Design and Implementation of a Cache-Conscious Index Manager for the Tachyon, a Main Memory DBMS

Kyung-Tae Lee, Inbum Jung, Chang-Yeol Choi
Department of Computer, Information, and Communications
Kangwon National University, Korea
mlogue@nate.com, ibjung@kangwon.ac.kr, cychoi@kangwon.ac.kr

Wan Choi
Division of Computer Systems
Electronics and Telecommunications Research Institute, Korea
wchoi@etri.re.kr

Sang-Wook Kim
School of Information and Communications
Hanyang University, Korea
wook@hanyang.ac.kr

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. In this paper, we discuss the cache-conscious index manager of the Tachyon, a next generation MMDBMS. The 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. Recently, the gap between the CPU processing and main memory access times is becoming much wider due to rapid advance of CPU technology. By devising data structures and algorithms that utilize the behavior of the cache in CPU, we are able to enhance the overall performance of MMDBMSs considerably. In this paper, we address the practical implementation issues and our solutions for them obtained in developing the cache-conscious index manager of the Tachyon. The main issues touched are (1) consideration of the cache behavior, (2) compact representation of an index entry, (3) support of variable-length keys, (4) support of multiple-attribute keys, (5) support of duplicated keys, and (6) definition of the system catalog for indexes. We also show the effectiveness of our approach through extensive experiments.

1. Introduction

Recently, real-time systems are expanding their application areas thanks to ever-growing computer technology. One of the promising approaches to manage real-time data is to replace disk with faster main-memory for their storage[22]. 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 disk-based DBMS[2][5][8].

The Real-time DBMS Team at Electronics & Telecommunications Research Institute and Data & Knowledge Engineering Lab. at Kangwon National University in Korea have been working together to develop the *Tachyon*, a next generation MMDBMS[2][8]. The Tachyon supports a deadline concept because it considers real-time applications as its major target. The Tachyon employs main-memory as a primary storage for performance reasons and hires the object-relational data model to accommodate diverse applications easily.

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[14], AVL-tree[14], T-tree[15], B-tree[4], CSS-tree[19], and CSB$^+$-tree[20] are the typical examples.

Previous research efforts mainly focus 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 the index manager of the Tachyon.

The main issues discussed in this paper are: (1) consideration of the cache behavior, (2) compact representation of an index entry, (3) support of variable-length keys, (4) support of multiple-attribute keys, (5) support of duplicated keys, and (6) definition of system catalog for indexes.

The paper is organized as follows. Section 2 briefly reviews the characteristics, overall architecture, and major components of the Tachyon. Section 3 introduces previous index structures and addresses their strong and weak points. Section 4 introduces the characteristics of the cache and the cache-conscious index adapted as an index structure in the Tachyon. Section 5 presents our strategies in developing the index manager of the Tachyon in detail. Section 6 summarizes and concludes the paper.

2. Tachyon

The Tachyon is a real-time MMDBMS that supports an object-relational model. Figure 2.1 shows the overall system architecture of the Tachyon. The *main-memory manager* is in charge of a main-memory pool. It allocates variable-length main-memory chunks from the pool and deallocates unnecessary chunks to the pool.

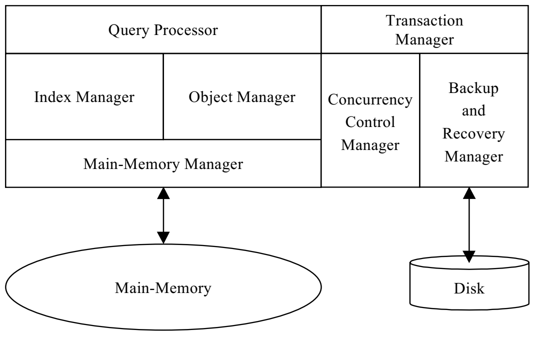


Figure 2.1. Overall system architecture.

The *object manager* stores and manages user data in the form of objects. A fixed-length main memory unit, called *partition*, stores one or more objects. The *index manager* speeds up the retrieval of qualified objects from a database. To facilitate this, the index manager selects one or more attributes as a key and builds an index from them. In multi-user environment, several transactions run concurrently. The *concurrency control manager* guarantees the logical and physical consistency of a database by controlling the execution orders of such transactions. The *backup and recovery manager* detects various transaction failures or system failures, and restores a database to a consistent state. Finally, the *query processor* optimizes declarative SQL-based queries[6] and converts it to a series of lower-layer function calls.

