Indexing Object Database using HC-Tree

Uma Perumal, Maha Ibrahim Shaabi

Abstract: ***One of the recent applications of object technology is in the area of databases. One of the stumbling blocks in the commercial development and deployment of object databases is the lack of an efficient indexing technique. The properties of object databases make the task of development of an indexing technique all the more difficult. This paper discusses the development of an indexing technique for object databases. A new indexing technique based on a new structure, HC-Tree has been proposed. Performance analysis has been conducted, and experimental and analytical results indicate that the HC-Tree is an efficient indexing structure for object databases. The performance of the HC-Tree has also been compared with that of the other popular existing techniques - CH-Tree, H-Tree and hcC-Tree.***

Keywords: Object Databases, Indexing, Query Retrieval, Data Structure.

I. INTRODUCTION

During the past several years, application of object-oriented concepts has become an important topic of research in a number of disciplines in Computer Science such as databases, programming languages, and knowledge representation and computer architecture. Object technology has evolved in three different disciplines: first in programming languages, then in artificial intelligence and then in databases due to the increasing demand for sophisticated data modeling capabilities by many database applications, object databases have recently attracted a significant amount of attention. The very popular relational model is inefficient when applied to complex domains like Geographic Information System, CAD/CAM and graphic databases. In such complex domains, object databases provide ideal solution to data modeling.

The primary motivation for this work has been to design an indexing structure for object databases that is an efficient as the B-Tree [C79] is for relational databases.

Indexing in Object Databases

Indexing in object databases is to be treated distinct from indexing in conventional databases, due to the object-oriented characteristics of the data model. Object-oriented data models offer additional features such as class hierarchies, inherited attributes and class compositions, which make the task of querying and indexing difficult.

Queries in an object database may be classified based on how the values are specified, and the target of the query. Based on how the query values are specified, queries may be classified into point queries and range queries. One major difference between conventional database and object database is that in an object database, a class can be specialized into number of subclasses.

Revised Manuscript Received on February 24, 2020.
Correspondence Author*
Uma Perumal*, Department of Computer Science, Jazan University, Kingdom of Saudi Arabia.
E-mail: prmluma@gmail.com
Maha Ibrahim Shaabi, Department of Computer Science, Jazan University, Kingdom of Saudi Arabia.

This implies that the access scope of a query against a class may be the instance of the class or may include instances of all classes derived directly or indirectly from that class. Bases on the target, queries may be classified into single-class queries and class-hierarchy queries.

In order to support these characteristics of object databases, an index of and object database must meet the following requirements:

i. The index must support efficient retrieval of instances of a single class.
ii. It must also support efficient retrieval of instance from class hierarchy.

Indexing techniques defined in the framework of object databases can be classified into structural - based on object attributes and behavioral - based on method execution profiles. The HC-Tree is a structural indexing technique.

Some of the common structural indexing methods are:

Single-class indexing - where on index is maintained for a combination of attributes of a class, irrespective of its position in the class hierarchy. Example: H-Trees [LOL92]

Class-hierarchy indexing - where a single secondary index is maintained for a combination of a class hierarchy. Example: CH-Trees [K90]

Two less popular techniques are nested attribute indexing - where the attribute indexed is an indirect, nested attribute of the class, and behavioral indexing - which is based on pre-computing or caching results of methods and maintaining an index. Both these methods are complex and have not been studied in detail in the literature.

II. EXISTING TECHNIQUES

The three main index structures proposed in the literature are CH-Tree, H-Tree and hcC-Tree.

CH-Tree (Class Hierarchy tree) was proposed by Kim et al [K90]. It is based on the B+-tree. A single index is maintained for all classes in a class hierarchy. The OIDs of objects of all classes in the class hierarchy for a given value of the indexed attribute are clustered together in the index.

H-Tree (Hierarchy Tree), which was proposed by Low et al [LOL92], is a dynamic associative search index for OODB. It is a hierarchical indexing structure based on the B+-tree. One H-Tree structure is maintained for each class in the class hierarchy and these trees are nested according to their superclass-subclass relationships.

