# T*-tree : A Main Memory Database Index Structure for Real Time Applications

Kong-Rim Choi, Kyung-Chang Kim
Dept. of Computer Science, Hong-Ik Univ. Seoul Korea
Tel : 02 - 320 - 1606
E-mail : *kckim @ cs. hongik. ac. kr*

## Abstract

*In this paper, we propose an indexing structure, called T*-tree, for efficient processing of real time applications under main memory database management systems (MMDBMS). T*-tree is an index structure for rapid data access and saves memory space under MMDBMS. T-tree is well known to be one of the best index structures for ordered data in MMDB. Existing T-tree is a balanced tree that evolved from AVL and B-trees, and a binary tree with many elements in a node. T-tree retains the intrinsic binary search nature, and is also good in memory use. However T-tree has a major disadvantage; the tree traversal for range queries and the movement of overflow /underflow data due to data insertion/deletion on internal nodes. We propose T*-tree as a alternative structure, which is an improvement from T-tree for better use of query operations, including range queries and which contains all other good features of T-tree. We also show the pseudo-algorithms of search, update, delete, and rebalancing operations for T*-tree, with performance test results. The results indicate that T*-tree provides better performance for range queries compared to T-tree.*

*Keywords : index structure, main memory databases, real-time application, range query*

## 1. Introduction

The doubling trend of the chip densities every year, and the rapidly decreasing cost of RAM, makes it possible for us to expect that main memory sizes of a few giga bytes or more will be feasible and perhaps even common in the near future. It is quite likely that databases, at least for some applications, will eventually fit entirely in main memory [1]. There are many application areas where real time access and high data performance are required. One of the approaches to fit the requirement of real time applications is to use huge amount of memory which can be used to store the main database. In this approach, we must make additional considerations in the design of the database management system - the algorithms and access structures for efficient query processing and for saving memory space. We propose an index structure for efficient processing of real time application under main memory databases (MMDB). Two approaches to improve performance in database management systems with large main memories, have been studied.

The first one, the existing disk-resident database approach, makes the buffer pool very large, making it possible for most or perhaps all of the data needed for each transaction to be retained in the buffer pool[8,12,14]. The other approach, the memory resident database approach, is to use the large amount of memory as the main store for database [1,2,4,5,8,9,10].

In the conventional disk based database system, access structures are usually designed to reduce the numbers of disk access. The idea is to quickly find the block where the data resides and to bring the block into memory. Once brought into memory, the entire block may need to be searched for appropriate data. But, similar structures in main memory databases may not be appropriate, since there is no disk access in memory-resident databases.

The objectives of the access structures used in main memory database systems are efficient utilization of CPU time and main memory space rather than reducing I/Os.

The remainder of this paper is organized as follows: Section 2 decribes the existing data structures; B-tree, AVL-tree, Threaded-Binary-tree and T-tree which are usually used in main memory. Section 3 introduces the new index structure, T*-tree, which evolves from T-tree for better use of main memory for real time applications. We also describe the basic algorithms of T*-tree operations. Section 4 compares T*-tree to existing index structures using execution time in certain cases and also using simulation results of several types of queries. Finally, Section 5 presents our conclusions.

## 2. Main Memory Index Structures

There has been much work in the area of MMDB data structures [1,4,5,15]. SiDBM uses the AVL-tree because of its fast search time [3]. But, it has been found that B+-trees are preferable to AVL-trees in the MMDB system due to the high percentage of the tree which must be memory resident for the tree to be competitive. Lehman proposed a new indexing structure, T-tree, designed especially for use in MMDB systems [1].

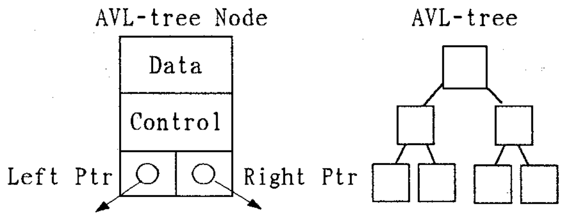
In this section, we describe the existing index structures used in main memory, such as B-tree, AVL-tree, Threaded-Binary-tree and T-tree (including the concept of T+-tree).