3. Index Structures

3.1. 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 much better performance for range queries because index entries having adjacent key values can be easily accessed with a sorted order.

The binary search tree[12] has a simple structure. Since the binary search tree is not balanced, however, its search performance heavily depends on the distribution of key values and the insertion/deletion orders of objects.

The AVL-tree[14] 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 also accomplishes the balancing property using the *rotation* operation. The biggest problem of the AVL-tree, however, is its storage overhead[15].

The T-tree[15] 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 much smaller.

The B^+-tree[4] 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 the tree traverse.

The CSB^+-tree[20], which was developed by considering the *cache behavior*, is a variant of the B^+-tree. The B^+-tree stores (number of key values + 1) child node pointers in each internal node, but the CSB^+-tree stores all the child nodes of any given internal node in contiguous memory space, and keeps only its first child pointer. The rest of the child nodes can be found by adding their offsets to that pointer. Because the number of key values stored in a node of the CSB^+-tree is larger than that in a node of the B^+-tree, search performance of the CSB^+-tree improves significantly.

3.2. 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.

The chained-bucket hashing[14] 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, however, it is not easy to estimate an optimal size for a hash table.

The extensible hashing[7] consists of data pages and a directory. Data pages store data objects and a directory stores pointers to data pages. The directory has 2^k(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[16] also has a dynamic structure. It maintains data pages in physically contiguous space, thus makes page addressing simple by calculation without the directory. The linear hashing permits overflow chains, which may cause search performance to degrade. Whenever necessary, it splits data pages in a pre-defined order, resulting in high space utilization.

Lehman et al.[15] modifies the linear hashing to be suitable for MMDBMSs. The directory is re-introduced in locating data pages for making it unnecessary to store data pages in physically contiguous pages. Since the modified liner hashing does not allocate empty pages, it has space utilization much better than the extensible hashing.

3.3. Our choice for Tachyon

The CSB$^+$-tree has the balancing property and also small storage space 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 CSB$^+$-tree stores multiple entries in a node, its storage overhead is small. Dynamic allocations and deallocations of nodes make the CSB$^+$-tree adapt to dynamic situations. The CSB$^+$-tree performs well for range queries, and is also capable of processing exact-match queries via tree traversal.

For these reasons, we chose the CSB$^+$-tree as an index structure in the Tachyon 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.

4. Cache-Conscious Index

4.1. Cache

The cache memory is a small fast static RAM that speeds up running of processes by holding recently referenced data within it[21]. The cache block is a basic transferring unit between the cache and main memory, and its typical size ranges from 32 bytes to 128 bytes.

When memory references are satisfied by the cache, we say the *cache hits* to occur. In this case, the process proceeds at the CPU speed. In contrast, when memory references are not satisfied by the cache, we say the *cache misses* to occur. In cases of the cache misses, the process proceeds at the memory speed since it has to fetch the corresponding cache block from main memory.

The traditional assumption widely-accepted in the area of computer architecture was that the costs of memory references are almost the same independent of their locations. However, this assumption is no longer valid due to the big speed gap between cache and main memory. Reference [1] studied the performance of several commercial DBMSs in main memory. The conclusion they reached was that a significant portion of execution time is spent on cache misses. Therefore, making the *cache hit ratio* higher would be a imperative task in MMDBMSs[21].

4.2. CSB$^+$-tree

The CSB$^+$-tree[20], which was developed by considering the cache behavior, is a variant of the B$^+$-tree. Every node in a CSB$^+$-tree of order d contains m keys, where $d \le m \le 2d$. The CSB$^+$-tree stores all the child nodes (we call them a *node group*) of any internal node in a contiguous memory space, and keeps only the first child pointer in each internal node. The rest of child nodes in a node group can be found by adding their offsets to the first child pointer.

Since the internal node in the CSB$^+$-tree stores only one child node pointer, it stores a more number of key values than the B$^+$-tree. This feature provides two advantages. First, this feature incurs a smaller number of cache misses than the B$^+$-tree in index searching. This is because the CSB$^+$-tree has more key comparisons in a node than the B$^+$-tree. Second, because the fan-out of each internal node gets larger, the CSB$^+$-tree uses storage space less than the B$^+$-tree.

Figure 4.1 depicts an example of the CSB$^+$-tree of order 1. Each dashed box represents a node group. The arrows from internal nodes represent the first child pointers. All the nodes within a node group are stored physically adjacent to one another in memory space.

