Design and Implementation of a Main Memory Index Structure in a DBMS

Duck-Ho Bae
College of Information and Communications
Hanyang University, Seoul, Korea

Jong-Dae Kim
College of Information and Communications
Hanyang University, Seoul, Korea

Se-Mi Park
College of Information and Communications
Hanyang University, Seoul, Korea

Sang-Wook Kim*
College of Information and Communications
Hanyang University, Seoul, Korea

ABSTRACT

The main memory DBMS (MMDBMS) efficiently supports various database applications that require high performance since it employs main memory rather than disk as a primary storage. An index manager is an essential sub-component of a DBMS used to speed up the retrieval of objects from a large volume of a database in response to a certain search condition. Previous research efforts on indexing proposed various index structures. However, they hardly dealt with the practical issues occurred in implementing an index manager on a target DBMS. In this paper, we touch these issues and present our experiences in developing the index manager. The main issues are (1) compact representation of an index entry, (2) support of variable-length keys, (3) support of multiple-attribute keys, and (4) support of duplicated keys.

Keywords: *DBMS, Main Memory DBMS, Indexing*

1. INTRODUCTION

The Main-Memory Data Base Management System (MMDBMS) uses main memory as a primary storage for eliminating the cost of disk accesses, which have been known as the main performance bottleneck of the disk-based DBMS [1][3][5].

An index manager in a DBMS supports the fast retrieval of target objects that satisfy a query condition from a database. To facilitate this functionality, the index manager chooses one or more attributes as a key and builds an index from them. There have been many research efforts to devise efficient index structures for database systems. The binary search tree, AVL-tree, T-tree [8], B-tree [2], chained-bucket hashing, extensible hashing [4], and linear hashing [10] are the typical examples. Cache-conscious index structures such as the CSS-tree [13] and the CSB+ [14] have also been proposed. They could optimize search performance in some degree at the expense of update performance by considering the cache behavior.

Previous research efforts mainly focused on designing efficient index structures appropriate for their own application domains. However, they rarely dealt with the practical issues occurred in implementing an index manager on a target DBMS.

In this paper, we investigate design and implementation issues experienced in developing an index manager. The main issues discussed in this paper are: (1) compact representation of

* *Corresponding author. E-mail : wook@hanyang.ac.kr*
Manuscript received Sep. 11, 2007 ; accepted Sep. 27, 2007

This research was supported by the MIC(Ministry of Information and Communication), Korea, under the ITRC support program of supervised by the IITA(IITA-2005-C1090-0502-0009) via Cheju National University.

an index entry, (2) support of variable-length keys, (3) support of multiple-attribute keys, and (4) support of duplicated keys.

The paper is organized as follows. Section 2 introduces previous index structures and addresses their strong and weak points. Section 3 presents the issues and their solutions experienced in developing the index manager in detail. Finally, Section 4 summarizes and concludes the paper.

2. INDEX STRUCTURES

This section reviews previous index structures proposed for database systems.

2.1 Tree-based indexes

Since tree-based indexes traverse down a tree to locate an object, their performance for exact-match queries is worse than that of hashing-based indexes. However, tree-based indexes show better performance for range queries because index entries having adjacent key values can be easily accessed with a sorted order.

The binary search tree has a simple structure. Therefore, its algorithms for search, insertion, and deletion are also simple. Since the binary search tree is not balanced, its search performance heavily depends on the distribution of key values and the insertion/deletion orders of objects.

The AVL-tree is a balanced binary tree in that the difference in the heights of the two subtrees of any node is at most one. It maintains the simple structure of the binary search tree and accomplishes the balancing property using the rotation operation. The biggest problem of the AVL-tree, however, is its storage overhead. Each node stores the following three information: (1) a key entry <key value, object pointer>, (2) two pointers to its left and right subtrees, and (3) related control information. Therefore, the total amount of information to be kept for just one entry is too large.

