Query Processing in Main Memory Database Management Systems

Tobin J Lehman
Michael J Carey

Computer Sciences Department
University of Wisconsin
Madison, WI 53706

ABSTRACT

Most previous work in the area of main memory database systems has focused on the problem of developing query processing techniques that work well with a very large buffer pool In this paper, we address query processing issues for *memory resident* relational databases, an environment with a very different set of costs and priorities We present an architecture for a main memory DBMS, discussing the ways in which a memory resident database differs from a disk-based database We then address the problem of processing relational queries in this architecture, considering alternative algorithms for selection, projection, and join operations and studying their performance We show that a new index structure, the T Tree, works well for selection and join processing in memory resident databases We also show that hashing methods work well for processing projections and joins, and that an old join method, sort-merge, still has a place in main memory

1 Introduction

Today, medium to high-end computer systems typically have memory capacities in the range of 16 to 128 megabytes, and it is projected that chip densities will continue their current trend of doubling every year for the foreseeable future [Fis86] As a result, it is expected that main memory sizes of a gigabyte or more will be feasible and perhaps even fairly common within the next decade Some researchers believe that many applications with memory requirements which currently exceed those of today's technology will thus become memory resident applications in the not-too-distant future [GLV83], and the database systems area is certain to be affected in some way by these trends Previous studies of how large amounts of memory will affect the design of database management systems have focused almost entirely on how to make use of a large buffer pool [DKO84, DeG85, ElB84, Sha86]

With memory sizes growing as they are, it is quite likely that databases, at least for some applications, will eventually fit entirely in main memory For those applications whose storage requirements continue to exceed memory capacities, there may still be often-referenced relations that will fit in memory, in which case it may pay to partition the database into memory resident and disk resident portions and then use memory-specific techniques for the memory resident portion (much like IMS Fastpath and IMS [Dat81]) 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 [HoT85] Linton has also proposed the use of a database system as the basis for constructing program development environments [Lin84] Snodgrass has shown that the relational model provides a good basis for the development of performance monitoring tools and their interfaces [Sno84] Finally, Warren (and others) have addressed the relationship between Prolog and relational database systems [War81], and having efficient algorithms for relational operations in main memory could be useful for processing queries in future logic programming language implementations

Motivated by these considerations, we are addressing the question of how to manage large *memory resident* relational databases Whereas traditional database algorithms are usually designed to minimize disk traffic, a main memory database system must employ algorithms that are driven by other cost factors such as the number of data comparisons and the amount of data movement. We are studying these issues, evaluating both old and new algorithms to determine which ones make the best use of both CPU cycles and memory (Note that while memory can be expected to be large, it will never be free) We have focused mostly on query processing issues to date, but we also plan to examine concurrency control and recovery issues in our research — main memory databases will still be multi-user systems, and many applications will require their data to be stored safely on disk as well as in main memory

The remainder of this paper is organized as follows Section 2 describes our main memory DBMS architecture, pointing out ways in which the organization of main memory databases can profitably differ from disk-based databases Sections 3 presents our work on algorithms for implementing selection, projection, and join operations Both algorithms and performance results are given for each of these operations Finally, Section 4 presents our conclusions and discusses their impact on query optimization

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

2 Main Memory DBMS Architecture

In this section, we present the design of a main memory database management system (MM-DBMS) that we are building as part of a research project at the University of Wisconsin-Madison The key aspects of the design are the structure of relations, indices, and temporary lists (for holding query results and temporary relations) Ideas for approaching the problems of concurrency control and recovery are in the development stages The design is presented under the assumption that the entire database resides in main memory, ignoring (for now) the case of a partitioned database

2 1 Relations

Every relation in the MM-DBMS will be broken up into partitions, a partition is a unit of recovery that is larger than a typical disk page, probably on the order of one or two disk tracks In order to allow more freedom of design of these partitions, the relations will not be allowed to be traversed directly, so all access to a relation is through an index (Note that this requires all relations to have at least one index) Although physical contiguity is not a major performance issue in main memory (indeed, the tuples of a relation *could* be scattered across all of memory), keeping the tuples grouped together in a partition aids in space management and recovery, as well as being more efficient in a multi-level cache environment (In a single-level cache, cache block sizes are typically smaller than the size of a tuple, but in a multi-level cache where there are several cache block sizes, the larger sized cache blocks could hold most or all of a partition)

