International Journal of Computer Applications (0975 – 8887)
Volume 17– No.4, March 2011

Dynamic Interpolation B-Tree: A New Access Method

Dr. P. Dinadayalan
Department of Computer Science
Kanchi Mamunivar centre for P.G. Studies
Pondicherry, India

Dr. Gnanambigai Dinadayalan
Department of Computer Science
Indira Gandhi College of Arts and Science
Pondicherry, India

ABSTRACT

The performance of Object-Oriented Database depends on the access method implemented in the data model. Dynamic Interpolation B-tree (DIB) is a new indexing technique supporting query processing in Object-Oriented Databases which is effective and efficient for multimedia databases. This is a new access method which supports range queries on Object-Oriented Databases. DIB supports inheritance and aggregation hierarchies. DIB has the structure of Dynamic Interpolation B-tree. Dynamic Interpolation B-tree consists of hashing and B-tree. Both hashing and B-tree are dynamic. DIB technique is compared with other techniques obtained from more traditional organizations. In this new technique all the operations are done efficiently. The result shows that the Dynamic Interpolation B-tree is significantly better than the traditional indexing methods over a wide range of parameters in terms of range of parameters, retrieval and update cost so that the storage overhead grows slowly with the number of indexed attributes.

Keywords

B-tree, DIB, interpolation, hashing, OODBS, databases and index.

1. INTRODUCTION

Database Management Systems evolved from file systems, which support storage of large amount of data. Researchers in database field, however, found the data has its value, and models based only on data should be introduced to improve the reliability, security, efficiency of the access. Data models [1][3][12] provide a way in which the stored data is organized as specified structure or relation for quick access and efficient management. Many models, such as Hierarchical model, Network model, Entity- Relationship model, Functional model, Relational model, Object-Oriented model [12], has come into existence and played important roles since the emergence of the database management systems. Each of these models modeled the data and the relationship between the data in different ways. Each of the models encountered some limitations in being able to represent the data which resulted in other models to compensate for the limitations.

Object-oriented database management system [1] [3] has been developed in recent years. Object-oriented data model, which is supported by the system, has three basic characteristics. The first is the possibility of directly modeling complex, nested object. The second is to organize classes (types) into inheritance hierarchies. The third is to support high-level declarative query languages. An important issue related to query languages concerns optimization techniques and access structures able to reduce query-processing costs. In this paper, we are going to discuss and summarize some indexing techniques for object-oriented data management system and proposed approaches. Efficient execution of queries is achieved by the allocation of suitable access structure and the use if sophisticated query optimizers. Access structures typically used in relational DBMSs are based on variations of the B-tree structure or hashing techniques[1][2][3]. An index is maintained on an attribute or combination of attributes of a relation. Since an object-oriented data model has many differences from the relational model, suitable indexing techniques must be developed to efficiently support object-oriented query language. An object-oriented database can be organized along two dimensions: aggregation, and inheritance [4]. We will discuss indexing techniques for those two dimensions respectively and also present integrated organizations supporting both two dimensions. The rest of the paper is organized as follows. In Section 2 summarizes the access methods proposed in the literature. Section 3 introduces the concept of Dynamic Interpolation B-tree and possible approaches to implement the DIB. In Section 4, we present the comparison and storage overheads. Finally, we conclude the paper in Section 5.

2. RELATED WORK

OODBS offers completely a different kind of access pattern than conventional databases [1][2][3][4]. This difference in access pattern is mainly due to additional semantics offered by the data model of object-oriented databases. These semantic help of the index designer take into account the following factors into consideration:

- The object can be recursively composed of other objects, this is called aggregation hierarchy [4].
- An object may be derived from another other object thus forming an inheritance hierarchy [4].
- An object may be related to other objects by both inheritance and aggregation hierarchies [4].