### 2.1 AVL-trees

AVL-trees[6] are used as indices in the AT&T Bell Laboratories Silicon Database Machine. The AVL-tree was designed as an internal memory data structure [Figure 1]. It uses a binary search, which is very fast, since the binary search is intrinsic to the tree structure (no mathematical calculations are needed). Updates always affect a leaf node and may result in an unbalanced tree, so the tree is kept balanced by rotation operations. The AVL-tree has one major disadvantage - its poor storage utilization. Each tree node holds only one data item, so there are two pointers and some control information for every data item.

### 2.2 B-trees

B-trees[7] are well suited for use in disk resident database systems, since they are broad shallow trees and require few w node accesses to retrieve a value [Figure 2]. Most database

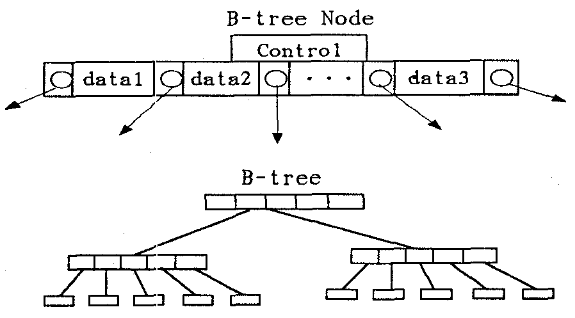


**[Figure 1] The AVL-tree**

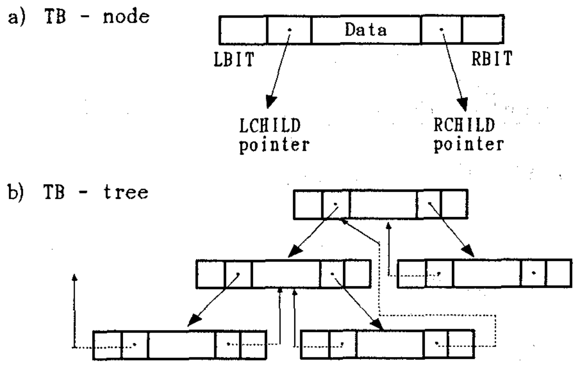
systems use a variant of B-tree, the B+-tree, which keeps all of the actual data in the leaves of the tree. For main memory use, the B-tree is preferable to B+-tree because, in the main memory there is no advantage to keeping all of the data in the leaves since it only wastes space. B-trees are good for memory use since their storage utilization is good (the pointer to data ratio is small, as leaf nodes hold only data items and they comprise a large percentage of the tree); searching is reasonably quick (a small number of nodes are searched with a binary search) ; and updating is fast (data movement usually involves only one node).

## 2.3 Threaded Binary-trees

On any binary trees, there are more null links than actual pointers. A clever way to make use of these null links has been deviced by A. J. Perlis and C. Thornton[13]. This idea is to replace the null links by pointers, called threads, to other nodes in the tree. If the pointer for rightchild is normally equal to nil, it will replace with a pointer to the node which would be visited after current node when traversing the tree in inorder. A null leftchild link at the any node is replaced by a pointer to the node which immediately precedes node of the tree in inorder. [Figure3] shows the Threaded-Binary-tree [TB-tree] which designed from binary trees with its new threads drawn in as dotted lines.



[Figure 2] The B-tree

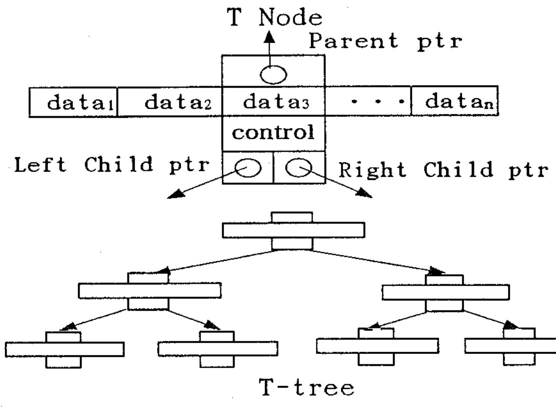


[Figure 3] The Threaded Binary tree

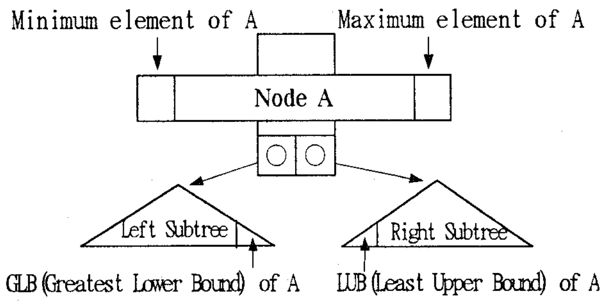
TB-trees require only one bit overhead per pointer to distinguish whether the pointer is actually a child pointer or a thread pointer and helps in ascending and descending scans for tree traversals. But, only leaf node can have a thread pointer and there is no pointer to the successor node on the internal nodes which has two child subtrees on TB-trees. Therefore, scanning to successor node on internal node must use the inorder tree traversal.