The tuples in a partition will be referred to directly by memory addresses, so tuples must not change locations once they have been entered into the database For a variable-length field, the tuple itself will contain a pointer to the field in the partition's heap space, so tuple growth will not cause tuples to move [1] Since tuples in memory can be randomly accessed with no loss in performance, it is possible for the MM-DBMS to use pointers where it would otherwise be necessary to copy data in a disk-based DBMS For example, if foreign keys (attributes that reference tuples in other relations) are identified in the manner proposed by Date [Dat85], the MM-DBMS can substitute a tuple pointer field for the foreign key field (This field could hold a single pointer value in the case of a one to one relationship, or it could hold a list of pointers if the relationship is one to many) When the foreign key field's value is referenced, the MM-DBMS can simply follow the pointer to the foreign relation tuple to obtain the desired value This will be more space efficient, as pointers will usually be as small as or smaller than data values (especially when the values are strings) This will also enhance retrieval performance by allowing the use of precomputed joins Consider the following example

Employee Relation (Name, Id, Age, Dept_Id)
Department Relation (Name, Id)

Query 1 Retrieve the Employee name, Employee age, and Department name for all employees over age 65

Most conventional DBMSs lack precomputed joins and would require a join operation to answer this query Even with precomputed joins, a conventional DBMS would need to have the Department tuples clustered with the Employee tuples or it could pay the price of a disk access for every Department tuple retrieved In the MM-DBMS, using precomputed joins is much easier Assuming that the Emp Dept_id field has been identified as a foreign key that references Department tuples, the MM-DBMS will substitute a Department tuple pointer in its place, The MM-DBMS can then simply perform the selection on the Employee relation, following the Department pointer of each result tuple

Assuming that the Department Relation does not have pointers to the Employee Relation, retrieving data in the other direction would still require a join operation, but the join's comparison can be done on pointers rather than on data Using the relations from the example above, consider the following query

Query 2 Retrieve the names of all employees who work in the Toy or Shoe Departments

To process this query, a selection will be done on the Department relation to retrieve the "Shoe" and "Toy" Department tuples, and the result will then be joined with the Employee relation For the join, comparisons will be performed using the tuple pointers for the selection's result and the Department tuple pointers in the Employee relation While this would be equivalent in cost to joining on Dept_Id in this example, it could lead to a significant cost savings if the join columns were string values instead

2 2 Indices

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, compression techniques, and calculating storage requirements for 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 Figure 1 shows an example of two indices built for the Employee relation (The indices are shown as sorted tables for simplicity)

The MM-DBMS design has two types of dynamic index structures, each serving a different purpose The T Tree [LeC85], a relatively new index structure designed for use in main memory, is used as the general purpose index for ordered data It is able to grow and shrink gracefully, be scanned in either direction, use storage efficiently, and handle duplicates with little extra work Modified Linear Hashing, a variant of Linear Hashing [Lit80] that has been modified for use in main memory [LeC85], is used for unordered data Several other index structures were constructed to aid in the examination of join and project methods shown later in this paper The array index structure [AHK85] was used to store ordered data It is easy to build and scan, but it is useful only as a read-only index because it does not handle updates well Chained Bucket Hashing [AHU74] was used as the temporary index structure for unordered data, as it has excellent performance for static data (Originally, Chained Bucket Hashing was going to be used for static structures in the MM-DBMS, but it has since been replaced by Modified Linear Hashing, because it was discovered that the two have similar performance when the number of elements remains static)

2 3 Temporary lists

The MM-DBMS uses a temporary list structure for storing intermediate result relations A temporary list is a list of tuple pointers plus an associated result descriptor The pointers point to the source relation(s) from which the temporary relation was formed,

[1] In rare cases where a tuple causes the heap space to overflow, it will have to be moved to another partition, in which case a forwarding address will be left in its old position

and the result descriptor identifies the fields that are contained in the relation that the temporary list represents The descriptor takes the place of projection — no width reduction is ever done, so there is little motivation for computing projections before the last step of query processing unless a significant number of duplicates can be eliminated Unlike regular relations, a temporary list can be traversed directly, however, it is also possible to have an index on a temporary list

As an example, if the Employee and Department relations of Figure 1 were joined on the Department Id fields, then each result tuple in the temporary list would hold a pair of tuple pointers (one pointing to an Employee tuple and one pointing to a Department tuple), and the result descriptor would list the fields in each relation that appear in the result Figure 1 also shows the result list for such an equijoin on Department Id (Query 1)

Employee Relation

Ptr	*Name*	*Id*	*Age*	*Dept_id*
124	Dave	23	24	459
105	Suzan	12	27	459
137	Yaman	44	54	411
110	Jane	43	47	411
102	Cindy	22	22	409