Fig. 1 is an example of aggregation and inheritance hierarchies that are dealt in the following a Customers database schema as shown in figure 1 is used as an example. An Order consists of four attributes. First and last attributes are of primitive types and other two attributes (Customer and Item) are of composite types. Class Customer consists of three attributes namely Customercode, Name and address. Attributes Customercode, Name and Address are of primitive data types. Class Item consists three attributes namely Itemcode (primitive), Itemname (primitive) and Amount (primitive). Class Order is inherited by two more classes such as Oldcustomer and Newcustomer.

2.1 Indexing on Aggregation Hierarchy

A class groups all objects with similar attributes and behavior[4]. If specifies a set of attributes that define object structure and a set of methods that define object behavior. An attribute defines a name and the corresponding domain. The domain associated with the attribute can either belong to a primitive type like integer or non-primitive type like image. The fact that a class is a domain of attribute of another class

establishes an association called aggregation relationship between those classes. A number of indexing has been suggested to answer queries efficiently along aggregation hierarchies and they are Multi Index, Join Index, Nested Index, Path Index, Access Relations and Direct Links.

Multi-index [2][4] is the first of the indexing technique for OODBS. The first proposed organization for indexing aggregation graphs is based on allocation a B^+-tree indexes. The B^+-tree index is on each class traversed by the path. For a multi-index organization, solving a nested predicate requires a scanning of a number of indexes equal to the length of the path. Under this organization the retrieval operation is performed by:

- Using the results of this index lookup as keys for a search on the index preceding the last one in the path so on until the first index is scanned.
- Onward until the first index is scanned. Only reverse traversal strategies can be supported using this organization. The major advantage of multi-index is the low update cost.

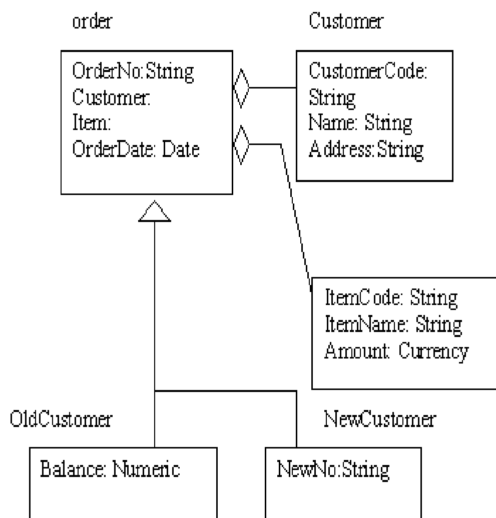

Figure 1. A Customer database for aggregation and inheritance hierarchies

Join-index [2][4] was introduced to perform joins in relational model efficiently and also was used to efficiently implement complex object. A binary join-index is implemented as binary relation and may be kept. An index, binary join index, is implemented as binary relation and kept two copies. Each repeat is implemented as a B+-tree. A Binary Join Index sequence can be used in a multi-index organization to implement the various index components along a given path for aggregation graphs. Both forward and reverse traversal strategies can be supported by a Join Index organization. As the reverse traversal is suitable for solving queries and for the forward traversal happens when it is necessary to identify all objects. So all objects need to be determined that being reference directly or indirectly by a given object. The reverse traversal is already hold by the multi-index. Although, no such technique does not support forward traversal, so executed by directly accessing the objects. The advantages of using a sequence of Join Index may be useful in complex queries. Also the usage of a sequence of Join Indices may make forward traversal faster when objects accesses are expensive.

Nested index [2][4] provides a direct association between an object of a class at the end of the path and the corresponding instance of the class at the beginning of the path. The retrieval using this organization is quite efficient. However the major problem of this indexing technique is the update operations which require access to several objects in order to determine the index entries to be updated The update operations require both forward and backward traversal of objects.

Path Index [2][4] maps a specific nested attribute to the classes located along the given path. It is similar to nested index with a single index being maintained on a path. Path index can be used to solve nested predicates against all classes along the path.

