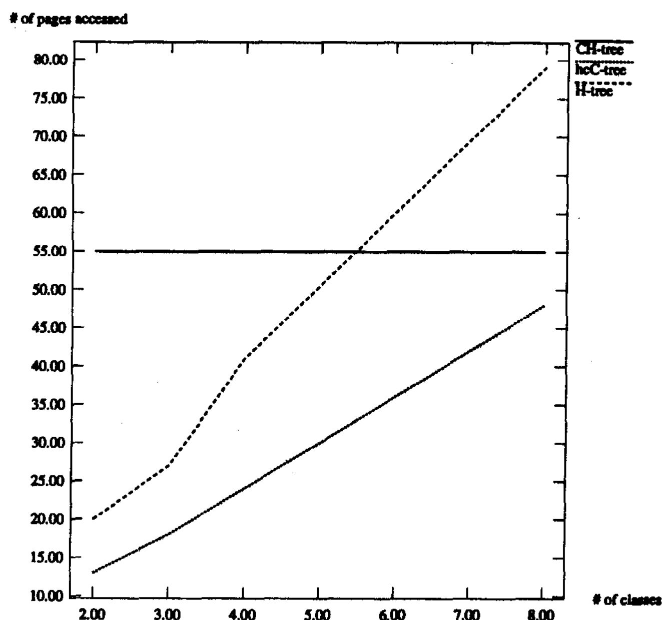

Figure 10: CHR query, Partial Overlap, 8 classes

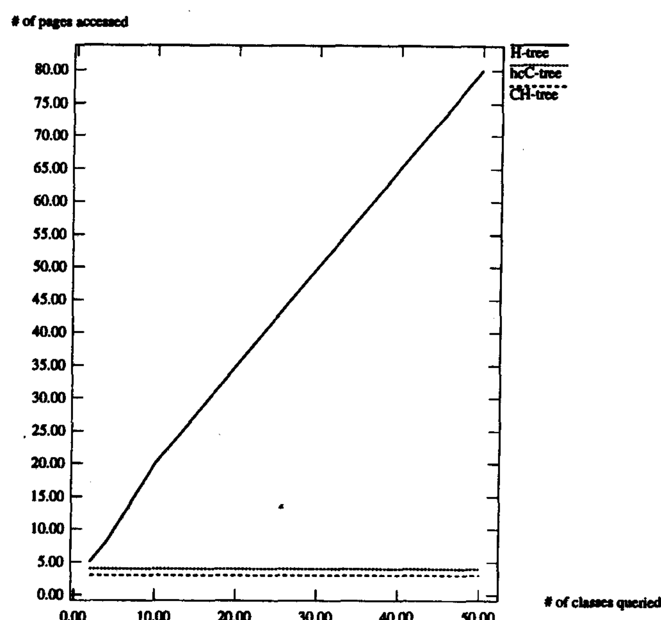

Figure 11: CHP query, Complete Overlap, 50 classes

and hcC-trees would recognise this fact rather early. So, both H-trees and CH-trees are more dependent on the actual data distribution than hcC-trees.

4.2 Effect of varying the number of classes in the class-hierarchy

In this set of experiments we had 50 classes in the class-hierarchy. The number of classes significantly affects the CH-tree since for single class queries the proportion of irrelevant information grows as the number of classes grows. For the H-trees, the greater the number of classes, the more the jumps into other nested trees and this makes the class hierarchy queries more expensive. We only considered the complete overlap case and each class had 30,000 instances. Therefore, there were totally 1.5 million objects in the database. We do not report the results on SCP, SCR and CHR queries for this setting. SCP and SCR are very similar to the results with 10 classes in the class hierarchy. We do not report on CHR as it does not throw up anything new that CHP (described below) does not throw up. We only report the results of CHP queries.

CHP Queries

We report the results of an experiment in which the number of classes queried is fixed at 50 while the range is varied. Figure 11 shows the results of this experiment. CH-trees and hcC-trees perform the best in this experiment. This shows that as the number of classes in the hierarchy grows H-tree performance degrades for CHP queries since there are more jumps out of superclass H-trees into subclass H-trees.

5 Conclusion

Queries in object oriented database system can be against a class or the class hierarchy rooted at a class. This poses conflicting requirements on how an index associates oids to key values. We proposed a new index structure, called hcC-trees, that solves the conflicting requirements by superimposing two different oid clustering methods on top of a single B^+-tree like structure. The tree had to be designed carefully to make sure that while the access efficiency for search queries is good, the update cost of the index structure is bounded by the height of the tree.

Our implementation of hcC-trees, along with CH-trees and H-trees, revealed that hcC-trees are fairly easy to implement and in fact simpler than H-trees since there is no nesting of trees. The experiments showed that hcC-trees consistently performed well for all types of queries. In contrast, H-trees did not perform as well for CHR and CHP queries while CH-trees did not perform as well for SCR queries.

Acknowledgement

We would like to thank Rohit Dube for building on our earlier code and implementing H-trees.

References

[BK89] E. Bertino and W. Kim. Indexing techniques for queries on nested objects. *IEEE Transactions on Knowledge and Data Engineering*, 1(2), June 1989.

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

[IKO93] Yoshiharu Ishikawa, Hiroyuki Kitagawa, and Nobou Ohbo. Evaluation of signature files as set access facilities in oodbs. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Washington D.C., May 1993.

[KKD89] W. Kim, K. C. Kim, and A. Dale. *Indexing Techniques for Object Oriented Databases*, pages 371–394. Addison Wesley, 1989.

[KM90] A. Kemper and G. Moerkotte. Access support in Object Bases. In *Proc. ACM Intl. Conf. on Management of Data*, pages 364–374, 1990.

[LLOH91] C. C. Low, H. Lu, B. C. Ooi, and J. Han. Efficient access methods in deductive and object-oriented databases. In *Proc. Intl. Conf. on Deductive and Object- Oriented Databases*, 1991.

[LOL92] C. C. Low, B. C. Ooi, and H. Lu. H-trees: A Dynamic Associative Search Index For OODB. In *Proceedings of the 1992 ACM-SIGMOD Conference on the Management of Data*, pages 134–143, June 1992.