HcC-Tree (hierarchy class Chain tree) was proposed by Seshadri et al [SS94]. It satisfies the requirements for indexes in object databases by superimposing two different OID clustering methods on top of a single B+-tree like structure. The hcC-tree maintains the OIDs of instances of classes in two chains - the class chain and the hierarchy chain. This enables the hcC-tree to support both class-hierarchy queries and single-class queries.

III. THE HC-TREE

In this paper, we propose a new structure for indexing object databases, and devise an indexing technique based on this structure. Our indexing structure is called HC-tree (Hierarchy-Chain Tree). It is a structural indexing structure, and is similar to the CH-tree in that it maintains a single index for all the instances of all classes in a class hierarchy. One major advantage of this structure is that it maintains the superclass - subclass relationship in a seamless manner.

3.1 HC-Tree structure

The proposed technique is based on the HC-tree structure. The HC-tree is based on the B+-tree structure. It has been proved in the literature that the B+-tree is an efficient indexing structure. Hence we have made use of the B+-tree as the underlying structure and have modified it to accommodate the requirements of object database indexing.

A HC-tree contains nodes of two types: (a) Internal nodes, and (b) Value nodes (leaf nodes)

Internal Nodes

The internal nodes of the HC-tree are similar to the internal nodes of the B+-tree. Each internal node has 'm' keys and 'm+1' pointers to nodes at lower levels. Each internal node of order'd' contains at most '2d' keys and '2d+1' pointers. Actually, the number of keys (and thereby the number of pointers), may vary from node to node, but each node must have at least'd' keys and'd+1' pointers. As a result, as in the case of the B+-tree, each node is at least half full. Thus the fan out for an internal node is between'd' and '2d+1'.

In addition to the keys and the pointers, an internal node also maintains a bitmap, with one bit for each class in the hierarchy, to represent the presence of OIDs of the instances of the class in the sub tree starting at the particular internal node. The bit corresponding to a class is 1 if the class has at least one instance with the indexed attribute value in the range of values represented in the sub tree rooted at the node, and 0 otherwise. The maintenance of bitmaps significantly improves the search process by eliminating the need for unnecessary traversals into the depth of the tree.

Value-nodes

The value-nodes are the leaf nodes of the HC-tree. Each value-node has a key value, and stores a list of instances of the indexed classes, having their indexed attribute value equal to the key. The value-nodes also have bitmaps, which serve the same purpose as bitmaps in the internal nodes. As in the case of B+-tree, the value nodes are linked together to form a chain of value-nodes, to speed up sequential search. Actually, each value-node maintains a list of the classes having at least one instance with its key value as the value of the indexed attribute. For this purpose, CID-nodes are used.

CID-nodes

The CID-nodes are used to store the OIDs of the instances of a particular class. The CID-node corresponding to a particular class maintains a list of the OIDs of its instances, which have their indexed attribute value equal to the key value of the value-node. For the purpose of representing the superclass-subclass relationship in the class hierarchy, each CID-node maintains two pointers - one to the CID-node of its left-most child representing the sub-class, and the other to the next sibling representing the children of its superclass. The CID-nodes are nested according to the position of the classes in the class hierarchy. This is a very efficient method of representing the class hierarchy as it makes optimal usage of storage space and is also very dynamic.

OID-nodes

Each CID-node in the class list of a value-node, in turn acts as the head of a list of its instances, which have the indexed attribute value equal to that of the key value. The instances (OIDs) are represented using the OID-nodes, with one OID-node representing the identity of an object. It has to be noted here that only the identity of an object is stored in the index. Its state and behavior are stored only in the database. Each database instance has only one OID-node entry, which helps in tremendous memory saving when compared to the hcC-tree, where each instance has two entries - one in the hierarchy chain, and other in the class chain.

3.2 HC-Tree indexing

A new indexing technique based on the HC-tree has been proposed. The technique is fundamentally based on the class-hierarchy indexing method. Performance studies of various indexing schemes for object databases have shown that an indexing scheme of one index for all classes in a class hierarchy performs better than an indexing scheme that supports one index per class.