Access relation [2] is a generation of the join indices for OODBS. Instead of supporting traversal (or join) of two connected classes (relations), access support relations support the traversal along the path arbitrary length. The relations may be created by joining all of the classes on the path. Similar to join indices, two copies of an access support relation are stored and clustered correspondingly on the OIDs of objects in the two end classes of the path.

Direct links [4] maintain links connecting objects in two separate classes or fast object traversal. The direct links are similar to projecting the OIDs of the end classes on the access support relations. Thus, it may go from one end of the path to the other end effectively.

2.2 Indexing on Inheritance Hierarchies

The inheritance hierarchy indexing is addressed in different kind of approaches. The various approaches are analyzed and considered about storage, update and retrieval costs. The storage requirements of a class-hierarchy index is the total number of index pages necessary to maintain the UIDs of all instances of all classes on the class hierarchy rooted at the indexed class. Retrieval costs depend on whether the query is a point query or a range query.

The first group of approach is the SC-index, the H-tree and the CG-tree group. The SC-index, H-tree and the CG-tree group attribute values in the leaf nodes of B^+-tree on the base of a class where instance with the value appears. The second group is the CH-tree and the hcC-tree group. In the following section will discuss about these Indexing techniques.

Single-class index (SC-index) [4] is based on maintaining a separate B+-tree on the indexed attribute for each class in the inheritance hierarchy.[Maier et al] This approach is very efficient for SC-queries. However, it is not optimal for CH-queries, because it requires scanning all the indexes allocated on the classes in the queried inheritance hierarchy.

Class-hierarchy index (CH-tree) [4] is based on maintaining a unique B+-tree for all classes in the hierarchy. An index entry in a leaf node may thus contain the OIDs of instances of any class in the indexed inheritance hierarchy. However, the CH-tree retrieves many unnecessary leaf node pages, when the queries apply to a single class only. The performance of the CH-tree has inverse trend with respect to the SC-index. The CH-tree is more efficient for queries whose access scope involve all classes in the indexed inheritance hierarchy,

whereas a SC-index is efficient for queries against a single class.

H-tree [4] is a variant of the SC-index. H-tree is similar to the SC-index in the way that B+-tree is maintained on the indexed attribute for each class in the inheritance hierarchy. In the H-tree, however, the B+-tree are linked based on their class-subclass relationships by pointers in the internal nodes of the B+tree. H+tree aims at improving the performance of the SC-index for CH-queries.

CG-tree [4] enhance the H-tree by collecting all pointers between different class's indexes in special nodes which create one additional level located just before the leaf node level of B+tree. CG-tree avoids reading unnecessary internal nodes so it provides more efficient than H-tree. However, the CG-tree has a high storage overhead and update cost due to the class directories.

X-tree is a dynamic indexing technique similar to the R-tree and R*-tree. Data are stored in the leaf nodes, which appear at the same level of the tree. Each leaf node entry consists of the key value K, the object identifier OID and the identifier CID of the class the object belongs to. If all entries with the same key value K do not fit one leaf node, two or more nodes are allocated and all node entries with same class identifier are grouped together. The basic idea is to keep the structure as hierarchical as possible, and at the same time to avoid splits in the nodes that would result in high overlap.

3. DYNAMIC INTERPOLATION B-TREE (DIB)

Indexing is a software technique used to retrieve the data from the persistence storage. Index is usually maintained on an attribute or its combination. Hashing and B-tree is the most widely used technique for indexing relational databases and earlier generation of databases. The current generation of Database Systems, OODBS, also use either Hashing and B-trees for indexing purpose. Hashing and B-trees have been used by adding rich semantic features of OO Data model. A survey made [1][2][3][4] show that the all OODBS use either SC and CH trees for indexing purpose. SC and CH techniques are not suitable to answer all kinds of point and range queries. Hashing and B-tree variants are the conventional indexing techniques. Hashing exhibits static nature of data and uses digital properties of the keys while B-tree exhibits dynamic nature and uses relational properties of the keys to map the keys to the universe of key values.