The T-tree [8] solves the storage overhead of the AVL-tree. While the T-tree uses the structure and the balancing scheme that are identical to those of the AVL-tree, it stores multiple entries in a node. As a result, storage overhead for each key entry gets smaller.

The B-tree [2] is a completely balanced index structure widely used in disk-based DBMSs. It maximizes the fan-out of a node to reduce the height of a tree, and thus minimizes the number of disk accesses in tree traverse. In most cases where the fan-out of a node is at least 2, the B-tree has storage overhead larger than the T-tree.

2.2 Hashing-based indexes

Hashing-based indexes compute the position of an object directly from its key value. Therefore, they have better performance in processing exact-match queries compared with tree-based indexes. However, they show worse performance in processing range queries because the index entries having adjacent key values tend to be scattered in hashing-based indexes.

The chained-bucket hashing uses a fixed-size hash table, and thus shows good search performance only when it has a hash table that is optimal to a given database. When the hash table is too small, overflow buckets degrade search performance. On the contrary, a hash table wastes storage space when it is too large. In dynamic environment where insertions and deletions of objects frequently occur, however, it is not easy to estimate an optimal size for a hash table.

The extensible hashing [4] consists of data pages and a directory. Data pages store data objects and a directory stores pointers to data pages. The directory has 2k (k=0,1,...) pointers. The directory adapts to dynamic environment by splitting and merging. When the number of pointers stored in a directory exceeds its capacity, the directory doubles. Therefore, the extensible hashing seriously wastes storage space for its directory when most objects are concentrated in a few data pages.

The linear hashing [10] also has a dynamic structure. It maintains data pages in physically contiguous space, thus performs page addressing by calculation rather than directory searching. The linear hashing permits overflow chains, which may cause search performance to degrade. Whenever necessary, it splits data pages in the pre-defined order, resulting in increased space utilization.

Lehman et. al [8] modifies the linear hashing to be suitable for MMDBMSs. The directory is re-introduced for locating data pages for making it unnecessary to store data pages in physically contiguous pages. Since the modified linear hashing does not allocate empty pages, it has space utilization much better than the extensible hashing.

2.3 Our Choice

We selected an index structure based on the three criteria: (1) search performance for both exact-match and range queries, (2) storage overhead, and (3) adaptability to dynamic environment.

For range queries, we chose the T-tree for its balancing property and small storage overhead. The balancing property enables us to guarantee good search performance regardless of the key distribution and the insertion/deletion orders of objects. Since the T-tree stores multiple entries in a node, its storage overhead is small. Dynamic allocations and deallocations of nodes make the T-tree adapt to dynamic situations. The T-tree performs well for range queries, and is also capable of processing exact-match queries via tree traversal.

At first, we intended to employ an additional hashing-based index structure for exact-match queries. The chained-bucket hashing was left out because of its static nature. Since dynamic hashing indexes find the location of the target data page by computation, they have to maintain directory pages (in the extensible hashing and the modified linear hashing) or data pages (in the linear hashing) in physically contiguous space. However, it is not trivial to determine the optimal size of contiguous main memory.

The easiest way is to allocate the largest size of contiguous space at a database setup stage. If the size of contiguous space is too small compared with the database size, however, performance degrades seriously due to overflow chains. On the contrary, main memory space wastes when the size of contiguous space is too large. The other way is to allocate larger contiguous space whenever needed after releasing current contiguous space. However, it could not be always possible to obtain larger contiguous space due to memory fragmentation in dynamic environment. When larger space is not available, overflow chains would be unavoidable.

Traversing of overflow chains is almost same as traversing of a tree. Furthermore, the length of overflow chains increases in O(n), where n is the number of objects while the depth of the T-tree grows in O(log n). Therefore, even for exact-match queries, the hashing-based indexes could perform worse than the T-tree under dynamic environment. In addition, the length of overflow chains is highly dependent on the size and distribution of a database. Therefore, the hashing-based indexes cannot guarantee the worst-case search performance.