## 2.4 T-trees

T-tree was proposed as an indexing structure, especially for the main memory database systems, by Lehman. Basically, T-tree is a binary tree with many elements in each node, and is evolved from AVL-trees and B-trees. [Figure 4] shows a T-tree and a node of a T-tree, called a T Node. It retains the intrinsic binary search nature which is inherited from the AVL-tree, and has the good update and storage characteristics of the B-tree, because of having many elements in a node. To· distinguish between threads and normal pointers in the memory representation, there are two boolean fields to the structures, LBIT and RBIT. If LBITR field is true, then LCHILD contains a thread and otherwise it contains a pointer to the left child. As shown in [Figure 5], for each internal node A, there is a corresponding leaf



[Figure 4] The T-tree

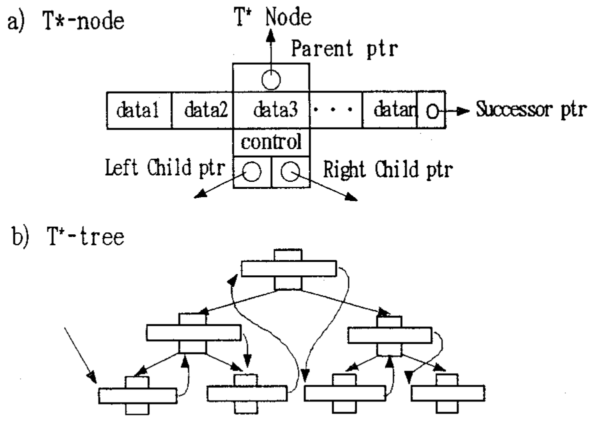


[Figure 5] The Bound of a T-node

node (or half-leaf) that holds a data value that is the predecessor to the minimum value in A, and there is also a leaf (or half leaf) that holds the successor to the maximum value in A. The predecessor value is called the GLB (greatest lower bound ) of an internal node A, and the successor value is called the LUB (least upper bound ) of a node A. Since the data in a T node is in a sorted order, its leftmost element is the smallest element in the node and its rightmost element is the largest. T-tree indexing structures, on the result of simulation experiments compared against AVL-trees, simple arrays and B-trees, have shown the best performance for ordered data in main memory database systems.

## 3. T*-tree Index Structure

### 3.1 The structure of T*-tree



[Figure 6] The T*-tree

We propose a new indexing structure called T*-tree, that is basically redesigned to solve some of the problems in the T-tree for better performance. [Figure 6] shows a node of a T*-tree, called a T*-node, and a T*-tree. The T*-tree is also a binary tree with many elements in a node, and the structure of it is exactly the same as the T-tree's, except that it has a additional pointer, called a successor pointer which points directly to the successor node for each node in T*-trees. Therefore, each node of a T*-tree has a pointer to the successor node, which makes it easier to scan sequentially by using a simple linked list rather than a tree traversal for query processing, such as range query or sequential search. This pointer can also be used to pass down directly to the leaf node due to insert/delete underflows. This is one major difference from T-tree extended with threads of TB-trees. Because the TB-trees can not have a thread pointer on internal nodes, there is no way directly to the successor node from internal node. Therefore, scanning successor nodes on internal nodes must use the inorder tree traversal.

The data items in a T*-node is also in a sorted order like T-node. The first data element is always inserted into the rightmost room for entry as a minimum value in a leaf node, while it is inserted into leftmost room as a maximum value in a T-node. This is another difference from T-tree. It make direct access for data to pass down to GLB node due to insert overflows, and to borrow from GLB node due to deleted underflows using the additional pointers.

## 3.2 The Operations for T*-trees

**3.2.1. Search Operation.** Searching in a T*-tree is exactly the same as searching in a T-tree and similar to that of a binary tree. The difference is that comparisons are made of the minimum and maximum values of the node, rather than a single value as in a binary tree node.