In this paper, we propose a new technique, called Dynamic Interpolation B-tree, to support efficient query evaluation in object-oriented databases. By using this DIB technique the physical structure of the database cannot be changed, since the access structure is built on top of the actual database. It can build on top of the actual database. Fig.2 shows the overall architecture of the DIB. DIB combines the working of hashing and B-tree. This index organization has a hash table. Each entry in the hash table maintains a partial range of keys, a pointer to B-tree which index all data in its range and its height. In this index organization, both the hash and B-tree are dynamic. Dynamic hashing is a combination of hashing techniques with trie structure. A trie is a tree in which branching is determined not by the entire key value but by only a portion of it. In addition, branching is based on consideration of that key alone, not on the comparison of a search key with a key stored inside the node. The key can be from any OIDs or attribute set. Any other key representation can be easily converted to binary. The keys are binary numbers set with n bits.

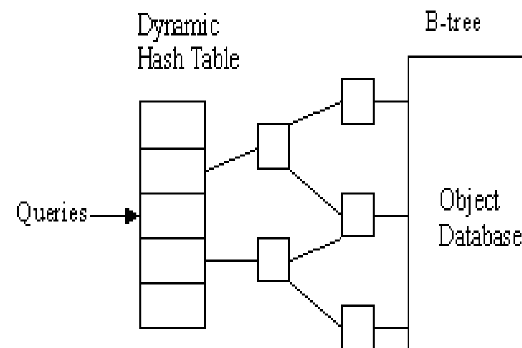

Figure 2. Structure of Dynamic Interpolation B-tree

The hashing technique allows the hash function to be modified dynamically to accommodate the growth or shrinking of the database. The dynamic hashing that grows to handle more items. The associated hash function must change as the table grows. Sometimes dynamic path directory shrinks the table to save space when items are deleted. The dynamic path directory in which the hash function is the last few bits of the key and the table refers to buckets. Table entries with the same final bits may use the same bucket. If a bucket overflows, it splits, and if only one entry referred to it, the table doubles in size. If a bucket is emptied by deletion, entries using it are changed to refer to an adjoining bucket, and the table may be halved.

B-tree on the other hand exhibits dynamic nature and uses relational properties of the keys to map the keys to the universe of key values. The B-tree is connected with the hash table using a pointer. A B-tree is a tree data structure that keeps data sorted and allows insertions and deletions in logarithmic amortized time. It is most commonly used in databases and file systems.

An optimization of a tree which aims to keep equal numbers of items on each sub-tree of each node so as to minimize the maximum path from the root to any leaf node. As items are inserted and deleted, the tree is restructured to keep the nodes balanced and the search paths uniform. Such an algorithm is appropriate where the overheads of the reorganization on update are outweighed by the benefits of faster search.

In B-trees, internal nodes can have a variable number of child nodes within some pre-defined range. When data is inserted or removed from a node, its number of child nodes changes. In order to maintain the pre-defined range, internal nodes may be joined or split. Because a range of child nodes is permitted, B-trees do not need re-balancing as frequently as other self-balancing search trees, but may waste some space, since nodes are not entirely full. The lower and upper bounds on the number of child nodes are typically fixed for a particular implementation.

Dynamic Interpolation B-tree Path Index will be effective and efficient for multimedia databases. Both hashing and B-tree are dynamic in DIB. Each entry in the hash table maintains a

partial range of keys, a pointer to B-tree which index all data in the hash table and its height. It also maintains a header which keeps track of minimum and maximum key value in the whole key range followed by a count of entries in the hash table.

The hash table search uses the interpolation formula. An entry I in the hash table where the key K may be found is computed by formula:

$$I = ((K - K_{low}) / (K_{high} - K_{low})) * n \qquad (1)$$

where K_{low} and K_{high} are the least and highest key value in the search space and n is the number of entries in it.