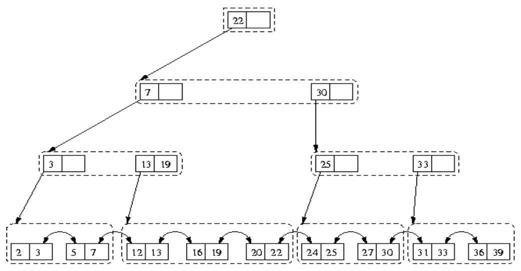


Figure 4.1. A CSB$^+$-tree of order 1[Rao00]

5. Development of Index Manager in Tachyon

5.1. Entry structure

Reference [13] proposes to store only an object address without maintaining its key value in an index entry. This method is feasible in that we can easily access the key value from the object in main memory by using the object address. However, it deteriorates the performance of search, insertion, deletion operations since every object access incurs a cache miss.

For resolving this problem, we adopt the approach to store a key value on an entry in this implementation. Thus, an entry has a form of <key value, object address>. Compared with the method in [13], this approach incurs a large storage overhead, and also makes the complexity of DBMS algorithms get higher. However, it achieves the high performance of search, insertion, deletion operations by avoiding additional cache misses in key comparisons.

A key is called a *variable-length key*[10] when the lengths of key values are variable depending on objects. In case of the variable-length key, each key value is represented as <length, attribute value>

in an index entry. In case of the fixed-length key, the lengths of key values are identical in all objects. Thus, only <attribute value> is stored without the length field in an index entry.

The *multiple-attribute key*[10] is defined as a key that consists of more than one attribute. If an attribute is of variable-length, it is represented as <length, attribute value>. Otherwise, only <attribute value> is kept without the length field.

The *duplicate key*[10] allows different objects have the same value for that key. In case of the duplicate key, an entry consists of <key value, list of object addresses>. It is possible for an entry to get larger than a node due to so many duplicates of the same key value. In this case, the chain of overflow nodes is introduced, which will be elaborated in Section 5.2.

5.2. Node structure

The CSB$^+$-tree is composed of three types of nodes: the internal, leaf, and overflow nodes.

The internal node has the structure of *<data[ ], p0, nEntries, freeOffset, type>*. data[ ] stores index entries, and occupies all the space of a node except for *p0, nEntries, freeOffset*, and *type*. *p0* points to the first child of this node. *nEntries* represents the number of entries currently stored in this node. *freeOffset* indicates the offset from which the contiguous free space starts in the node. Finally, *type* has a value of *INTERNAL,* which indicates the type of this node. Normally, a cache block is smaller than or equal to 256 bytes. Thus, we hire "unsigned char" type of one byte for *nEntries, freeOffset*, and *type* fields for space optimization.

In case of the ordinary B$^+$-tree, there are *slot[i]* fields in a node[10]. Each *slot[i]* corresponds to an index entry, and has the offset from which the entry starts in the node. The reason for using slot[i] is that it enables the binary search even in case of variable-length keys[10]. However, this employing of *slot[i]* makes the number of entries stored in a node get smaller, thus increases the number of cache misses in tree traversal.

In our implementation, we do not hire the *slot[i]* concept for this reason. In case of fixed-length keys, we are able to perform the binary search in a node. In case of variable-length keys, however, we have to perform the linear search in a node. The linear search is more costly than the binary search. However, we note that the number of index entries in a cache block is not that large, and also the linear search in the node performs within the cache rather than main memory. Therefore, the cost of the linear search is not that serious compared with that of the cache miss.

The leaf node has the structure of *<data[ ], nEntries, freeOffset, (prefixLen), prevNode, nextNode, type>*. The fields *data[ ], nEntries, freeOffset,* and *type* are used for the same purpose as in the internal node. *type* has a value of *LEAF,* which indicates the type of this node. *prevNode* and *nextNode* are pointers for keeping all the leaf nodes as a doubly-linked list. *(prefixLen)* is used only for variable-length keys, and is the size of the common prefix of the key values in a node. So, the common prefix is stored once in a node, and only the remaining part is stored in each entry. This is to reduce the number of cache misses by raising storage utilization. Especially, this effect of performance improvement gets higher as a database becomes larger. As in the internal node, we hire "unsigned char" type of one byte for *nEntries, freeOffset, (prefixLen)*, and *type* fields for space optimization.

