A Study of Index Structures for Main Memory Database Management Systems

Tobin J. Lehman
Michael J. Carey

Computer Sciences Department
University of Wisconsin
Madison, WI 53706

ABSTRACT

One approach to achieving high performance in a database management system is to store the database in main memory rather than on disk. One can then design new data structures and algorithms oriented towards making efficient use of CPU cycles and memory space rather than minimizing disk accesses and using disk space efficiently. In this paper we present some results on index structures from an ongoing study of main memory database management systems. We propose a new index structure, the T Tree, and we compare it to existing index structures in a main memory database environment. Our results indicate that the T Tree provides good overall performance in main memory.

1. Introduction

It is projected that memory chip densities will continue their current trend of doubling every year for the foreseeable future, and, as a result, it is expected that main memory sizes of a gigabyte or more will be feasible and perhaps even common within the next decade [Fish86]. This large increase in the amount of main memory will have a profound impact on database management systems as it will be possible, in some cases, to store the entire database in main memory, removing disks from the path of normal query processing operations. (Disks will still be needed to store stable backup versions of the database, of course.) In addition to traditional database applications, there are a number of emerging applications for which main memory sizes will almost certainly be sufficient, applications that wish to be able to store and access relational data mostly because the relational model and its associated operations provide an attractive abstraction for their needs. Horwitz and Teitelbaum have proposed using relational storage for program information in language-based editors, as adding relations and relational operations to attribute grammars provides a nice mechanism for specifying and building such systems [Horw85]. Linton has also proposed the use of a database system as the basis for constructing program development environments [Lint84]. Snodgrass has shown that the relational model provides a good basis for the development of performance monitoring tools and their interfaces [Snod84]. Finally, Warren (and others) have addressed the relationship between Prolog and relational database systems [Warr81], and having efficient algorithms for relational operations in main memory could be useful for processing queries in future logic programming language implementations.

So far, there have been two approaches proposed for using large amounts of main memory in database systems. The first approach is to make 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. DeWitt et al [DeWi84], Shapiro [Shap86], and Elhardt et al [Elha84] have taken this approach in their work. Minimizing disk accesses still tends to be the primary performance goal for algorithm design when this approach is taken. The other major approach, the memory-resident database approach, is to use the large amount of memory as the main store for the database. This approach requires a redesign of the database management system — the algorithms and data structures for query processing, concurrency control, and recovery must all be restructured to stress the efficient use of CPU cycles and memory rather than disk accesses and disk storage. The designs proposed by Krishnamurthy et al [Amma85] and Leland et al [Lela85] have been based on this latter approach. In this approach, disk accesses are only an issue for crash recovery purposes, as recovery information must still reside on disk.

There are several arguments in favor of using existing systems and simply enlarging the buffer pool: no work is needed, since changing the buffer pool size simply involves changing a constant in the DBMS, and the speed of the overall system will increase since fewer disk accesses will be required. However, a buffer oriented system has a number of inherent handicaps. Every reference to the database must go through the buffer manager, causing the buffer manager to check its page list to see if the page is in memory, then perform the address calculation, and finally, if necessary, pin the page to guarantee that it is not swapped out prematurely. Thus, it seems that there is much to be gained by redesigning a database system to avoid contact with the disks and the buffer manager to the maximum extent possible.

Motivated by increasing memory sizes and these buffer-related arguments, we are studying the memory-resident database approach to main memory database management systems. We are evaluating both old and new database algorithms to determine which ones make the best use of CPU cycles and memory in a main memory database environment. Although we assume that there is a large amount of memory available, we are not willing to assume that it is infinite. Therefore, we judge data structures and algorithms according to both speed and storage efficiency criteria. The first phase of our study has addressed query processing issues. Although relations are memory resident, indices are still needed for fast access to the data. The initial part of our query processing study addressed index structures for main memory databases, and this is the focus of the rest of the paper.

The remainder of this paper is organized as follows: Section 2 describes the differences between main memory index structures and disk index structures. Section 3 enumerates a number of existing index structures and then introduces a new index structure, the T Tree, which was designed for use in main memory. Section 4 describes the index tests and gives the results of those tests. Section 5 suggests some improvements to the T Tree index structure, and, finally, Section 6 gives our conclusions.

This research was partially supported by an IBM Fellowship, an IBM Faculty Development Award, and National Science Foundation Grant Number DCR-8402818.

Tobin Lehman's new address is IBM Almaden Research Center, San Jose, California 95120

2. Disk Versus Main Memory

Index structures designed for main memory are different from those designed for disk-based systems. The primary goals for a disk-oriented index structure are to minimize the number of disk accesses and to minimize disk space. A main memory oriented index structure is contained in main memory, hence there are no disk accesses to minimize. Thus, the primary goals of a main memory index are to reduce overall computation time while using as little memory as possible. Since relations are memory resident, it is not necessary for a main memory index to store actual attribute values. Instead, pointers to tuples can be stored in their place, and these pointers can be used to extract the attribute values when needed. This has several advantages. First, a single tuple pointer provides the index with access to both the attribute value of a tuple and the tuple itself, reducing the size of the index. Second, this eliminates the complexity of dealing with long fields, variable length fields, and compression techniques in the index. Third, moving pointers will tend to be cheaper than moving the (usually longer) attribute values when updates necessitate index operations. Finally, since a single tuple pointer provides access to any field in the tuple, multi-attribute indices will need less in the way of special mechanisms.

3. Main Memory Index Structures

There are many data structures available for consideration as index structures. There are two main types: those that preserve some natural ordering in the data and those that randomize the data. The index structures being studied here are: arrays, AVL Trees, and B Trees, for the order-preserving class[1]; and Chained Bucket Hashing, Linear Hashing and Extendible Hashing, for the randomizing class[2]. We describe briefly each algorithm in turn. Then, having considered these algorithms, we will introduce a new algorithm called the T Tree. The T Tree is an order-preserving tree structure designed specifically for use in main memory.

3.1. Existing Index Structures

Arrays are used as index structures in IBM's OBE project [Amma85]. They use minimal space, providing that the size is known in advance or that growth is not a problem (e.g., that one can somehow use the underlying mapping hardware of virtual memory to make the array grow gracefully). The biggest drawback of the array is that data movement is *O(N)* for each update, so it appears impractical for anything but a read-only environment.

AVL Trees [Aho74] are used as indices in the AT&T Bell Laboratories Silicon Database Machine [Lela85]. The AVL Tree was designed as an internal memory data structure (Figure 1). It uses a binary tree search, which is very fast since the binary search is *intrinsic* to the tree structure (no arithmetic 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.

B Trees [Come79] are well suited for disk use since they are broad shallow trees and require few node accesses to retrieve a value (Figure 1). Most database systems use a variant of the B Tree, the B+ Tree, which keeps all of the actual data in the leaves of the tree. For main memory use, however, the B Tree is preferable to the B+ Tree because, in main memory, there is no advantage to keeping all of the data in the leaves — 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).

[1]Radix structures provide an excellent method for maintaining ordered data *for some special types of data* [Knut73, Will84], but since they are not general enough to be used for all data they are not considered here.

