Design and Implementation of the Index Manager for a Main Memory DBMS Tachyon

Sang-Wook Kim
Department of Computer, Information, and Communications Engineering
Kangwon National University

Sang-Hyun Park
Data Management Group
IBM T.J. Watson Research Center

Wan Choi
Realtime DBMS Group
Electronics and Telecommunications Research Institute

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 experiences obtained in developing the index manager of the *Tachyon*, a next generation MMDBMS. 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 of the Tachyon as solutions. The main issues touched are (1) compact representation of an index entry, (2) support of variable-length keys, (3) support of multiple-attribute keys, (4) support of duplicated keys, (5) concurrency control, and (6) backup and recovery.

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 disk-based DBMS [1][3][5]. Realtime DBMS Team at ETRI and Data & Knowledge Engineering Lab. at Kangwon National University have been working together to develop the *Tachyon*, a next generation MMDBMS. The Tachyon employs main-memory as a primary storage 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, AVL-tree, T-tree [8], B-tree [2], chained-bucket hashing, extensible hashing [4], and linear hashing [10] are the typical examples. Recently, 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 the index manager of the Tachyon. The main issues discussed in this paper are: (1) compact representation of an index entry, (2) support of variable-length keys, (3) support of multiple-attribute keys, (4) support of duplicated keys, (5) concurrency control, and (6) backup and recovery.

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 presents the issues and their solutions experienced in developing the index manager of the Tachyon in detail. Section 5 presents our strategies employed for recovery in the index manger of the Tachyon. Finally, Section 6 summarizes and concludes the paper.

2 Tachyon

This section briefly introduces the Tachyon, a next generation DBMS.

The Tachyon is:

1. a real-time DBMS that has a deadline concept for supporting real-time applications.

2. a main-memory DBMS that maintains all the database in main-memory for achieving high performance.

3. an object-relational DBMS that supports both relational and object-oriented data models for accommodating diverse applications.

The Tachyon has been built with C++ programming language on a UNIX-based platform. Eventually, however, the Tachyon would be migrated to a *SROS-based* platform[1].

Figure 1 shows the overall system architecture of the Tachyon. The *main-memory manager* is in charge of a main-memory pool. It allocates/deallocates variable-length main-memory chunks from/to the pool. 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. A set of partitions, called segment[2], maintains all objects with the same type. A database consists of multiple segments. 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 may be executed 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 or system failures and restores a database to a consistent state. For this, the backup and recovery manager also writes *log records* [6] corresponding to normal update operations and periodically makes backup copies of main-memory data in disk. Finally, the *query processor* optimizes declarative SQL-based queries and converts it to a series of lower-layer function calls.

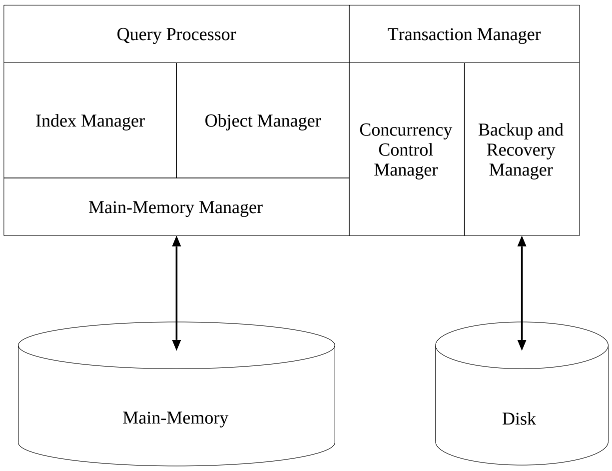


Figure 1: Overall System Architecture of the Tachyon.

[1]SROS [7] is a real-time operating system developed at ETRI.
[2]It corresponds to a relation in a relational database.

3 Index Structures

This section reviews previous index structures proposed for database systems.

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 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.

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 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 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 [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 liner hashing does not allocate empty pages, it has space utilization much better than the extensible hashing.

3.3 Our Choice for Tachyon

For the Tachyon, we selected its 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 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 Design and Implementation 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 of the Tachyon.

4.1 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.

The $minKey(N)$ 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[3], we delete that node from the T-tree. We perform the rotation operation if the T-tree gets unbalanced by this node-deletion.

4.2 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.

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

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*, $\langle attrDesc[0], attrDesc[1], ..., attrDesc[MAX-1]\rangle$.