For these reasons, we chose the T-tree as an index structure for both exact-match and range queries. By employing a single index structure for both exact-match and range queries, we have enjoyed an additional advantage of making other sub-components such as the concurrency, backup, and recovery managers much simpler.

3. IMPLEMENTATION ISSUES OF INDEX MANAGER

In this section, we review the T-tree structure and present several implementation issues and our solutions experienced in developing the index manager.

3.1 T-tree

The T-tree [8] enforces the following balancing property: the height of the two subtrees of any node differs by at most one. When the balance is broken by deletion or insertion, the T-tree re-balances itself using the rotation operation. There are three kinds of nodes in the T-tree: the internal, half-leaf, and leaf nodes. Internal nodes have two subtrees. Half-leaf nodes have a single subtree. Leaf nodes do not have a subtree. The number of entries in an internal node and a half-leaf node should be between the pre-defined minimum and maximum. The leaf node also has a constraint on its maximum number of entries, however has no constraint on its minimum.

minKey(N) and maxKey(N) denote the smallest and the largest key values, respectively, stored in node N. Given node N and key value k, we say that N bounds k when k is between minKey(N) and maxKey(N). GLB(k) and LUB(k) represent the greatest lower bound and the least upper bound of key value k, respectively. That is, GLB(k) is the key value just before k and LUB(k) is the key value right after k in a T-tree. For any internal node N in a T-tree, there exists: (1) a leaf node or a half-leaf node that contains GLB(minKey(N)), and (2) a leaf node or a half-leaf node that contains LUB(maxKey(N)).

The search algorithm of the T-tree is similar to that of the binary search tree. Two comparisons are required at each node N of the T-tree: one for comparing a query value with minKey(N) and one for comparing a query value with maxKey(N). The search algorithm consists of two steps: The first step traverses the tree to locate the node that bounds a query value. The next step performs binary search within that node in order to find the entry whose key value equals to a query value.

To insert a new key value, we find the node N that bounds the value and insert it into N. If N overflows by this insertion, minKey(N) is shipped to a leaf node or a half node that has GLB(minKey(N)). When there is no node that bounds a new key value, we insert it into a leaf node or a half-leaf node at which the traversal ends. When the node overflows by this insertion, we create a new leaf node under the overflowed node. If the T-tree becomes unbalanced by a newly created leaf node, we perform the rotation operation to make the tree balanced again.

To delete a key value, we find the node N that bounds the key value and delete the corresponding entry. If N underflows, we move LUB(maxKey(N)) into N. If this makes a leaf node empty[1], we delete that node from the T-tree. We perform the rotation operation if the T-tree gets unbalanced by this node-deletion.

3.2 Support of variable-length keys and multiple-attribute keys

A key is called a variable-length key when at least one of its organizing attributes has a variable length. Variable-length keys make various algorithms used in DBMSs complicated, especially concurrency control and recovery algorithms [6][12].

For efficient support of variable-length keys, we store only an object address in an index entry. Using the address of an object, we can easily access its key value from the object in main memory. Thus, we do not need to maintain key values in the T-tree. Our approach has the following advantages: (1) By fixing the length of index entries, many DBMS algorithms become quite simple; (2) the storage overhead of the T-tree gets smaller.

For key comparisons, however, we need to know which attributes in an object comprise a key. For this purpose, the system catalog [6] maintains the following information on each T-tree in TtreeInfo: UorD, root, numAttributes, <attrDesc[0] attrDesc[1], ..., attrDesc[MAX-1]>.