[2]There were several dynamic hashing algorithms to choose from. Extendible Hashing and Linear Hashing were chosen because of their popularity and the fact that most other methods are based on variants of these methods. Methods using spiral storage [Kjel84] or partial expansions [Lars80] require more data reorganization [Mull84], so, they would not do as well in a dynamic main memory environment.

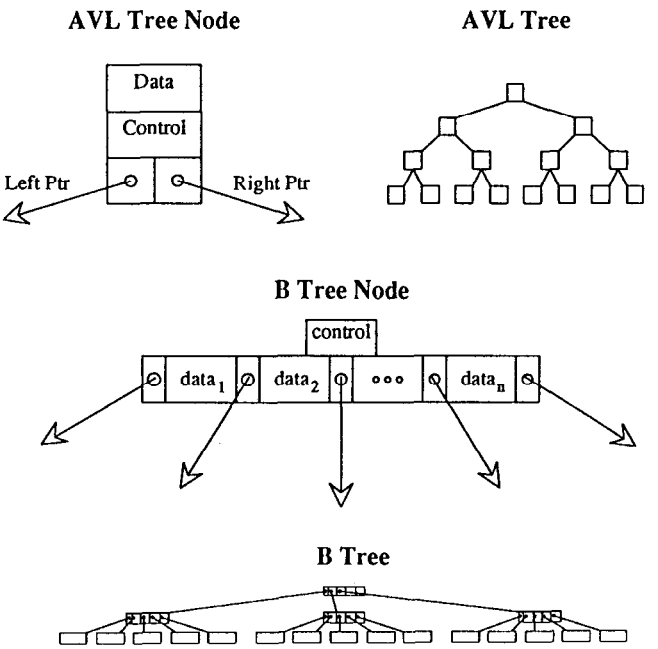


Figure 1— Tree Structured Indices

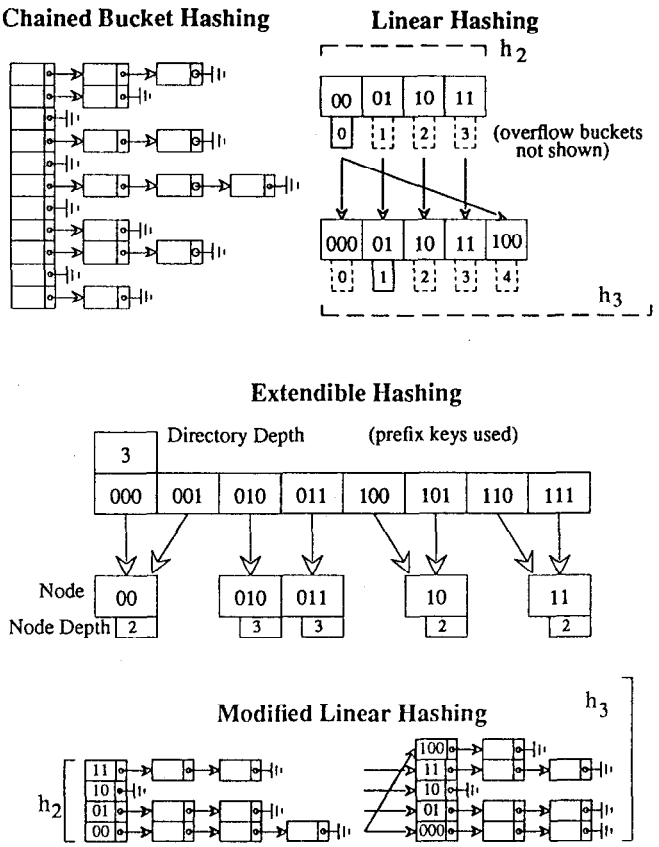


Figure 2 — Hashing-based Indices

Chained Bucket Hashing [Knut73] is a static structure used both in memory and on disk (Figure 2). It is very fast because it is a static structure — it never has to reorganize its data. But, this advantage is also its disadvantage; because it is static, it may have very poor behavior in a dynamic environment because the size of the hash table must be known or guessed at before the table is filled. If the estimated size is too small, then performance can be poor, and if the estimated size is too large, then much space is wasted. At best, there is some space wasted, since each data item has a pointer associated with it.[3]

Extendible Hashing [Fagi79] employs a dynamic hash table that grows with the data, so the table size does not need to be known in advance (Figure 2). A hash node contains several items and splits into two nodes when an overflow occurs. The directory grows in powers of two, doubling whenever a node overflows and has reached the maximum depth for a particular directory size. One problem with Extendible Hashing is that *any* node can cause the directory to split, so the directory can grow to be very large if the hash function is not sufficiently random.

Linear Hashing [Litw80] also uses a dynamic hash table, but it is quite different from Extendible Hashing (Figure 2). A Linear Hash table grows linearly as it splits nodes in predefined linear order — as opposed to Extendible Hashing, which splits the nodes that overflow. The decision to split a node and extend the directory in a controlled fashion can be based on criteria other than overflowing nodes, providing several advantages over the uncontrolled splitting of Extendible Hashing. First, the buckets can be ordered sequentially, allowing the the bucket address to be calculated from a base address — no directory is needed. Second, the event that triggers a node split can be based on storage utilization, keeping the storage cost constant for a given number of elements.

Modified Linear Hashing is oriented more towards main memory than the regular version mentioned above (Figure 2). (Litwin's description of Linear Hashing does not exclude this style of Linear Hashing, but his description is generally targeted more towards disk applications [Litw80].) By using larger contiguous nodes rather than a directory, normal Linear Hashing can waste space with empty nodes (when a node corresponding to a hash entry has no data items). Also, unless a clever scheme can be worked out with the underlying virtual memory mapping mechanism, the contiguous nodes have to be copied into a larger memory block when the directory grows. Modified Linear Hashing uses a directory much like Extendible Hashing, except that it grows linearly, and chained single-item nodes, allocated from a general memory pool. (Unlike Chained Bucket Hashing, there are typically many items per hash value, so multiple item nodes could indeed be used here to increase storage utilization.) The splitting criteria is based on performance, i.e. the average length of the hash chains, rather than storage utilization. Monitoring average hash chain length provides more direct control over the average search and update times than monitoring storage utilization.

3.2. The T Tree

The T Tree, a new data structure, evolved from AVL Trees and B Trees. The T Tree is a binary tree with many elements in a node. Figure 3 shows a T Tree and a node of a T Tree, called a T Node. Since the T Tree is a binary tree, it retains the intrinsic binary search nature of the AVL Tree. Also, because a T node contains many elements, the T Tree has the good update and storage characteristics of the B Tree. Data movement is required for insertion and deletion, but it is usually needed only within a single node. Rebalancing is done using rotations similar to those of the AVL Tree, but it is done much less often than in an AVL Tree due to the possibility of intra-node data movement.

[3] By grouping several items in a node, the data to pointer ratio could be reduced, but the extra data space might be wasted. Our tests showed that the optimal table size for a given number of elements has an average of only two items per hash value.