4. COMPARISONS AND STORAGE OVERHEADS

To evaluate the performances of the three index organizations (SC-tree, CH-tree, and DIB), then executed a large number of simulation experiments on the basis of mathematical model. Various experiments were conducted on DIB. The performance of DIB is compared with that of SC-trees and CH-trees. Initially, through a series of preliminary experiments, it has determined the parameters that characterize the topology of the database are those that can affect in varying degrees the size and performance of an index. From the result of these experiments it was found that DIB organization offers better performance than the traditional indices in nearly all classes. In this experiments 25,000 random objects are used in building the DIB. The DIB organization is the combination of dynamic hashing and B-tree. The searching time of Dynamic Interpolation B-tree Path Index (DIB) is the sum of the searching time Dynamic Hash table and B-tree.

4.1 Storage Cost

In all experiments performed, it has obtained that the traditional indices have the lowest storage cost. In particular Dynamic Interpolation B-tree has the best costs when in the path there are inheritance hierarchies and aggregation hierarchies while the other index organization has the highest cost. Therefore, it may be preferable to privilege organizations providing good performance, even if they have less storage requirements.

Total_DIB_SearchTime= Hash_SearchTime + B-Tree_Search time .... (2)

4.2 Retrieval Costs

From the simulation experiments that have been performed, the Dynamic Interpolation B-tree offers in general good performance unlike the traditional organizations. The DIB organization offers better performance than traditional indices in nearly all cases. The SC-tree and CH-tree organizations are advantageous only when most retrieval operations have as target the last classes of the path. Therefore, the interesting cases are those when the target of the query is one of the classes at the beginning of the path. If the database has small dimension, the traditional indices have costs that does not differ much from the DIB organization. Also when all the classes but the first class of the path are low, the costs of the SC-tree and CH-tree are acceptable. Not how the position of the target class of the query is important. In fact, if the target class is the last of the path, the costs for the traditional organizations are low while, if the position of the such class is one, the costs of the SC-tree and CH-tree grow exponentially for varying values and for different positions of the class. The costs of the DIB organization do not depend on the presence of inheritance and aggregation hierarchies, while this factor influences the performance of the SC-tree organization. Its costs, indeed, depend on the number of classes in the path, unlike those of the CH-tree organization that only depend on the length of the path. Another factor influencing the costs SC-tree and CH-tree organizations is the length of the path mainly when the target classes are positioned in the beginning of the path. Also the cost of the DIB organization grows for increasing dimensions of the range but to a lower degree than the SC-tree and CH-tree organizations.

4.3 Delete Costs

A delete operation, unlike a retrieval operation, is generally more expensive in a DIB organization than in the traditional organizations. While the deletion of an instance from a class is expensive for the SC-tree and CH-tree organizations only if such class is characterized, this operation is expensive for the DIB organization independently on the class from which the instance is removed. There are some cases, however in which the costs for the DIB organization are only slightly higher or even lower than those of traditional organizations. An optimal case for the DIB is when all indexed attributes are single-valued. The trend of the costs of the DIB and of the SC-tree and CH-tree just described is independent from the presence in the path of inheritance and aggregation hierarchies. In conclusion the DIB organization offers good performance when all classes in the path have single-valued indexed attributes or when only the first class of the path has a multi-valued attribute in the path and all other classes have single-valued indexed in the path, or when the delete operation has as target the class of the path that has a high value.

4.4 Insert Costs

The insert costs for the three organizations are lower than those of delete operations, since the number of updates to the index structure is smaller. The main difference between delete costs and insert costs is for the DIB organization. There is only one difference: while a delete operation has costs for the DIB organization that are independent from the class from which an object is inserted and immediately following classes in the path have low value. In particular, the performance of the DIB organization for the insert operation is as good as the other organizations when the classes at the beginning of the path have low value.