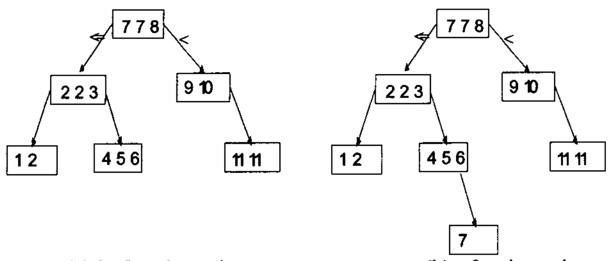
UorD indicates whether a duplicate key is allowed or not. root stores the address of the root node of the T-tree and numAttributes the number of organizing attributes. TtreeInfo also stores the information on each organizing attribute in attrDesc[i]. As a result, the multiple-attribute key, which is defined as a key that consists of more than one attribute, are easily supported. Each organizing attribute is described by the three fields <offset, size, dataType>. offset is the starting position of an attribute within an object. size and dataType are its maximum size and data type, respectively.

3.3 Support of duplicate keys

When a system allows duplicate keys, different objects may have the same key value. The simple way [6] is to replace the structure of index entries with <key value, list of object addresses>. This makes the index entries be of variable length. Furthermore, overflow nodes have to be introduced since the size of an index entry possibly gets larger than that of a leaf

[1] The T-tree permits underflowed leaf nodes.

page. All of these make the algorithms in other DBMS sub-components complicated.



(a) before insertion. (b) after insertion.

Fig. 1. Inserting the duplicate key 7.

As a solution to this problem, we take the approach to allocate an independent index entry to each object regardless of its key value. That is, we allocate m index entries in the T-tree for m objects even when they have the same key value. Our approach does not change the structure of index entries and also does not require additional mechanisms to handle special cases. This makes other algorithms kept simple. In Figure 1, we observe that the only change is to replace the sign "<" in left child pointer with "≤".

Figure 1 shows the two stages of the T-tree; the left one before inserting a duplicate key value 7 and the right one after inserting that key value. We assume that the blocking factor is 3. For easier explanation, we express each node entry with its key value instead of its object address. In order to insert 7, we first search for the node that bounds 7, which is the root node having no more space. In this case, we go down the tree with the key value and insert it into a newly created leaf node. As shown in Figure 1, multiple index entries with the same key value can be dispersed in more than one node. Therefore, even though we find the first node that bounds the value given at query time, we have to check its GLB value in order that we do not miss other nodes that bound the value.

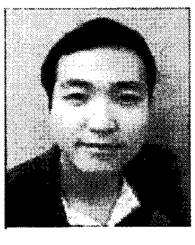
4. CONCLUSIONS

MMDBMSs provide a promising solution to improve DBMS performance by replacing disk with main memory. The index manager is an essential DBMS sub-component that supports the fast retrieval of target objects. There have been a lot of research efforts to devise efficient index structures. However, they hardly address the issues on their implementation and seamless integration with DBMSs.

This paper investigated practical issues experienced in developing an index manager, and proposed our approaches to them. The main issues discussed in this paper are:(1) compact representation of index entries, (2) support of variable-length keys, (3) support of multiple-attribute keys, and (4) support of duplicate keys. We believe that our contribution would help MMDBMS developers highly reduce their trial-and-errors even when they employ different index structure other than the T-tree for their target DBMS.