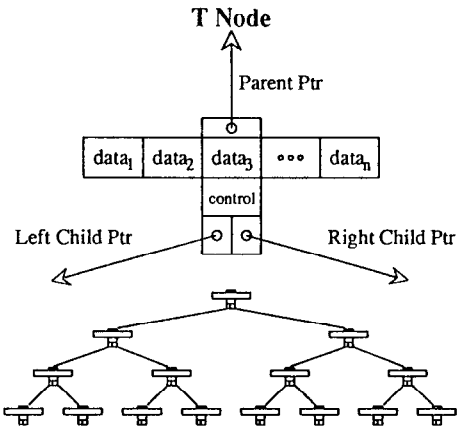


Figure 3 — The T Tree

To aid in our discussion of T Trees, we begin by introducing some helpful terminology. There are three different types of T-nodes. A T-node that has two subtrees is called an *internal node*. A T-node that has one NIL child pointer and one non-NIL child pointer is called a *half-leaf node*. A T-node that has two NIL child pointers is called a *leaf node*. For each internal node A, there is a corresponding leaf (or half-leaf) that holds the 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 *greatest lower bound* of the internal node A, and the successor value is called the *least upper bound* of A, as shown in Figure 4. For a node N and a value X, if X lies between the minimum element of N and the maximum element of N (inclusive), then we say that node N *bounds* the value X. Since the data in a T-node is kept in sorted order, its leftmost element is the smallest element in the node and its rightmost element is the largest.

Associated with a T Tree is a minimum count and a maximum count. Internal nodes keep their occupancy (*i.e.* the number of data items in the node) in this range. The minimum and maximum counts will usually differ by just a small amount, on the order of one or two items, which turns out to be enough to significantly reduce the need for tree rotations. With a mix of inserts and deletes, this little bit of extra room reduces the amount of data passed down to leaves due to insert overflows, and it also reduces the amount of data borrowed from leaves due to delete underflows. Thus, having flexibility in the occupancy of internal nodes allows storage utilization and insert/delete time to be traded off to some extent. Leaf nodes and half-leaf nodes have an occupancy ranging from zero to the maximum count.

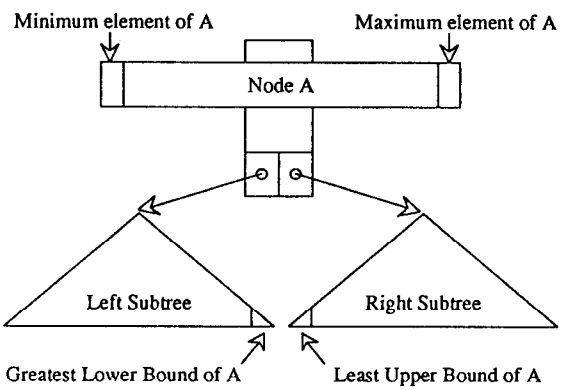


Figure 4 — The Bounds of a T-node

3.2.1. Search and Update Operations for T Trees

Search Algorithm

Searching in a T Tree is similar to searching in a binary tree. The main difference is that comparisons are made with the minimum and maximum values of the node rather than a single value as in a binary tree node. The algorithm works 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 subtree pointed to by the left-child pointer. Else, if the search value is greater than the maximum value of the node, then search down the subtree pointed to by the right-child pointer. Else, search the current node.

The search fails when a node is searched and the item is not found or when a node that bounds the search value cannot be found.

Insert Algorithm

The insert operation begins with a search to locate the bounding node. The new value is inserted into the bounding node and then the tree is checked for balance. If the T Tree is unbalanced as a result of the insertion (or deletion, for that matter), the appropriate rebalancing operation checks the nodes in the path between the root and the leaf where the insertion or deletion took place. Insertion works as follows, although we postpone our discussion of the rebalancing rotations until Section 3.2.2:

(1) Search for the bounding node.

(2) If a node is found, then check for room for another entry. If the insert value will fit, then insert it into this node and stop. Else, remove the minimum element from the node, insert the original insert value, and make the minimum element the new insert value. Proceed from here directly to the leaf containing the greatest lower bound for the node holding the original insert value. The minimum element (the new insert value) will be inserted into this leaf, becoming the new greatest lower bound value for the node holding the insert value.

(3) If the search exhausts the tree and no node bounds the insert value, then insert the value into the last node on the search path (which is a leaf or a half-leaf). If the insert value fits, then it becomes the new minimum or maximum value for the node. Otherwise, create a new leaf (so the insert value becomes the first element in the new leaf).

(4) If a new leaf was added, then check the tree for balance by following the path from the leaf to the root. For each node in the search path (going from leaf to root), if the two subtrees of a node differ in depth by more than one level, then a rotation must be performed (see Section 3.2.2). Once one rotation has been done, the tree is rebalanced and processing stops.

A design note is in order here: When an internal node overflows, and thus its minimum value is removed and passed down to a leaf, the insert into the leaf requires no data movement because the value becomes the leaf's rightmost entry. If instead the maximum value had been removed from the internal node, it would have to be inserted as the leftmost entry in the leaf, requiring intra-node data movement. Hence, removing the minimum value instead of the maximum value avoids this data movement.

Delete Algorithm

The deletion algorithm is similar to the insertion algorithm in the sense that the element to be deleted is searched for, the operation is performed, and then rebalancing is done if necessary. The algorithm works as follows:

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

(2) If the delete will not cause an underflow (*i.e.* if the node has more than the minimum allowable number of entries prior to the delete), then simply delete the value and stop. Else, if this is an internal node, then delete the value and borrow the greatest lower bound of this node from a leaf or half-leaf to bring this node's element count back up to the minimum. Else, this is a leaf or a half-leaf, so just delete the element. (Leaves are permitted to underflow, and half-leaves are handled in step (3).)

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

(4) If the current node (a leaf) is not empty, then stop. Else, free the node and proceed to step (5) to rebalance the tree.

(5) For every node along the path from the leaf up to the root, if the two subtrees of the node differ in height by more than one, perform a rotation operation (see Section 3.2.2). 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.

3.2.2. Rebalancing a T Tree

The rebalancing operations for a T Tree are similar to those for an AVL tree [Aho74]. A T Tree's balance is checked whenever a leaf is added or deleted, as indicated in the descriptions of the insertion and deletion algorithms. The search path is checked from the leaf to the root — for each node in the path, if the node's two subtrees differ in height by more than one level, a rotation operation is needed. In the case of an insertion, at most one rotation is needed to rebalance the tree, so processing stops after one rotation. In the case of a deletion, a rotation on one node may trigger an imbalance for a node higher up in the tree, so processing continues after a rotation until an evenly balanced node is found. Figure 5 shows the simple LL rotation and the more complex LR rotation for the case of an insert. These are two of the four types of rotations used to rebalance an AVL tree or a T Tree. The algorithms for the RR and RL rotations are symmetrical to the LL and LR rotations, respectively, so they are not shown.[4] The rebalancing rotations for deletion are