The searching algorithm is as follows:

(1) The search always starts at the root of the tree.

(2) If the search value is less than the minimum value of the node, then search down the left subtree pointed to by the left-child pointer.

If the search value is greater than the maximum value of the node, then search down the right subtree pointed to by the right-child pointer. Else, search the current node.

Since the data in a T*-tree node is in a sorted order, the binary search can be used. When a node is searched and the data item is not found, or when a node that bounds the search value cannot be found, the search fails and ends.

**3.2.2. Insert Operation.** The insert operation in a T*-tree is also similar to the insertion in a T-tree, and it begins with a search to get the bounding node for the insert value. The new value is inserted into the bounding node and then the tree is checked for balance. The main difference to the T-tree's insertion is that the maximum element in a node always moves directly to the LUB node and inserted into the rightmost room for entry as a minimum value due to the insertion overflows, rather than minimum element to the GLB

The algorithm of the insert operation for T*-tree works as follows:

(1) Search for the bounding node.

(2) If a node is found, then check it for room for another entry.

If the insert value will fit, then insert it into this node and stop. Or, remove the maximum element from the node, insert the original value, and make the maximum element the new value.

Proceed from here directly to LUB for the node holding the original insert value by using a successor pointer, since this new value must become a minimum value in the right subtree for the node holding the insert value.

(3) If the entire tree was checked for search, and the node to bound the insert value was not found, then insert the value into the last node on the search path.

If it has no room for the insert value, then create a new leaf node so that the insert value becomes the first element in the new leaf node, and change the successor pointer to the appropriate node.

(4) If a new node was created, then check the tree for balance by following the path from the leaf to the root. For each node in the search path, if the two subtrees of a node differ in depth by more than one level, then a rotation must be performed. Once one rotation has been done, the tree is rebalanced and processing stops.

**3.2.3. Delete Operation.** The deletion algorithm is almost the same as the insertion algorithm in the sense that the element to be deleted is searched for, the deletion operation is performed, and then rebalancing is done if necessary. If a node deletion occurs due to the delete operation, then change the successor pointer to the appropriate node, and then check the tree for balance by following the path from the leaf to the root. If the data deletion causes an underflow in the internal node, then delete the value and borrow the LUB (least upper bound) of this node from a leaf to make this node's occupancy (element count) back up to the minimum count. In this case, T*-tree can move directly to the LUB node for data borrowing by using the successor pointer, instead of the tree traversal of the T-trees.

The algorithm of the delete operation in a T*-tree, works as follows:

(1) Search for the node that bounds the delete value, and search for the delete value in the node, reporting an error and stopping if it is not found.

(2) If the delete will not cause an underflow, then delete the value and stop.

Else if this is an internal node, then delete the value and borrow the LUB value of this node from a leaf to make this node's occupancy back up to the minimum count.

Else, this is a leaf or a half-leaf node, then just delete the element, because all leaves are permitted to underflow.

(3) If the node is a half-leaf and can be merged with a leaf, combine the two nodes into one node as a leaf, and discard the other node. Proceed to step (5).

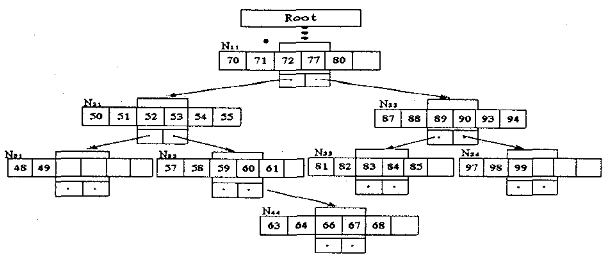
(4) If the current node (a leaf) is not empty due to deletion, then stop.

Else, discard the empty node, change the successor pointer to the appropriate node, and proceed to step (5) to rebalance the tree.

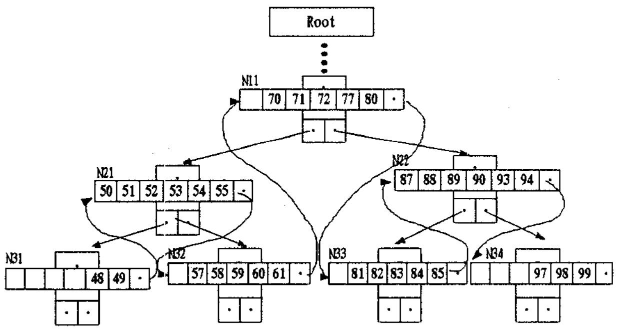
(5) After the node deletion, check the tree for balance by following the path from the leaf up to the root.