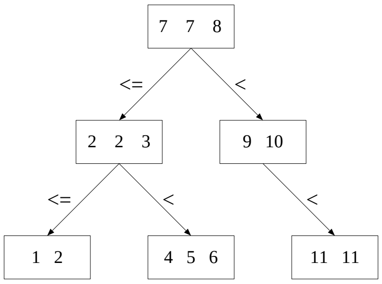
UorD indicates whether a duplicate key is allowed or not. *root* stores the address of the root node of the T-tree and *numAttributes* the number of organizing attributes. TtreeInfo also stores the information on each organizing attribute in $attrDesc[i]$. As a result, the multiple-attribute key, which 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.

4.3 Support of duplicate keys

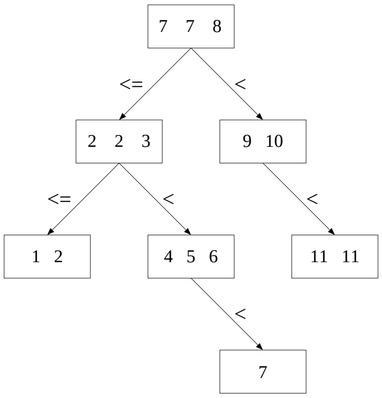
When a system allows duplicate keys, different objects may have the same key value. The simple way [6] is to replace the structure of index entries with ⟨key value, list of object addresses⟩. This makes the index entries be of variable length. Furthermore, overflow nodes have to be introduced since the size of an index entry possibly gets larger than that of a leaf page. All of these make the algorithms in other DBMS sub-components complicated.

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 2, we observe that the only change is to replace the sign "$<$" in left child pointer with "$\leq$".

Figure 2 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 2, 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.



(a) before insertion.



(b) after insertion.

Figure 2: Inserting the duplicate key 7.

5 Concurrency Control and Recovery

In this section, we suggest efficient methods for concurrency control and recovery used in the index manager of the Tachyon.

5.1 Concurrency control

The two-phase locking protocol (2PL) is the most popular one to maintain the consistency of objects in multi-user environment. We can consider employing the 2PL to maintain the consistency of indexes, too. However, it causes severe degradation in concurrency because most transactions access indexes. If every transaction needs to acquire a lock on the root node of an index and hold that lock until its completion, all transactions run serially.

To resolve this problem, we employ a *latch* [11][12], a cheaper variation of a lock. Each transaction acquires a latch on the entire T-tree just before accessing the tree and releases the latch right after completing the access. A latch scheme is enough to guarantee the consistency of indexes because the index is just *meta data*. Users do not care about the content of the index itself but see the actual objects stored in a database. We have to hire the 2PL if we need to maintain the *logical consistency* of the index. But, what we exactly want to assure is its *physical consistency* to maintain a correct structure. This makes the latch feasible in our case.

Mohan et. al. [11] proposed the ARIES/IM that also maintains the physical consistency of the B-tree in disk-based DBMSs. Each node has its own latch and a transaction acquires the latch just before accessing the node and releases the latch right after accessing the node. If the ARIES/IM detects an inconsistency in the B-tree, it forces transactions to obtain the *tree latch*, which is assigned to each B-tree, and thus prohibits transactions from accessing inconsistent B-tree contents.

The latch granule of the ARIES/IM (a node) is smaller than that of our approach (a tree). Transactions accessing different parts of the same tree could run concurrently in the ARIES/IM while they have to run serially in our approach. Thus, the ARIES/IM provides concurrency higher than ours. The tree traversal accompanies multiple disk accesses in disk-based DBMSs. A latch has to be held on a tree until a transaction completes multiple disk accesses. Therefore, high concurrency for indexes is conspicuous in disk-based DBMSs. On the contrary, the duration during which a latch is held on a T-tree is very short in MMDBMSs since the tree traversal does not incur disk accesses. Therefore, high concurrency for indexes is negligible in MMDBMSs.

The ARIES/IM is very sophisticated and needs a lot of latches to be acquired and released during a lifetime of a transaction. The cost for acquiring or releasing latches is not that significant in disk-based DBMSs since the tree traversal incurs multiple disk accesses. In MMDBMSs, however, this is not the case since the cost occupies a fairly large portion of the entire cost for the tree traversal. These reasons make us decide to choose the tree as a latch granule.

5.2 Phantom problem

A transaction may want to retrieve a set of objects satisfying a given condition multiple times within its lifetime. In situations where insertions and deletions are frequent, the returned object set could be different depending on the retrieval time. We call this phenomenon *phantom problem* [6]. As a practical solution to the phantom problem, Mohan et. al. [11] proposed a method named the *next-key locking*. We modify the next-key locking method to be suitable for MMDBMSs.