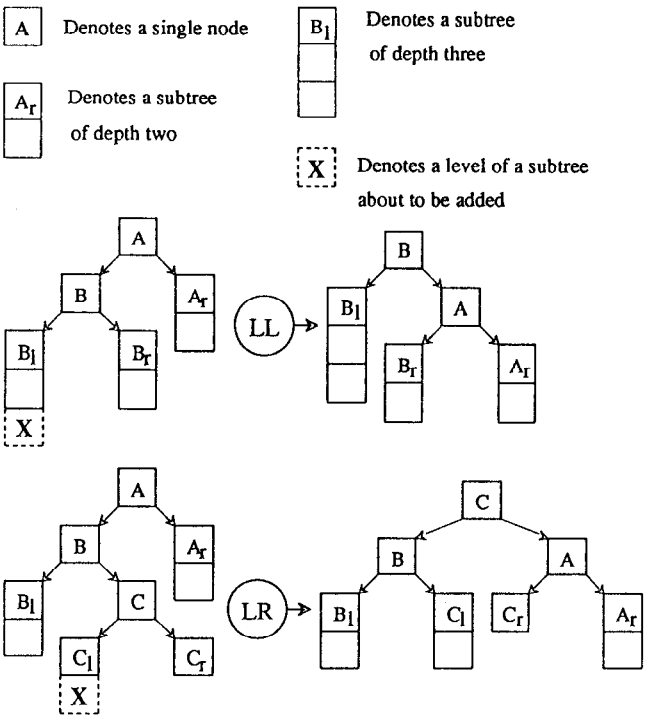


Figure 5 — T Tree Rebalancing Operations

[4] The names of the rotations (LL, RL, LR, and RR) are derived from the child of the node that causes the imbalance. In the LL rotation in Figure 5, the Left Left child of A is longer; in the LR rotation, the Left Right child is longer.

identical to rebalancing after insertion, except that the cause of the imbalance in the tree is that a subtree has grown shorter rather than longer.

The T Tree requires one special case rotation that the AVL tree does not have. When an LR or RL rotation is done and the node C is a leaf (and both nodes A and B are half-leaves), a regular rotation would move C into an internal node position, as shown in Figure 6. If C has only one item, which is always the case during an insert, then C would *never* get to hold more than a single item as an internal node (unless it is later rotated back into a leaf position), because items are always inserted *between* the low and upper bound of an internal node. Since this would be detrimental to storage utilization, a special rotation operation first moves values from B to C so that, after the rotation, C is a full internal node. This special case rotation is shown in Figure 6.

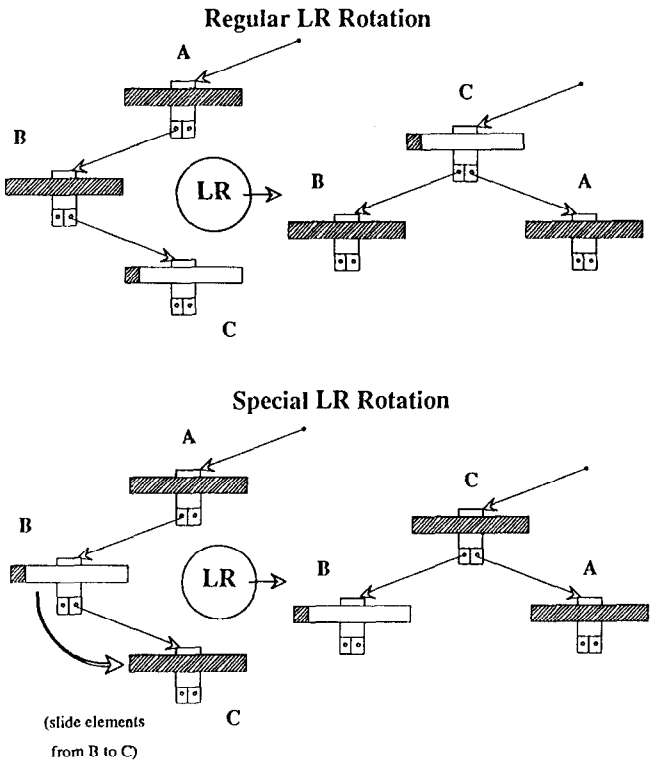


Figure 6 — Special T Tree Rebalancing Operations

3.2.3. Other Properties of the T Tree

T-nodes are equipped with child and parent pointers so that they can be traversed in either direction. From any node in a T Tree, all of the other nodes can be retrieved — in order. This property is useful when using the T Tree to perform a merge join [Lehm86b]. This property also allows the T Tree to store and retrieve duplicates with no extra work. Duplicate values (if allowed) are placed next to items with equal values — node boundaries are ignored. The update and rotation operations work as before. To retrieve a set of tuple pointers corresponding to a single value, one simply searches for the value, marks the spot where the first such item is found, then scans left, and then right from that spot.

4. Index Performance Tests

Each of the index structures described in Section 3 took part in the index performance study. The index tests simulated the operations of a normal database management system so that the index structures would be compared in a realistic environment. Each algorithm implementation was done in "main memory" style, that is, the indices held pointers to records (memory addresses) instead of keys or actual records. To supply the indices with data, a random number generator filled the test relation with unique 4 byte integers that were used as key values for the indices and, since the values could be read out of the relation, the search and delete operations were always done on existing elements, hence they were always successful. The chosen index size was 30,000 elements, because this was the largest number of elements that most of the implementations could support for the amount of memory available.[5]

4.1. Index Test Description

The tests were run in the following order: the data structures were built, then searched, then subjected to three query mixes, then searched with range queries, then scanned, and then half of the values were deleted. We describe each test in more detail below.

Insert 30,000 elements — To measure the insert cost of, 30,000 elements were inserted into each index structure. The inserts were each separate, as opposed to a special block insert. The index structures allowed unique values only, thus, each insert operation involved a search to ensure that the item was not already there.

Search for 30,000 random elements — To measure the retrieval speed of the indices, each index was searched for 30,000 different elements, with each element requiring a new search.

Query mix — Each index structure was tested in a "normal" update environment by performing a mix of inserts, searches, and deletes. Three query mixes were used: the first query mix was composed of 80 percent searches, 10 percent inserts and 10 percent deletes; the second was composed of 60 percent searches, 20 percent inserts and 20 percent deletes; and the third was composed of 40 percent searches, 30 percent inserts and 30 percent deletes. To keep the index structures at a constant size, the three operations were interspersed and the percentages of inserts and deletes were kept equal, thereby making the results easier to evaluate.

Query mix(1) — 24,000 searches, 3000 inserts, 3000 deletes
Query mix(2) — 18,000 searches, 6000 inserts, 6000 deletes
Query mix(3) — 12,000 searches, 9000 inserts, 9000 deletes

Range queries — Order-preserving data structures often need to supply a list of values corresponding to a given range. Three range queries were tested — each one testing different amounts of searching and scanning. The lower and upper bounds of a range known to be 10, 100, or 1000 elements apart were given to each index. The index would search for the lower bound value, then scan the elements in order until it found or passed the upper bound value. Note that these tests are inappropriate for the hashing algorithms, because they do not store values in logical order.

Range query 10: 30,000 queries, retrieving 10 elements per query
Range query 100: 3,000 queries, retrieving 100 elements per query
Range query 1000: 300 queries, retrieving 1000 elements per query

Sequential scan — To test the scanning speed of each data structure, each item in the index was read. For the array and tree data structures, the values were read in logical order. For the hashing structures, the values were read in physical order.

Delete 15,000 elements — To get a more realistic delete cost, only half of the index structure was deleted. By deleting only half of the index, the cost of a single delete can be calculated as roughly 1/15,000th of the total time, because the data structures were not greatly affected by having half of their elements removed. A T Tree, for example, would become only one level smaller, and a B Tree would probably not even change height. Removing all of the elements would have given a false impression of the delete cost since the cost per item deleted would be much less for the second half of the elements than the first half.