5. CONCLUSION

Dynamic Interpolation B-tree is an index organization for Object-Oriented Databases which supports range queries indexing several classes along aggregation and inheritance hierarchies. DIB indexing technique is compared with the other indexing techniques to overcome the limitation of the later. Dynamic Interpolation B-tree organization offers the best retrieval performance in most cases. The costs of various organization for modification operations (delete, insert) are highly depended on the object references topologies. It is generally required to store enormous data and retrieve data from the database in a shorter duration. The operations such as insertions, deletions and updation are found to be working properly. Due to the dynamic nature of objects, Dynamic Interpolation B-tree structure is best suited for Object-

Oriented Databases. It is shown that this technique performs better than most widely used indexing techniques.

6. REFERENCES

[1] Awais Rashid, Peter Sawyer, "A database evolution taxonomy for object-oriented databases", Journal of Software Maintenance and Evolution: Research and Practice, Volume 17, I ssue 2, p.p.93–141, March/April 2005.

[2] A.V. Aho, J.E. Hopcroft, J.D. Ullman, "The Data Structures & Algorithms," Addition –Wesley,1998.

[3] E. Bertino, "An Indexing Technique for Object-Oriented Databases," Proc. Seventh Int'l Conf. Data Eng., pp. 160–170, Kobe, Japan, 1991.

[4] E. Bertino and W. Kim, "Indexing Techniques for Queries on Nested Objects," IEEE Trans. Knowledge and Data Eng., vol. 1, no. 2, pp. 196–214, June 1989.

[5] R.J.Enbody, "Dynamic Hashing Schemes", ACM Computing Surveys, Vol.20, No.2, April 1998.

[6] A. Kemper and G. Moerkotte, "Advanced Query Processing in Object Bases Using Access Support Relations," Proc. 16th Int'l Conf. Very Large Data Bases, pp. 290–301, Brisbane, Australia, Aug. 1990.

[7] A. Kemper and G. Moerkotte, "Access Support in Object Bases," Proc. 1990 SIGMOD Conf., pp. 364–374, Atlantic City, N.J., May 1990.

[8] W. Kim, "A Model of Queries for Object-Oriented Databases," Proc. IEEE Int'l Conf. Very Large Data Bases, pp. 423–432, Amsterdam, 1989.

[9] W. Kim, K.C Kim, and A. Dale, "Indexing Techniques for Object-Oriented Databases," W. Kim and F.H. Lochovsky, eds., Object-Oriented Concepts, Databases, and Applications, pp. 371–394, Reading,Mass., Addison-Wesley, 1989.

[10] W.C. Lee and D.L. Lee, "Combining Indexing Technique with Path Dictionary for Nested Object Queries," Proc. DASFAA '95,Fourth Int'l Conf. Database Systems for Advanced Applications, pp.107–114, Singapore, Apr. 1995.

[11] D.L. Lee and W.C. Lee, "Using Path Information for Query Processing in Object-Oriented Database Systems," Proc. Conf. Information and Knowledge Management, pp. 64–71, Gathersberg, Md., Nov.1994.

[12] W.C. Lee and D.L. Lee, "Path Dictionary: A New Approach to Query Processing in Object-Oriented Database", IEEE Transaction on Knowledge and Data Engineering (TKDE), Volume 10, No. 3, May/June 1998, pp. 371-388.

[13] Paul Rodrigues, S.Kuppuswami, "Concurrency of Operations on IB-trees", Pondicherry University, Pondicherry, 2000.

[14] Paul Rodrigues, "Performance Amelioration of Object-Oriented Databases ", Ph.D report, 1999, Pondicherry University, Pondicherry, India.

[15] Pichayotai Mahatthanapiwat, Wanchai Rivepiboon ,"Virtual path signature: An approach for flexible searching in object-oriented databases", International Journal of Intelligent Systems, Volume 19, Issue 1-2, p.p. 51–63, January/ February 2004.