Let us use the notations $minKey(C)$ and $maxKey(C)$ for the minimum and maximum key values satisfying a query condition C. The Tachyon employs the partition, a fixed-size memory chuck, as a storage unit of objects and a lock granule. For a range query with a condition C, we request shared locks on all the partitions that include at least one object whose key value is between $minKey(C)$ and $maxKey(C)$ or whose key value is right after $maxKey(C)$. Likewise, for a transaction that inserts or deletes an object O, we request exclusive locks on all the partitions that contain O or an object whose key value is right after the key value of O.

Suppose that a transaction T1 is accessing a set of objects that satisfies its query condition and another transaction T2 wants to insert an object O that also satisfies its query condition. In this case, a transaction T2 could not acquire the lock for the partition containing the object whose key value is right after that of O until T1 completes its execution. Thus, this prevents the phantom problems caused by insertion. Similarly, when deleting an object O, our modified next-key locking requests two exclusive locks: one lock on the partition containing O and the other (an additional one) on the partition containing the object whose key value is right next to the key value of O. This solves the phantom problems caused by rollbacks of transactions that have deleted objects. As a result, our method effectively guarantees no phantom problem with a smaller number of additional locks.

Our method assumes that users issue range queries only on the attributes with indexes. Otherwise, we have to scan all segments to find the object that has the next-key value. The assumption is realistic and agrees with most situations since we normally build indexes on such attributes that may be used for range queries through physical database design.

5.3 Backup and recovery

Backup and recovery are the fundamental DBMS features to protect a database against various transaction or system failures [6]. The backup manager accumulates log records for normal update operations and periodically makes backup copies of a current database in disk. When detecting system failures, the recovery manager brings the corrupted database into consistent state using the log records and the latest backup copy.

Logging is the primary reason to degrade the performance of MMDBMSs since it carries disk accesses while other operations perform just in main memory. If we hire the logging mechanism for indexes widely used in disk-based DBMSs, MMDBMSs would suffer from serious performance degradation caused by writing a lot of log records into disk[4]. Thus, we have to devise an effective method that minimizes the number of log records necessary for recovery purposes.

Motivated by the fact that indexes are the meta data that could be recreated by the objects in a database, the Tachyon employs the following approach: (1) it first recovers the objects in a database using the same recovery algorithm as in disk-based DBMSs; (2) it rebuilds all the indexes by referring to the recovered objects. The proposed approach has the following advantages: (1) transaction processing gets faster since we do not create any log records for indexes during normal

[4]Note that a single operation for inserting or deleting an index entry could generate more than one log record since it may update more than one index page.

processing; (2) backup processing gets faster since we do not need to make backup copies for indexes; (3) recovery processing also gets faster since it needs to access a smaller number of log records and a smaller size of a backup copy from disk; (4) backup and recovery algorithms become much simpler.

Indexes have to be recovered correctly when a transaction fails during its normal operation. For this purpose, we make a single logical log record for each update on an index. For example, when a transaction inserts an index entry e into an index I, we generate a log record in the form of $\langle Insert, I, e \rangle$. If the transaction fails, the recovery manager performs its reverse operation that deletes e from I. The Tachyon keeps the logical log records in main memory just while the transaction runs, and discards them when the transaction ends. Thus, the logical logging for indexes does not incur any disk accesses.

In summary, our approach enhances all the performance in normal processing, backup processing, recovery processing from system failures, and recovery processing from transaction failures. Considering that all the database resides on main memory, we expect that the effect of performance improvement would be significant in MMDBMSs.

6 Conclusions

MMDBMSs provide a promising solution to innovate on 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 the index manager of the Tachyon, 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, (4) support of duplicate keys, (5) concurrency control, and (6) backup and recovery. 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.

Acknowledgment: This research was supported by the 1999 Post-Doc. Program and the 2000 Basic Research Program (Grant No. 2000-1-51200-006-1) of the KOSEF. We would like to appreciate contributions of Sang-Min Yeom and Yun-Ho Kim, who implemented all the ideas presented in this paper on the Tachyon Index Manager. Sang-Wook Kim would also like to thank Jung-Hee Seo, Suk-Yeon Hwang, and Grace(Ju-Young) Kim for their encouragement and support.

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, pp. 121-137, 1979.

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

[4] 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.

[5] 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.

[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*, pp. 2918-2923, 1998.

[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*, pp. 294-303, Aug. 1986.

[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*, pp. 104-117, 1987.

[10] 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.

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

[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*, pp. 78-89, 1999.

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