Storage costs — The storage cost of each of the data structures after building the index with thirty thousand elements is sometimes different from the storage cost after the query mixes, even though the number of elements is the same, so each storage cost value was collected. The storage cost represents the total number of bytes needed

[5] Extendible Hashing still ran out of memory when run with the smallest node sizes (e.g., 2, 4 and 6 items per node). This is due, in part, to an inefficient storage allocator and, in part, to the algorithm itself (as will be explained more fully in Section 4.4).

to store the given number of elements. The storage cost computations assume that byte (as opposed to word) alignment is permissible, and that the smallest unit of allocation is a byte. (The pointer size for both data and nodes is four bytes.)

4.2. Reducing the Number of Test Parameters

Many of the index structures have several variable parameters and, in order to keep the number of tests reasonable, some of these parameters had to be held constant. Preliminary tests on the index structures gave us some indications of reasonable values for the parameters. For each structure, the number of variable parameters was reduced to one, node size (except for Modified Linear Hashing, where the average hash chain length was varied). The restriction to one similar variable allowed the execution times of the index structures to be compared side by side in the same graphs. A discussion of the parameter reduction process follows.

The B Tree has a variable node size, a variable leaf size, and two possible node/leaf search methods: linear and binary search. Preliminary tests showed that varying the leaf and node sizes separately did not significantly affect the results, so they were varied together at the same size. Linear search on small nodes (4 and 6 elements) was 5 percent faster than the best binary search (on any size node), but small nodes cause the tree to use much more space. Since storage and performance are both important, the binary search was used. The T Tree has a variable node size and two possible node-searching methods: linear search and binary search. For the same reasons as for the B Tree, binary search was used for the tests.

Extendible Hashing has a variable node size and three different possible node search techniques: linear search, binary search, and hash search. A hash search in the node would speed up the search time, but it would introduce problems in collision resolution. Separate chaining would waste at least half of the node's space on pointers, and open addressing would cause difficulties with delete by wasting either time or space to fill in holes left by deleted items [Knut73]. A binary search would have required keeping the node sorted, and therefore would have added data movement cost for insertion and deletion. Linear search seemed to be the simplest and quickest method, so this was used for the tests.

Linear Hashing has a variable node size, a variable overflow bucket size, a variable storage utilization, and three possible node/bucket search techniques: linear search, binary search and hash search. The overflow bucket size could range from a fraction of the node size to several times the node size. The storage, insert and delete costs were found to be better for larger buckets, while the search and query mix costs were better for small buckets. An overflow bucket that was half the size of the node gave reasonable overall performance. The storage utilization (the number of data bytes used divided by the total number of data bytes currently available) could range anywhere from a few percent to close to 100 percent. It is used to trigger a change in directory size — when the storage utilization goes up above a threshold value (usually because of an excess of overflow buckets), then the directory expands. Similarly, when the storage utilization falls below the threshold value (due to too many empty data slots), the directory shrinks. A lower storage utilization threshold parameter value implies that less time needs to be spent on data reorganization, since more space can be wasted; less reorganization cost means higher performance. A value of 70 percent seemed to give the best overall performance / storage cost ratio. Lastly, for the same reasons stated for extendible hashing, linear search was used to search the nodes and the overflow buckets.

Chained Bucket Hashing has only one parameter — table size. The preliminary tests showed that a table size equal to half the number of elements gives the best storage cost / performance ratio. Modified Linear Hashing is very similar to Chain-bucket Hashing, except that its hash table is dynamic. It uses its only parameter, the average length of the hash chains, to determine when to split the next hash chain. When the number of elements increases to the point that the average chain length value (size of table / number of data elements) is greater than the threshold value, the directory expands, and when the value is less then the threshold value, the directory shrinks. The final set of the values chosen for the tests appear in Table 1.

Data Structure	*Parameters*
Array	no parameters
AVL Tree	no parameters
B Tree	binary search, variable node size
T Tree	binary search, minimum node size set at two less than the maximum node size, variable (maximum) node size
Chained Bucket Hash	table size fixed at 50% of element count (at 15,000)
Extendible Hashing	linear search, variable node size
Linear Hashing	linear search, overflow bucket size of half the node size, 70% storage utilization factor, variable node size
Mod. Linear Hashing	linear search, variable average hash chain length

Table 1 — Parameters Chosen for the Tests

4.3. Implementation Details

Each algorithm was implemented in the C programming language and run in single-user mode on a VAX 11/750 with 2 megabytes of real memory (with no virtual memory involved). The timing measurements were taken using "getrusage", a timing facility available in BSD UNIX 4.2. The execution times reflect that of a main memory computer with a processing power of about 0.5 MIPS. Some details about the implementation of the algorithms that might help the reader to understand the results more fully follow:

- A standard hash function was used in all of the hashing implementations.

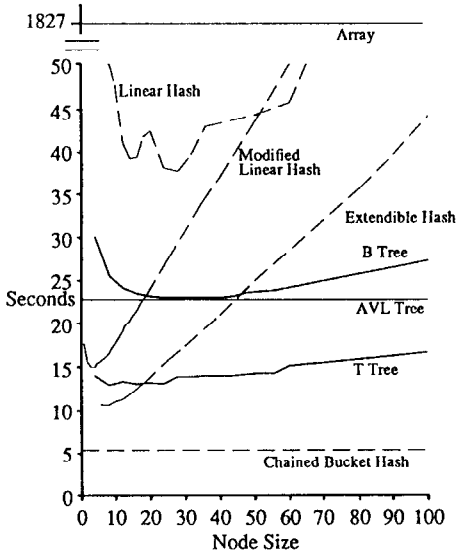
  (key * P) *mod* table size (where P is a large prime)

  In the cases where the table size was a power of two, the *mod* operation was done by stripping off bits with a bit mask operation.
- Since there was no virtual memory mechanism available, the Linear and Modified Linear Hashing directories had to be reallocated from memory when the table size grew or shrunk. To amortize the cost of this reallocation over several increases in directory size, new directories were allocated with an additional 1000 bytes for future growth. The memory for the array, however, was constant, being allocated once for the duration of the tests.
- Linear Hashing used an extra array of pointers to point to the last overflow bucket in each chain. This helped the insertion and deletion performance while sacrificing only a small amount in storage. This array is figured into the the storage costs in Graph 7 in the test results.

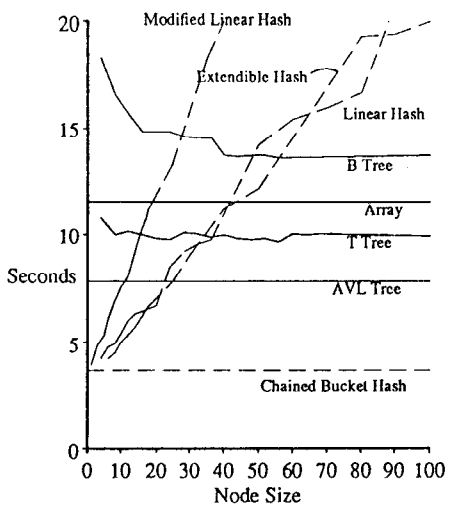
4.4. The Test Results

Graphs 1 through 6 show the execution times of the index structures for a representative subset of the tests, while Graph 7 shows the storage costs. The graphs use solid lines to represent the order-preserving structures and dashed lines to represent the hashing structures. Note that the different node sizes on the X-axis represent the average chain length for Modified Linear Hashing rather than node size — Modified Linear Hashing always used single element nodes. Also note that the array index structure had large execution times in all of the tests involving updates because of its *O(N)* data movement. The array is therefore omitted from further discussion involving updates.[6] The reader should keep in mind that the lowest point (time or storage) in each graph line is more important than the overall shape of the line. We are interested in the one node size that can support the fastest execution times while requiring the least amount of storage.