Employee Indices

Name	*Id*
102	105
124	102
110	124
105	110
137	137

Department Relation

Ptr	*Name*	*Id*
243	Toy	459
201	Shoe	409
213	Linen	411
287	Paint	455

Result Relation

(124, 243)
(105, 243)
(137, 213)
(110, 213)
(102, 201)

Result Descriptor

Emp Name
Emp Age
Dept Name

Figure 1 — Relation and Index Design

2 4 Concurrency Control and Recovery

The MM-DBMS is intended to provide very high performance for the applications that it is capable of serving, many of which will require their data to be stored safely on disk as well as in memory Thus, the MM-DBMS must have a fast recovery mechanism The system is intended for multiple users, so it must also provide concurrency control While we have not yet finished the design of these subsystems, we wish to point out some of the major issues that are guiding their design

One proposed solution to the recovery problem is to use battery-backup RAM modules [LeR85], but this does not protect memory from the possibility of a media failure — a malfunctioning CPU or a memory failure could destroy a several gigabyte database Thus, disks will still be needed to provide a stable storage medium for the database Given the size of memory, applications that depend on the DBMS will probably not be able to afford to wait for the entire database to be reloaded and brought up to date from the log Thus, we are developing an approach that will allow normal processing to continue immediately, although at a slower pace until the working sets of the current transactions are read into main memory

Our approach to recovery in the MM-DBMS is based on an active log device During normal operation, the log device reads the updates of committed transactions from the stable log buffer and updates the disk copy of the database The log device holds a change accumulation log, so it does not need to update the disk version of the database every time a partition is modified The MM-DBMS writes all log information directly into a stable log buffer before the actual update is done to the database, as is done in IMS FASTPATH [IBM79] If the transaction aborts, then the log entry is removed and no undo is needed If the transaction commits, then the updates are propagated to the database After a crash, the MM-DBMS can continue processing as soon as the working sets of the current transactions are present in main memory The process of reading in a working set works as follows Each partition that participates in the working set is read from the disk copy of the database The log device is checked for any updates to that partition that have not yet been propagated to the disk copy Any updates that exist are merged with the partition on the fly and the updated partition is placed in memory Once the working set has been read in, the MM-DBMS should be able to run at close to its normal rate while the remainder of the database is read in by a background process A related proposal for main memory database recovery has been developed in parallel with ours [Eic86], since both schemes are in their development stages, however, it would be premature to compare them here

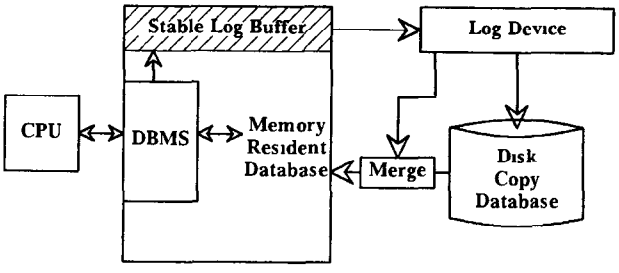


Figure 2 — Recovery Components

Concurrency control costs are different for a memory resident database Transactions will be much shorter in the absence of disk accesses In this environment, it will be reasonable to lock large items, as locks will be held for only a short time Complete serialization would even be possible if all transactions could be guaranteed to be reasonably short, but transaction interleaving is necessary for fairness if some transactions will be long We expect to set locks at the partition level, a fairly coarse level of granularity, as tuple-level locking would be prohibitively expensive here (A lock table is basically a hashed relation, so the cost of locking a tuple would be comparable to the cost of accessing it — thus doubling the cost of tuple accesses if tuple-level locking is used) Recall that the size of a partition is expected to be on the order of one or several disk tracks (since this is the unit of recovery) Partition-level locking may lead to problems with certain types of transactions that are inherently long (e g , conversational transactions) We will address these issues in future work

3 Query Processing in Main Memory DBMS

The direct addressability of data in a memory resident database has a profound impact on query processing With the notion of clustering removed, the methods for selection, join and projection acquire new cost formulas Old and new algorithms for these query processing operations were tested to determine which algorithms perform best in a main memory environment