The overflow node has the structure of *<data[ ], prevNode, nextNode, freeOffset, type>*. The fields *data[ ], freeOffset,* and *type* are used for the same purpose as in the leaf node. *type* has a value of *OVERFLOW,* which indicates the type of this node. *prevNode* and *nextNode* are pointers for keeping all the overflow nodes as a doubly-linked list. As in the internal node, we hire "unsigned char" type of one byte for *freeOffset* and *type* fields.

5.3. System catalog information for CSB$^+$-tree

For key comparisons, however, we need to know which attributes in an object comprise a key. For this purpose, the system catalog maintains useful information on the CSB$^+$-tree in *CSBtreeInfo* as *<UorD, root, numAttributes, attrDesc[0], attrDesc[1], ..., attrDesc[MAX-1]>*.

The field *UorD* indicates whether a duplicate key is allowed or not. The field *root* stores the pointer to the root node of the CSB$^+$-tree and the field *numAttributes* the number of organizing attributes. *CSBtreeInfo* also stores the information on each organizing attribute in the field *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>*. The field *offset* is the starting position of an attribute within an object. The fields *size* and *dataType* are the maximum size and data type of the attribute, respectively. The comparison for the multiple-attribute key is performed by repeatedly comparing all the organizing attributes that comprise the key.

6. Performance Evaluation

6.1. Experiment environment

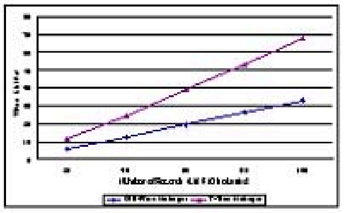
The hardware platform used in our experiments is a Sun-Ultra-10 workstation equipped with UltraSparc-IIi processor of 440 MHz speed, 2 Mbytes cache, and 1 Gbytes main memory. The software platform is the operating system of SunOS 5.7 with C++ compiler of g++ 3.0.1. As the competitor of the CSB$^+$-tree manager, we used the T-tree manager employed in our previous version of the Tachyon. In both cases, we adopted the 64 bytes nodes in the experiments which have the size same as that of a cache block. We evaluated the time performance by performing insertion, deletion, and search operations on one million key values.

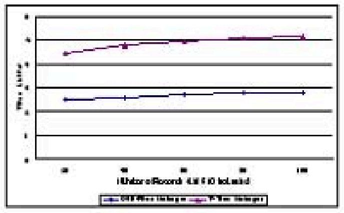
6.2. Results and analyses

Figures 6.1 shows the results of performing insertions, exact-match queries, range queries in case of the fixed-length(4 byte integer), single-attribute, and non-duplicate key. Figure 6.1(a) shows the insertion times when we insert 1,000,000 key values in steps of 200,000 key values. The result reveals that our CSB$^+$-tree manager achieves about 1.8 times speedup in insertions over the T-tree manager. This performance improvement of our CSB$^+$-tree manager is due to the smaller number of cache misses occurred in each insertion. The two major reasons of this are summarized as follows: (1) The CSB$^+$-tree is inherently cache-conscious since it has only one child pointer p0; (2) Key comparisons in a node do not incur additional cache misses in our CSB$^+$-tree since the node stores the key values within it. Figure 6.1(b) shows the search times when we perform 100,000 exact-match queries on 200,000, 400,000, 600,000, 800,000, and 1,000,000 key values. The result reveals that our CSB$^+$-tree manager achieves about 1.8 times speedup in exact-match queries over the T-tree manager. Figure 6.1(c) shows the search times when we perform range queries with changing selectivities of 5%, 10%, 15%, and 20% on 1,000,000 key values. The result reveals that the performance of our CSB$^+$-tree manager is about 1.6 times better than that of the T-tree manager.

Figures 6.2 shows the results of performing insertions, exact-match queries, range queries in case of the variable-length(7~10 byte character string), single-attribute, and non-duplicate key. Figure 6.2(a) shows the times spent in inserting 200,000, 400,000, 600,000, 800,000, and 1,000,000 key values into each tree. The result implies that our CSB$^+$-tree manager runs about 1.5 times faster than the T-tree manager. Figure 6.2(b) shows the times spent in performing 100,000 exact-match queries on 200,000, 400,000, 600,000, 800,000, and 1,000,000 key values. Our CSB$^+$-tree manager is shown to perform about 1.7 times faster than the T-tree manager. Figure 6.2(c) shows the times spent in performing 100,000 range queries with changing selectivities of 5%, 10%, 15%, and 20% on 1,000,000 key values. We see that the performance of our CSB$^+$-tree manager is about 1.5 times better than that of the T-tree manager.