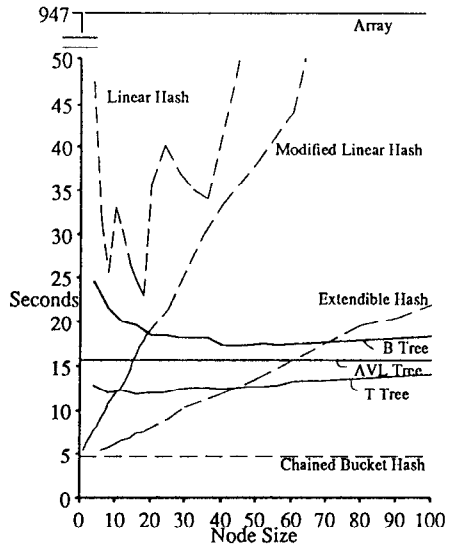
[6] We wish to point out, however, that if the array were used as an index, a practical way of *building* it would be to do a bulk insert into the array as a heap and then quicksort it afterwards. Tests from a main memory query processing paper [Lehm86b] show that an array index can be filled and sorted in approximately half the time it takes to build a T Tree (with node size 30).



Graph 1 - Index Insertion



Graph 2 - Index Search



Graph 3 - Query Mix of 60% Searches

INSERT 30,000 ELEMENTS — The results of this test are shown in Graph 1. The cost of a series of inserts depends on the search time and various operations particular to the individual data structure, such as the amount of data moved, the number of hash function calls, the number of tree rebalance operations, and the number of memory allocation operations. Looking at the AVL Tree search time versus the B Tree search time (Graph 2), it would seem that the AVL Tree should have the better insert time, but it has about the same insert time for B Tree Node sizes of 20 to 40 because it makes more memory allocation calls (one per data item) and uses a costly rotation operation to rebalance the tree after insertions. Inserting a value into the B Tree, on the other hand, requires fewer memory allocation calls, some intra-node data movement and, much less frequently, some inter-node data movement (when a node splits into two nodes).[7] The T Tree has a search time close to that of the AVL Tree, but it also has update characteristics similar to that of the B Tree; the T Tree makes fewer memory allocation calls than the AVL Tree and usually relies on intra-node data movement rather than tree rotations to keep the tree balanced. Hence, the T Tree has the best insertion times of the order-preserving index structures.

In each hashing structure, the search cost is independent of the number of elements in the table, as there is a fixed cost to jump to the right node and then search the set of elements at that node. Also, hashing structures require little data movement because the new item is added to the end of the heap of elements in the node. All four hashing methods have approximately the same search cost for small nodes (see Graph 2), and they all insert a data item in about the same way (by appending to the end of the list), yet their insert costs are very different. The difference between the hashing schemes is in the amount of work needed to resize the directory. Linear Hashing splits nodes in order, possibly splitting nodes that are not full. The node splitting criteria is based on both the number and the distribution of elements in the hash table.[8] This means that table growth (node splitting) is allowed only when the main nodes are close to full, so a nonuniform distribution causes some of the main nodes to remain underfilled while some of the other nodes acquire long overflow chains (and, correspondingly, long search times) due to the lack of reorganization. As shown in Graph 1, different node sizes yield different distributions, which have a direct affect on the search portion of the insert cost. Modified Linear Hashing, on the other hand, uses a node splitting criteria that is based solely on the total number of elements in the table, so the average length of the overflow chains is better controlled. It still uses the extra data reorganization of the Linear Hashing algorithm, but its more closely monitored overflow chain lengths result in a significantly decreased insert cost. Finally, Extendible Hashing requires that a node splits only when it overflows, so it does the least amount of reorganizing of data and thus has the fastest index insertion time of the dynamic hashing methods.

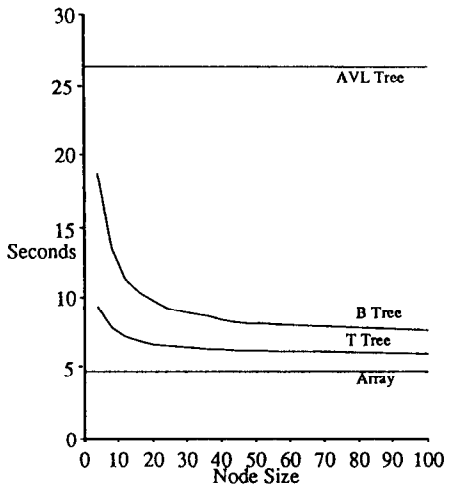
SEARCH — Graph 2 shows the search costs. The array uses a pure binary search. The overhead of the arithmetic calculation and movement of pointers is noticeable when compared to the "hardwired" binary search of a binary tree. In contrast, the AVL Tree needs no arithmetic calculations, as it just does one compare and then follows a pointer. The T Tree does the majority of its search in a manner similar to that of the AVL Tree, then, when it locates the correct node, it switches to a binary search of that node. Thus, the search cost of the T Tree search is slightly more than the AVL Tree search cost, as some time is lost in binary searching the final node. The B Tree search time is the worst of the four order-preserving structures, because it requires several binary searches, one for each node in the search path.

The hashing schemes have a fixed cost for the hash function computation plus the cost of a linear search of the node and any associated overflow buckets. For the smallest node sizes, all four hashing methods are basically equivalent. The differences lie in the search times as the nodes get larger. Linear Hashing and Extendible Hashing are just about the same, as they both search multiple-item nodes. Modified Linear Hashing searches a single-item node linked list, so each data reference requires traversing a pointer. This overhead is noticeable when the chain becomes long. (Recall that "Node Size" is really average chain length for Modified Linear Hashing.)

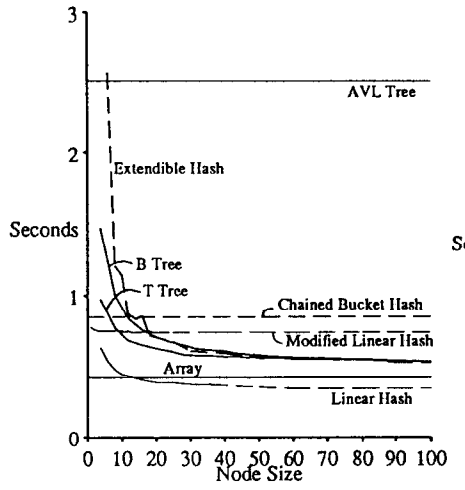
QUERY MIXES — Graph 3 shows the "main event" of the query tests. This test is most important, as it shows the index structures in a normal working environment. The query mix of 60 percent searches, 20 percent inserts and 20 percent deletes was representative of the three query mix graphs, so it is the only one shown; index structures with faster search times did somewhat better in the 80 percent search query mix and those structures with better update characteristics did somewhat better in the 40 percent search query mix. The T Tree performs better than the AVL Tree and the B Tree here because of its better combined search / update capability. The AVL tree is faster than the B Tree because it is able to search faster than the B Tree, but the execution times are fairly close because of the B Tree's better update capability. For the smallest node sizes, Modified Linear Hashing, Extendible Hashing, and Chained Bucket Hashing are all basically equivalent. They have similar search cost, and when the need to resize the directory is not present, they all have the same update cost. Linear Hashing, on the other hand, still reorganized its data in an attempt to maintain a particular storage

[7] Memory allocation and tree rotation costs exceeds this data movement cost for the node sizes used here due to the existence of the MOVC3 block move instruction on the VAX 11/750. This move instruction executes much faster than a hand optimized assembler routine of similar function.