If the two subtrees of a node differ in depth by more than one level, then perform a rotation operation. Since a rotation at one node may create an imbalance for a node higher up in the tree, balance-checking for deletion must examine all of the nodes on the search path until a node of even balance is discovered.

**3.2.4. Operation on Range Query.** There are still some problems to be improved in T-tree. The processing for range queries in a T-tree, traverses some unnecessary nodes with no data items to search, due to using the inorder tree traversal. [Figure 7] shows an example of this problem. Suppose that a range query," search all items which is greater than or equal to 67, and is less than or equal to 82", should be processed in a given T-tree. This execution steps are as follows. The search starts at the root of the tree for a node which bounds the minimum element 67. After searching a node of N44, ittraverses to the successor node in the tree using inorder tree traversal, until it gets to the last node which bounds the maximum element 82. In this case, the traversal path for the example is " N44- N32 - N21 - N11 - N22 - N33 ". Here, the nodes N32, N21, and N22 do not have any data items to search in the query. [Figure 8] shows the example for the same query of [Figure 7] in a T*-tree. The execution steps are as follows. The search starts at the root of the tree for a node which bounds the minimum element 67. After searching a node of N44, it traverses directly to the successor node using successor pointer in current node, until it gets to the last node N33 which bounds the maximum element 82. In this case, the traversal path for this example is "N44 - N11 - N33". N44, N11, and N33 are only pure nodes which have data items to be searched in the query. Also these successor



[Figure 7] The Example of a Range Query($67 \leq x \leq 82$) on a T-tree



**[Figure 8] The Example of a Range Query(67≤x≤82) on a T*-tree**

pointers make it easier to traverse directly to the node, bounding the least upper bound for insertion or deletion. From any node in a T*-tree, all of the other nodes can be retrieved directly using these pointers in order. This property is useful and makes it easier to shorten the traversal path when using the T*-tree to perform a merge join[4]. All other properties of the T-tree are inherited to the T*-tree.

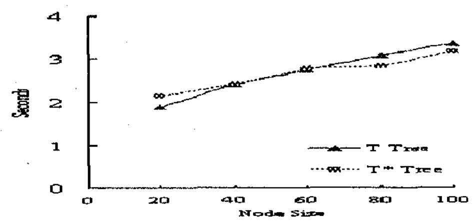
## 4. Performance Test Results

[Graph 1 thru 4] shows the performance test results for the T*-trees vs T-trees. These tests simulated the operations of a normal database management system so that two kinds of the index structures would be compared in a realistic environment. To supply the indices with data, a random number generator was used. All of the tests reported in this paper run on a SUN-SPARC-10 machine with 32 megabytes main memory, and each of algorithms for trees was implemented in C programming language. Each tree structure was tested for several aspects of index use; data insertion, deletion, search and sequential/range queries.

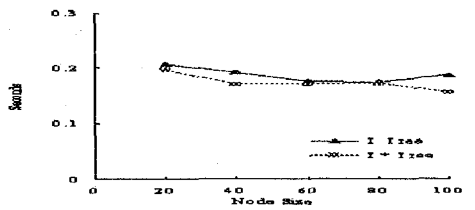
The tests were run in the following order; the unique data files were generated by the random number generator, then, each of the tree structures with varying node sizes was built for 10000 data insertion, searched for 4000 data items to measure the retrieval speed, performed test for deletion of 2000 data items and conducted ranged search for 100 data items. [Graph 1 thru 4] shows the test result for each operations with varying node size 20 to 100. The test results show almost the same execution time for the operations of data insertion, search and deletion with varying node sizes. In the case of range query, the T*-trees provided excellent performance against the T-tree and the execution time decreased almost 90 percent compared to T-tree.