We have already discussed that the HC-tree structure effectively supports superclass-subclass relationships. Hence this technique is efficient in satisfying both single-class and class-hierarchy types of queries.

3.3 Operations on the HC-tree index

The primary operations that are performed on an index are search, insertions and deletions.

Search: A HC-tree can be searched under two situations: one is the tree is searched for the instances of a single class and another one is the tree is searched for the instances of a class and also its direct or indirect subclasses.

The values specified may be a single value or a range. The bitmaps available at each node are used to improve the efficiency of the search process. The HC-tree recognizes the absence of any instance satisfying the specified query condition by looking at the bit in the bitmap corresponding to the searched class. The search process terminates whenever the bit is 0, and the depth of the tree is searched only when the value of the bits at the corresponding positions is 1 - indicating the presence of at least one value in the specified range belonging to the queried class (es). This significantly improves the search process. Once the value-node containing the entry for the instances of the class satisfying the query condition is identified, the CID list entry for that particular class is looked up and the OID nodes contained in its OID list are processed, to obtain the results of the query. The search process proceeds by looking at exactly one node at each level of the tree, and therefore, searching HC-trees is as efficient as searching B+-trees - with some additional time taken for processing the OID nodes.

For range queries, the value-node chain is followed until the key value in the value-node falls out of the specified range, and for each value-node, the CID-node list is processed.

***Insertions*:** Inserting a new entry into the HC-tree involves searching the index to locate the value-node where the instance has to be inserted. If a value-node for the value of the indexed attribute of the instance to be inserted already exists, then the CID-node corresponding to the class of the new object is located, and if found, the OID of the object is inserted into the OID-node list of the CID-node. If the CID-node for the class does not exist, then a new CID-node is inserted, the bitmaps of the nodes in the insertion path are duly adjusted to reflect the presence of the new node.

If no value-node is available for the key value, then a new value-node is inserted into the HC-tree. This process is similar to the insertion of the leaf node in a B+-tree, and may result in a split in case of an overflow. As a new value-node is added, an overflowed leaf node is split and the split may propagate up the tree until a node that can accommodate an additional key is found. In the worst case, the split may propagate up to the root node, in which case the height of the tree increases by one. Once a new value-node is created, the insertion process continues as outlined above and the OID of the instance is inserted into the OID-node list corresponding to the CID-node of the class to which it belongs.

***Deletions*:** The deletion process is simpler than the insertion process. As in the case of insertion, the value-node where the instance is available is located, and the OID removed from the OID list. In some cases, the instance may be the only instance of the class, in which case the CID-node has to be removed, and the bitmaps in the internal nodes altered. If the instance is the only OID entry that the value-node contains, the value-node is itself deleted. Deletion of a value-node (which is a leaf node) may cause the internal node to underflow, which affects the balance of the HC-tree. To restore balance, a redistribution of the keys is needed, and the underflowed node is merged with either its left or right sibling. Deletion may even cause the height of the tree to be reduced by 1, if the underflow occurs at the root node.

In some cases, the instance that is to be deleted may not be available in the index. This is recognized during the search process itself, and the deletion process stops if the search is unsuccessful.

IV. PERFORMANCE ANALYSIS

A performance analysis of the new technique has been done along with existing techniques. This section discusses the results obtained.

The CH-tree is very simple and easy to implement. The CH-tree maintains only one index for the entire class hierarchy, which has been found to be more efficient. The CH-tree has excellent update performance, space utilization and is most suited for point queries. Though CH-trees are simple, they fail to satisfy all the requirements of object database indexing. The major limitation of the CH-tree is that it does not support the superclass-subclass relationship naturally. This leads to degradation in performance when applied to complex hierarchies. Searching a class hierarchy is treated the same as searching a single class. In addition to this limitation, performance analysis has shown that the performance of the CH-tree degrades very fast as the queried range increases.