REFERENCES

[1] A. Ammann, M. Hanrahan, and R. Krishnamurthy, "Design of a Memory Resident DBMS", Proc. Intl. Conf. on COMPCON, Feb. 1985.
[2] D. Comer, "The ubiquitous B-Trees", ACM Computing Surveys, Vol. 11, No. 2, 1979, pp. 121-137.
[3] D. DeWitt et al., "Implementation Techniques for Main Memory Database Systems", Proc. Intl. Conf. on Management of Data, ACM SIGMOD, 1984, pp. 1-8.
[4] R. Fagin et al., "Extensible Hashing: A Fast Access Method for Dynamic Files", ACM Trans. on Database Systems, Vol. 4, No. 3, 1979, pp. 315-344.
[5] H. Garcia-Molina and K. Salem, "Main Memory Database Systems: An Overview", IEEE Trans. on Knowledge and Data Engineering, Vol. 4, No. 6, 1992, pp. 509-516.
[6] J. Gray and A. Reuter, Transaction Processing: Concepts and Techniques, Morgan Kaufman Publishers, 1993.
[7] S. I. Jun et al., "SROS: A Dynamically-Scalable Distributed Real-time Operating System for ATM Switching Network", Proc. IEEE Global Telecommunications Conference, 1998, pp. 2918-2923.
[8] T. Lehman and M. Carey, "A Study of Index Structures for Main Memory Database Management System", Proc. Intl. Conf. on Very Large Data Bases, VLDB, Aug. 1986, pp. 294-303.
[9] T. Lehman and M. Carey, "A Recovery Algorithm for a High-Performance Memory-Resident Database System", Proc. Intl. Conf. on Management of Data, ACM SIGMOD, 1987, pp. 104-117.
[10] W. Litwin, "Linear Hashing: A New Tool For File and Table Addressing", Proc. Intl. Conf. on Very Large Data Bases, VLDB, 1980, pp. 212-223.
[11] C. Mohan and F. Levine, "ARIES/IM: An Efficient and High Concurrency Index Management Method Using Write-Ahead Logging", IBM Research Report RJ 6846, 1989.
[12] C. Mohan et al., "ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging", ACM Trans. on Database Systems, Vol. 17, No. 1, Mar. 1992, pp. 94-162.
[13] J. Rao and K. A. Ross, "Cache Conscious Indexing for Decision-Support in Main Memory", Proc. Intl. Conf. on Very Large Data Bases, VLDB, 1999, pp. 78-89.
[14] J. Rao and K. A. Ross, "Making B+-Trees Cache Conscious in Main Memory", Proc. Intl. Conf. on Management of Data, ACM SIGMOD, 2000, pp. 475-486.



Duck-Ho Bae

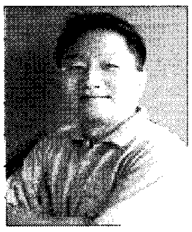
In 2006, he received the B.S. degree in Information & Communications from Hanyang University, Seoul, Korea. Now, he is working on the Master's degree in Electronics and Computer Engineering from Hanyang University. His research interests include embedded database systems, flash memory databases, mobile databases, and social network analysis.

Jong-Dae Kim

In 2003, he received the B.S. degree in Computer Science from Soongsil University, Seoul, Korea. Now, he is working on the Master's degree in Electronics and Computer Engineering from Hanyang University. His research interests include moving object management, data mining, embedded database systems, flash memory databases.

Semi Par

In 2007, he received the B.S. degree in Information & Communications from Hanyang University, Seoul, Korea. Now, he is working on the Master's degree in Electronics and Computer Engineering from Hanyang University. His research interests include embedded database systems, flash memory databases, mobile databases, and social network analysis.



Sang-Woo Kim

School of Information and Communications Hanyang University, Korea email: wook@hanyang.ac.kr Sang-Wook Kim received the B.S. degree in Computer Engineering from Seoul National University, Seoul, Korea at 1989, and earned the M.S. and Ph.D. degrees in Computer Science from Korea Advanced Institute of Science and Technology (KAIST), Daejeon, Korea at 1991 and 1994, respectively. From 1994 to 1995, he worked with the Information and Electronics Research Center in Korea, as a Senior Engineer. From 1995 to 2003, he served as an Associate Professor of the Division of Computer, Information, and Communications Engineering at Kangwon National University, Chunchoen, Kangwon, Korea. In 2003, he joined Hanyang University, Seoul, Korea, where he currently is a Professor at the School of Information and Communications. From 1999 to 2000, he worked with the IBM T. J. Watson Research Center, Yorktown Heights, New York, as a Post-Doc. He also visited the Computer Science Department of Stanford University as a Visiting Researcher in 1991. He is an author of over 80 papers in refereed international journals and conference proceedings. His research interests include storage systems, transaction management, main-memory DBMSs, embedded DBMSs, data mining, multimedia information retrieval, and geographic information systems, web data analysis. He is a member of the ACM and the IEE