Figures 6.3 shows the results of performing insertions, exact-match queries, range queries in case of the fixed-length(4 byte integer), multiple-attribute, and non-duplicate key. Figure 6.3(a) shows the times spent in inserting 1,000,000 key values with different numbers of organizing attributes into each tree. As the number of organizing attributes increases, the performance of our CSB$^+$-tree manager decreases while there are no changes in that of the T-tree manager.


(a) Insertions.


(b) Exact-match queries.

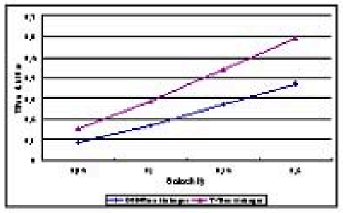
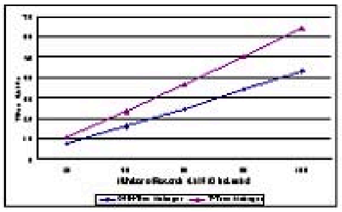
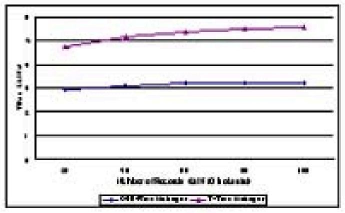

(c) Range-queries.

Figure 6.1. Results on the fixed-length, single-attribute, and non-duplicate keys.


(a) Insertions.


(b) Exact-match queries.

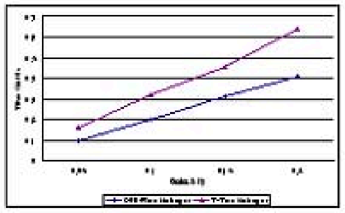
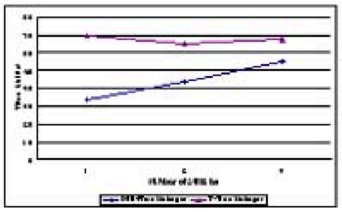
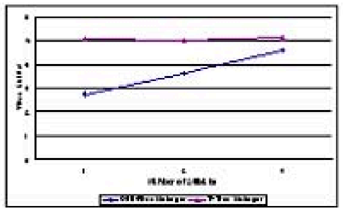

(c) Range queries.

Figure 6.2. Results on the variable-length, single-attribute, and non-duplicate keys.


(a) Insertions.


(b) Exact-match queries.

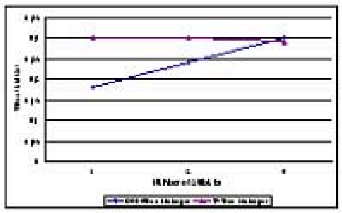

(c) Range queries.

Figure 6.3. Results on the fixed-length, multiple-attribute, and non-duplicate keys.

The reason for this is summarized as follows: multiple organizing attributes make the length of the index entry in our CSB$^+$-tree manager increase, and subsequently make the height of the CSB$^+$-tree get higher. This has the tree traversal and node splitting take more times. On the other hand, the number of organizing attributes does not affect the length of the index entry in the T-tree manager since key values are not stored in an index entry. Thus, the insertion performance is kept constant regardless of the number of organizing attributes.

Figure 6.3(b) shows the times spent in performing 100,000 exact-match queries on 1,000,000 key values with changing number of organizing attributes. The result is nearly identical to that in Figure 6.3(a). Figure 6.3(c) shows the times spent in performing 100,000 range queries with selectivity of 5% on 1,000,000 key values with changing number of organizing attributes. We observe that the result is quite similar to those in Figures 6.3(a) and 6.3(b).

7. Conclusions

MMDBMSs provide a promising solution to improve DBMS performance by replacing disk with main memory as storage media. Recently, MMDBMSs are expanding their application areas owing to fast growth of main memory technology. The index manager is an essential DBMS sub-component that supports the fast retrieval of target objects. This paper has investigated practical issues experienced in developing the index manager of the Tachyon, and has proposed our approaches to them. The main issues discussed are: (1) consideration of the cache behavior, (2) compact representation of index entries, (3) support of variable-length keys, (4) support of multiple-attribute keys, (5) support of duplicate keys, and (6) definition of system catalog for index. We have also verified the efficiency of our approach via extensive experiments.