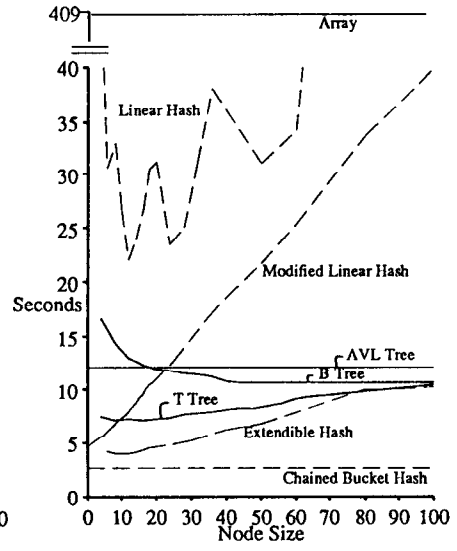
[8] This is referred to as *load control* in [Litw80], and it is based on the ratio of data elements to space allocated.



Graph 4 - Range Query



Graph 5 - Scan



Graph 6 - Delete

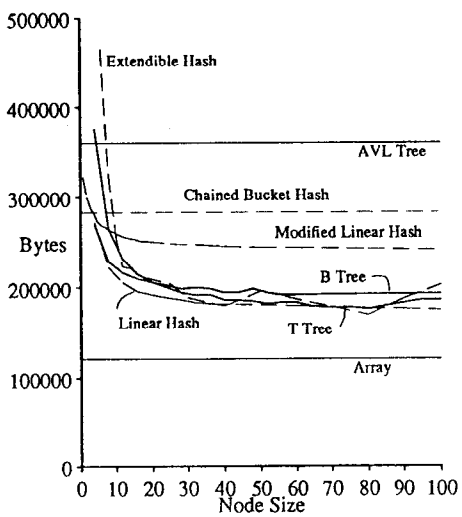
utilization factor, and it had a slower query mix execution time as a result.

RANGE QUERIES — The results of the range query tests are shown in Graph 4. The range query returning 100 elements for each of 3,000 queries was representative of the range query tests, so it is the only one shown here. A range query is composed of two parts: the search for the lower bound value, and then the scan through the data structure until the upper bound value of the range is found or surpassed. When few elements per query are returned, the range query times are similar to the search times. As the number of elements returned per query increases, the index structure with the best scanning speed becomes dominant. The array has the best scanning speed, as its elements are contiguous, while the T Tree is close because it has many logically contiguous elements in each node. (This is only true for leaf nodes in the case of the B Tree.) The slower search speed of the B Tree causes it to be third, and the slow scanning speed of the AVL Tree (since each data reference requires traversing a pointer) makes it the slowest of the four. Note that the hash tables do not preserve a logical ordering of the values, so they are not used for range queries.

SEQUENTIAL SCAN — The results of the sequential scan tests are given in Graph 5. For most of the structures a scan is fast, because there are segments of contiguous elements that can be traversed quickly. The main exception is the AVL tree, because the cost of walking a binary tree in order is exacerbated by the fact that every data item is in a different node. In general, those index structures using linked lists had a slower scan time than those structures that had segments of contiguous elements. (There is one anomaly here in that Linear Hashing seems to have beaten the array. The difference in times is only a tenth of a second however, and we attribute it to the error margin in the timing facility.)

DELETE — Graph 6 shows the execution times of the index structures for deleting half of each data structure. Chained Bucket Hashing needs to do little work to remove an item from the hash table, so it has the fasted execution time once again. Extendible Hashing rarely decreases its hash table, so delete is fast, but space may be wasted by its large directory and many partially filled nodes. At the shortest possible average chain length, Modified Linear Hashing is almost as fast as Chained Bucket Hashing; the difference in the two times is the amount of time devoted to reducing the size of the Modified Linear Hashing directory. The T Tree is faster than the AVL Tree and B Tree, as before, while the B Tree is a little faster than the AVL Tree because less work is required to keep it balanced.

STORAGE COST — The storage costs for Linear Hashing, Extendible Hashing, B Trees and T Trees were 5 percent less just after they were created than their storage cost after the query mixes. (The storage costs for the index structures after the query mixes are shown in Graph 7). The array uses the minimum amount of storage, so we discuss the storage costs of the other algorithms as a ratio of their storage cost to the array storage cost. First, we consider the fixed values: the AVL Tree storage factor is 3 because of the two node pointers it needs for each data item, and Chained Bucket Hashing has a storage factor of 2.3 because it has one pointer for each data item and part of the table remains unused (the hash function is not *perfectly* uniform). Modified Linear Hashing is similar to Chained Bucket Hashing for average an hash chain length of 2, but, as its hash chains grow longer, the number of empty slots in the table decreases and eventually the table becomes completely full. Finally, Linear Hashing, B Trees, Extendible Hashing and T Trees all had about equal storage factors of 1.5 for medium to large size nodes. Extendible Hashing tends to use the largest amount of storage for small nodes (2,4 and 6). This is because a small node size increases the probability that some nodes will get more values than others, causing the directory to double repeatedly and thus use large amounts of storage. As its nodes get larger, the probability of this happening becomes lower.



Graph 7 - Index storage costs

4.5. Test Observations

An important thing to notice about the hash-based indices is that, while the Extendible Hashing and Modified Linear Hashing had very good performance for small nodes, they also had high storage costs for small nodes. (However, the storage utilization for Modified Linear Hashing can probably be improved by using multiple-item nodes, thereby reducing the pointer to data item ratio). Extendible Hashing has another problem — storing duplicate values. If a page were to overflow with duplicate values, the directory could grow infinitely large and still not be able to resolve the problem. There is no normal overflow method in Extendible Hashing as there is in Linear Hashing. If duplicates were allowed, then nodes would need a special overflow pointer for duplicates, while processing regular values in the normal fashion. As for the other two hash-based methods: Chained Bucket Hashing has fast execution times, but it has fairly high storage costs, and it is only a static structure; and finally, Linear Hashing is just too slow to use in main memory.

Looking at the order-preserving index structures, AVL Trees have good search execution times and fair update execution times, but they have high storage costs. Arrays have good search and scan execution times and low storage costs, but any update activity at all causes it to have execution times *orders of magnitude* higher than the other index structures. AVL Trees and arrays do not have sufficiently good performance / storage characteristics for consideration as main memory indices. T Trees and B Trees do not have the storage problems of dynamic hashing methods; they have low storage costs for those node sizes that lead to good performance. The T Tree seems to be the best of choice for an order-preserving index structure, as it performs well in all of the tests.