The H-tree provides natural support for superclass-subclass relationship. It is efficient both for single class and class hierarchy queries. Though H-tree is more efficient indexing technique, satisfying all the requirements of object database indexing, its implementation and maintenance is complex, due to the nesting of H-trees used to maintain the class hierarchy. Updates in a H-tree are fairly complex. H-trees are efficient when the number of classes is small, and degradation in performance is observed when the number of classes increases.

The hcC-tree supports both class-hierarchy and single-class queries, by maintaining a class chain and a hierarchy chain. It has been found to be well suited to answer all types of queries efficiently. The major drawback of the hcC-tree is the need to maintain two separate chains. This introduces considerable space and time overheads. Memory is wasted, as the OIDs of the instances represented in the index will have to be stored in two separate chains, thereby resulting in doubling of index space, which is highly undesirable. The memory utilization is very poor, especially when the number of instances is large, which is usually the case with most object databases.

The HC-tree is based on class-hierarchy indexing, which has been proved in the literature to be better than single class indexing. A considerable improvement in memory utilization is obtained by effective maintenance of chains for representing the class hierarchy. Two chains are maintained though the OIDs are stored only once. The technique supports the superclass-subclass relationship naturally, and hence is better than the CH-tree. As the underlying structure is the very efficient B+-tree, this technique too, is very efficient as far as the index operations are concerned. Further, the tree being a self-maintaining tree, the update cost is bounded by the height of the tree only.

The proposed indexing technique supports all the four types of queries of the object databases efficiently, whereas some of the existing techniques have been found to be biased to some types of queries. This technique handles both single-class and class-hierarchy queries in an efficient manner.

While the hierarchy chains are needed to represent the class-hierarchy, this it can be considered to be the drawback of the HC-tree. The presence of the hierarchy chains results in greater complexity of the tree. This also increases the update costs, as the updation process may have to update the chains.

V. CONCLUSION

In this paper, we have presented the design of the new hierarchical structure for indexing object databases called the HC-tree. A method for indexing object databases based on this structure has also been proposed.

The HC-tree indexing technique is based on the class-hierarchy method of indexing. The proposed technique has been found to be effective for both single-class and class-hierarchy queries.

The HC-tree structure extends a natural support to generalization-specialization relationship in the problem domain. Taking all factors into consideration, it can be concluded that the HC-tree indexing proposed in this paper is more efficient than the existing techniques. The HC-tree has been found to be consistently performing well in answering all types of queries on object databases. It also makes optimal usage of memory, thus reducing the burden on available resources, while providing an efficient method of indexing object databases.

REFERENCES

1. [BF95] Bertino E., Foscoli P. Index Organizations for Object-Oriented Database Systems. *IEEE Transaction Knowledge and Data Engineering, Vol*. 7, No 2, 1995.
2. [C79] Comer D. E. The Ubiquitous B-tree, *ACM Computing Surveys*, Vol.11, No 2, 1979.
3. [KKD89] Kim W., Kim K. C and Dale A. Indexing Techniques for Object-Oriented Databases. *Object-oriented Concepts, Databases and Applications.* Addison-Wesley, 1989.
4. [K90] Kim W. Object-Oriented Databases: Definition and Research Directions. *IEEE Transaction Knowledge and Data Engineering, Vol*. 2, No 3, 1990.
5. [KKS92] Kim W., Kifer M. and Sagiv Y. Querying Object-Oriented Databases. In *Proceedings of ACM SIGMOD Conference on Management of Data (SIGMOD'92), 1992.*
6. [LOL92] Low C. C., Ooi B. C. and Lu H. H-Trees: A Dynamic Associative search index for OODB. In Proceedings *of ACM SIGMOD Conference on Management of Data, 1992.*
7. [RK95] Ramaswamy S., Kanellakis P.C., OODB Indexing by Class-Division. In *Proceedings of ACM SIGMOD Conference on Management of Data (SIGMOD'95), 1995.*
8. [SS94] Seshadri S., Sreenath B., The hcC-tree: An Efficient Index Structure for Object Oriented Databases. In *Proceedings of the 26th International Conference on Very Large Databases, 2000..*