Acknowledgment

Sang-Wook Kim was supported by the 2002 Basic Research Program(Grant No. R05-2002-000-01085-0) of the Korea Science & Engineering Foundation(KOSEF) and also by the research fund of Hanyang University(HY-2003). Inbum Jung was partially supported by 2003 Basic Research Program(Grant No. R05-2003-000-12146-0) of the Korea Science & Engineering Foundation(KOSEF). This work was also partially supported by the Brain Korea 21 Project in 2003 through Kangwon National University. Sang-Wook Kim would like to thank Jung-Hee Seo, Suk-Yeon Hwang, Grace(Joo-Young) Kim, and Joo-Sung Kim for their encouragement and support.

References

[1] A. Ailamaki, D. j. DeWitt, M. D. Hill, and D. A. Wood, "DBMSs on a Mordern Processor: Where Does Time Go?," *In Proc. Intl. Conf. on Very Large Data Bases, VLDB*, pp. 266-277, 1999.

[2] A. Ammann, M. Hanrahan, and R. Krishnamurthy, "Design of a Memory Resident DMBS," *Proc. Intl. Conf. on COMPCON*, Feb. 1985.

[3] P. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*, Addison-Wesley, 1987.

[4] D. Comer, "The ubiquitous B-Trees," *ACM Computing Surveys*, Vol. 11, No. 2, pp. 121-137, 1979.

[5] D. DeWitt et al., "Implementation Techniques for Main Memory Database Systems", *Proc. Intl, Conf. on Management of Data, ACM SIGMOD*, pp. 1-8, 1984

[6] R. Elmasri and S. B. Navathe, *Fundamental of Database Systems*, Second Edition, Benjamin/Cummings Publishing Company, 1994.

[7] R. Fagin et al., "Extensible Hashing: A Fast Access Method for Dynamic Files," *ACM Trans. on Database Systems*, Vol. 4, No. 3, pp. 315-344, 1979.

[8] H. Garcia-Molina and K. Salem, "Main Memory Database Systems: An Overview," *IEEE Trans. on Knowledge and Data Engineering*, Vol. 4, No. 6, pp. 509-516, 1992.

[9] J. Gray et al., "Granularity of Locks in a Shared Database," *Proc. Intl, Conf. on Very Large Data Bases, VLDB*, pp. 428-451, Sept, 1975.

[10] J. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques*, Morgan Kaufman Publishers, 1993.

[11] T. Haeder and A. Reuter, "Principles of Transaction-Oriented Recovery," *ACM Computing Surveys*, Vol. 15, No. 4, pp. 287-317, Dec. 1983.

[12] E. Horowitz, S. Sahni, and S. Freed, *Fundamentals of Data Structures in C*, Computer science Press, 1993.

[13] S. Kim et al., "Design and Implementation of the Index Manager in the Main Memory DBMS," I*n Proc, Intl, Symp. on Database and Applications(DBA 2002)*, pp. 473-478, 2002.

[14] D. Knuth, *The Art of Computer Programming*, Addison-Wesley, 1973

[15] T. Lehman and M. Carey, "A Study of Index Structures for Main Memory Database Management System," *Proc. Intl. Conf. on Very Large Data Base, VLDB*, pp. 294-303, Aug. 1986.

[16] W. Litwin, "Linear Hashing: A New Tool For File and Table Addressing," *Proc. Intl. Conf. on Very Large Data Bases, VLDB*, pp. 212-223, 1980.

[17] 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.

[18] 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, pp. 94-162, Mar. 1992.

[19] J. Rao and K. A. Ross, "Cache Conscious Indexing for Decision-Support in Main Memory," *In Proc. Intl. Conf. on Very Large Data Bases, VLDB,* pp. 78-89, 1999.

[20] J. Rao and K. A. Ross, "Making B+-Trees Cache Conscious in Main Memory," *In Proc. Intl. Conf. on Management of Data, ACM SIGMOD*, pp. 475-486, 2000.

[21] A. J. Simith, "Cache Memories," *ACM Computing Surveys*, Vol. 14, No. 3, pp. 473-530, 1982.

[22] S. H. Son(Editor), *Special Issue on Real-Time Database Systems, ACM SIGMOD Record*, Vol. 17, No. 1, Mar. 1988.

[23] J. S. M. Verhofstad, "Recovery Techniques for Database Systems," *ACM Computing Surveys*, Vol. 10, No. 2, pp. 167-195, Dec. 1978.

IEEE COMPUTER SOCIETY