## 5. Conclusion

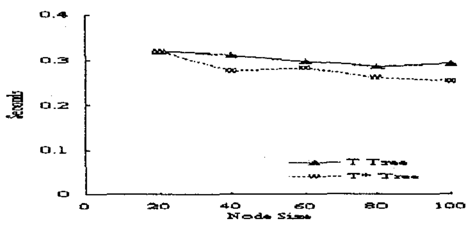
In this paper, we proposed a new indexing structure called T*-tree, to provide efficient processing of real time application and to improve several problems in the T-tree. T*-tree is a new structure for making rapid data access and for saving memory space under MMDBMS. The T*-tree indexing structure is basically redesigned from the T-tree, having an additional pointer, called successor pointer, which directly points to the successor node. This additional pointers incurs just a little overhead in memory space, because T*-tree has several data elements in a node with only one successor pointer attached. We have improved and described the pseudo-algorithms for the T*-tree operations and compared the time for certain operations to T-tree index structures. The data items in a T*-node is also in a sorted order like as T-node. But unlike in T-tree, the first data element is always inserted into the rightmost entry as a minimum value in a leaf node, while it is inserted into leftmost entry as a maximum value in a T-node. This enables direct access for data to pass down to GLB node due to insert overflows, and to borrow from GLB node due to deleted underflows using the additional pointers.



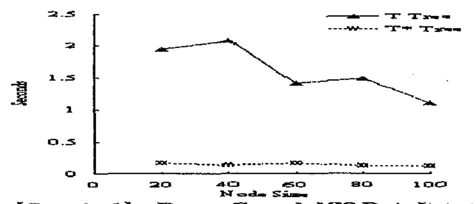
[ Graph 1] Insert 10,000 Items



[Graph 2] : Delete 2,000 Items



[Grahp 3] Search 4,000 Items



[Graph 4] Range Search 100 Data Items

Finally, T*-tree showed the better performance in sequential or range queries, and also in the operation for data insertion or deletion which cause the overflows or underflows. Therefore, we believe that the T*-tree is one of the more efficient indexing structures for real time applications under the MMDBMS environment.

## References

[1] Tobin J. Lehman, Michael J.Carey "A Study of Index Structures for Main Memory Database Management Systems", in Proceedings 12th Int. Conf. on Very Large Databases, Kyoto, Aug. 1986, pp.294-303

[2] Tobin J. Lehman, Eugene J. Shekita, Luis-Felipe Carera "An Evaluation of Starburst's Memory Resident Storage Component", IEEE Trans. on know, and Data Eng, vol. 4, december 1992.

[3] M. Leland and W. Roome, "The Sillcon Database Machine", Proc, 4th Int. Workshop on Database Machines, Grand Bahama Jsland, March 1985.

[4] Tobin J. Lehman, Michael J. Carey "Query Processing in Main Memory Database Management Systems"., Proc. ACM SIGMOD Conf., May 1986. pp239-250.

[5] David J. Randy H. Katz, Frank Olken, Leonard D. Shapiro, Micheal R. Stonebraker, David Wood "Implementation Techniques for Main Memory Database Systems", Int. Proc. of ACM SIGMOD, Boston 1984.

[6] A. Aho, J. Hopcroft and J. D. Ullman, "The Design and Analysis of Computer Algorithms", Addison-Wesley Publishing Company, 1974.

[7] D. Comer, "The Ubiquitous B-tree", Computing surveys 11, 2 (June 1979)

[8] Margarte H. Eich "Main Memory Database Reseaarch Directions" Technical Report 88-CAE-35

[9] H.V. Jagadish, Daniel Lieuwen, Rajeev Rastogi, Avi Silberschatz "Dali: A High Performance Main Memory Storage Mnager", Proc. of the 20th VLDB Conf. Santiago, chile 1994, pp48-59

[10] Kaist "FLASH:A Main Memory Storage System", KAIST Tech. Report, 94-257-7-1.

[11] Alfns Kemper, Guido Moerkotte, "Physical Object Management", Next-Generation Database Technology, chapter 9.

[12] Eugene J. Shekita, Michael Carey " A Performance Evaluation of Pointer-Based joins", Proc. ACM SIGMOD Int'l Conf., Atiantic city, Nj, May 1990, pp300-311

[13] Perlis, A. and Thornton, C., " Symbol Manipulation by Threaded Lists", CACM, Vol. 3, No. 4, April 1960, pp195-204

[14] Leonard D. Shapiro, "Join Processing in Database Systems with Large Memories", ACM Transactions on Database Systems, Vol. 11, No. 3, September,1986, pp239-264

[15] A. Amman, M. Hanrahan and R. Krishnamurthy, "Design of memory Resident DBMS," Proc. IEEE COMPCON, San Francisco, Feb 1985