4.6. Comparing Apples and Oranges

A major problem with this type of comparison is the issue of fairness. Though we did our best to code each index algorithm equally well, the results are close in some cases, and it is possible that constant factors could come into play and alter the results. Also, some algorithms may have had small advantages because of the test environment. For example, the data used for the tests was generated by a random number generator, so the hashing schemes may have been aided by having a uniform distribution of key values. Also, because unique key values were used and searches and deletes were always successful, those index structures using linear searches (i.e., the hashing structures) needed to search only half of the list, on the average, to find the desired element.

To ensure that the tests were *coded* as fairly as possible, the code was instrumented with counters to record the number of occurrences of various operations, such as the number of data compares, the amount of data movement, the number of hash calls, the number of tree rotations, the amount of memory allocated and freed, and the number of node searches. (These counters did not affect the execution times because they were compiled out of the code when the timing tests were run.) Using these counters, it was possible to observe the algorithms in action, making sure that the expected amount of work was being done. We plan to use this technology-independent information in the future to analyze the algorithms and determine their merits without the interference of machine-related dependencies.

5. Future Improvements

Some improvements and variations on the T Tree algorithms have been discovered since the index experiments were conducted. We briefly mention them here, and the reader is referred to [Lehm86a] for more detail.

(1) The search algorithm can be changed so that it always performs one compare per node rather than two (the existing worst case). This would reduce the number of compares from ($Log_2 N + 1/2 Log_2 N/K$) to a true $Log_2 N$. (N is the number of elements in the tree and K is the number of elements in a node.) Although this difference is not significant when the number of compares are simple, it results in significant gains when the compares are expensive.

The modified search algorithm compares the search key with only the minimum element in a node and uses that result to decide which subtree to search. If the right subtree is chosen, the current node is marked for future consideration because the elements in that node are still in the active search space. When the search reaches a leaf, the last marked node is searched with a binary search.

(2) A new type of T Tree, the T+ Tree, has been suggested. It keeps all of its data in leaf nodes, using the internal nodes (an AVL Tree) for guidance into chained multi-item leaves. The similarities to B+ Trees are obvious. The T+ Tree would have search and update performance similar to the T Tree, but the chained leaves would be easier to scan (using a simple linked list rather than a tree traversal) and possibly easier to maintain.

6. Conclusions

In this paper we introduced a new main memory index structure, the T Tree. We compared the T Tree structure against AVL Trees, simple arrays, B Trees, Chained Bucket Hashing, Extendible Hashing, Linear Hashing and Modified Linear Hashing. Our results indicate that a mix of two different index structures provides the best overall storage and performance. For unordered data, Modified Linear Hashing should give excellent performance for exact match queries. When used as a temporary structure where the size of the index is known in advance, the table size can be chosen initially to fit the application, thereby removing the reorganization overhead and allowing Modified Linear Hashing to behave like Chained Bucket Hashing. For ordered data, the T Tree provides excellent overall performance for a mix of searches, inserts, and deletes, and it does so at a relatively low cost in storage space. We plan to use the T Tree as the primary ordered access method for our prototype implementation of a main memory database management system, and we plan to use Modified Linear Hashing as the primary unordered access method.

7. Acknowledgements

Our thanks to Udi Manber for pointing out an optimization to the T Tree search algorithm and the possibility of T+ Trees. Also, the NSF-sponsored Crystal multicomputer project at the University of Wisconsin provided the many VAX 11/750 CPU-hours that were required for the index structure performance study.

8. References

[Aho74] A. Aho, J. Hopcroft and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley Publishing Company, 1974.

[Amma85] A. Ammann, M. Hanrahan and R. Krishnamurthy, Design of a Memory Resident DBMS, *Proc. IEEE COMPCON*, San Francisco, February 1985.

[Come79] D. Comer, The Ubiquitous B-Tree, *Computing Surveys 11*,2 (June 1979).

[DeWi84] D. J. DeWitt, R. Katz, F. Olken, L. Shapiro, M. Stonebraker and D. Wood, Implementation Techniques for Main Memory Database Systems, *Proc. ACM SIGMOD Conf.*, June 1984, 1-8.

[Elha84] K. Elhardt and R. Bayer, A Database Cache for High Performance and Fast Restart in Database Systems, *ACM Trans. on Database Systems 9*,4 (December 1984), 503-526.

[Fagi79] R. Fagin, J. Nievergelt, N. Pippenger and H. R. Strong, Extendible Hashing : A fast access method for dynamic files, *ACM Trans. on Database Systems 4*,3 (Sept. 1979), 315-344.

[Fish86] M. Fishetti, Technology '86: Solid State, *IEEE Spectrum 23*,1 (January 1986).

[Horw85] S. Horwitz and T. Teitelbaum, Relations and Attributes: A Symbiotic Basis for Editing Environments, *Proc. of the ACM SIGPLAN Conf. on Language Issues in Programming Environments*, Seattle, WA, June 1985.

[Kjel84] P. Kjellberg and T. Zahle, Cascade Hashing, *Proc. 10th Conf. Very Large Data Bases*, Singapore, August 1984, 481-492.

[Knut73] D. Knuth, *The Art of Computer Programming*, Addison-Wesley, Reading, Mass., 1973.

[Lars80] P. Larson, Linear Hashing with Partial Expansions, *Proc. 6th Conf. Very Large Data Bases*, Montreal, Canada, October 1980, 224-231.

[Lehm86a] T. Lehman, Design and Performance Evaluation of a Main Memory Relational Database System, Ph.D. Dissertation (University of Wisconsin-Madison), August 1986. (in progress).

[Lehm86b] T. Lehman and M. Carey, Query Processing in Main Memory Database Management Systems, *Proc. ACM SIGMOD Conf.*, May 1986.

[Lela85] M. Leland and W. Roome, The Silicon Database Machine, *Proc. 4th Int. Workshop on Database Machines*, Grand Bahama Island, March 1985.

[Lint84] M. Linton, Implementing Relational Views of Programs, *Proc. of the ACM SIGSOFT/SIGPLAN Software Eng. Symp. on Practical Software Development Environments*, Pittsburgh, PA, April 1984.

[Litw80] W. Litwin, Linear Hashing : A New Tool For File and Table Addressing, *Proc. 6th Conf. Very Large Data Bases*, Montreal, Canada, October 1980.

[Mull84] J. Mullin, Unified Dynamic Hashing, *Proc. 10th Conf. Very Large Data Bases*, Singapore, August 1984, 473-480.

[Shap86] L. D. Shapiro, Join Processing in Database Systems with Large Main Memories, *ACM Trans. on Database Systems*, 1986. (to appear).

[Snod84] R. Snodgrass, Monitoring in a Software Development Environment: A Relational Approach, *Proc. of the ACM SIGSOFT/SIGPLAN Software Eng. Symp. on Practical Software Development Environments*, Pittsburgh, PA, April 1984.

[Warr81] D. H. D. Warren, Efficient Processing of Interactive Relational Database Queries Expressed in Logic, *Proc. 7th Conf. Very Large Data Bases*, Cannes, Fance, September, 1981.

[Will84] D. E. Willard, New TRIE Data Structures Which Support Very Fast Search Operations, *Journal of Computer and Systems Sciences 28*,3 (June 1984), 379-394.