3 1 The Test Environment

All of the tests reported here were run on a PDP VAX 11/750 running with two megabytes of real memory (as opposed to virtual memory) Each of the algorithms was implemented in the C programming language, and every effort was made to ensure that the quality of the implementations was uniform across the algorithms The validity of the execution times reported here was verified by

recording and examining the number of comparisons, the amount of data movement, the number of hash function calls, and other miscellaneous operations to ensure that the algorithms were doing what they were supposed to (i e , neither more nor less) These counters were compiled out of the code when the final performance tests were run, so the execution times presented here reflect the running times of the actual operations with very little time spent in overhead (e g , driver) routines Timing was done using a routine similar to the 'getrusage' facility of Unix [2]

3 2 Selection

This section summarizes the results from a study of index mechanisms for main memory databases [LeC85] The index structures tested were AVL Trees [AHU74], B Trees [Com79][3], arrays [AHK85], Chained Bucket hashing [Knu73], Extendible Hashing [FNP79], Linear Hashing [Lit80], Modified Linear Hashing [LeC85], and one new method, the T Tree [LeC85] (Modified Linear Hashing uses the basic principles of Linear Hashing, but uses very small nodes in the directory, single-item overflow buckets, and average overflow chain length as the criteria to control directory growth) All of these index structures, except for the T Tree, are well-known, and their algorithms are described in the literature Thus, we describe only the T Tree here

3 2 1 The T Tree Index Structure

The T Tree is a new balanced tree structure that evolved from AVL and B Trees, both of which have certain positive qualities for use in main memory The AVL Tree was designed as an internal memory data structure It uses a binary tree search, which is fast since the binary search is *intrinsic* to the tree structure (i e , 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 The B Tree is also good for memory use — its storage utilization is better since there are many data items per pointer[4], searching is fairly fast since a small number of nodes are searched with a binary search, and updating is fast since data movement usually involves only one node

The T Tree is a binary tree with many elements per node (Figure 3) Figure 4 shows 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, and, 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

To aid in our discussion of T Trees, we begin by introducing some helpful terminology There are three different types of T-nodes, as shown in Figure 4 A T-node that has two subtrees is

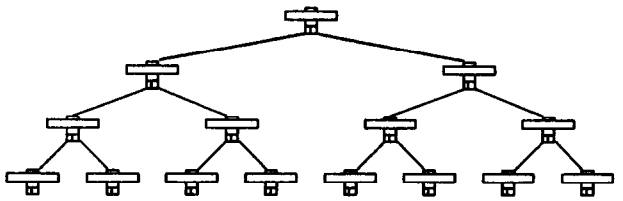


Figure 3 — A T Tree

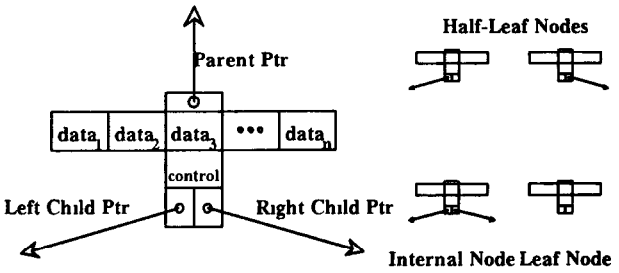


Figure 4 — T Nodes

called an *internal node* A T-node that has one NIL child pointer and one non-NIL child pointer is called a *half-leaf* A node that has two NIL child pointers is called a *leaf* 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. 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*

Associated with a T Tree is a minimum count and a maximum count Internal nodes 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

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 search consists of a binary tree search to find the node that bounds the search value and then a binary search of the node to find the value, if such a node is found

To insert into a T Tree, one first searches for a node that bounds the insert value If such a node is found, the item is inserted there If the insert causes an overflow, the minimum element[5] of that

[2]Unix is a trademark of AT&T Bell Laboratories

[3] We refer to the original B Tree, not the commonly used B+ Tree Tests reported in [LeC85] showed that the B+ Tree uses more storage than the B Tree and does not perform any better in main memory

[4] A B Tree internal node contains (N + 1) node pointers for every N data items while a B Tree leaf node contains only data items Since leaf nodes greatly outnumber internal nodes for typical values of N, there are many data items per node pointer

[5]Moving the minimum element requires less total data movement than moving the maximum element Similarly, when a node underflows because of a deletion, borrowing the greatest lower bound from a leaf node requires less work than borrowing the least upper bound These details are explained in [LeC85]

node is transferred to a leaf node, becoming the new greatest lower bound for the node it used to occupy If no bounding node can be found, then the leaf node where the search ended is the node where the insert value goes If the leaf node is full, a new leaf is added and the tree is rebalanced

To delete from a T Tree, one first searches for the node that bounds the delete value Then, one searches the node for the delete value If a bounding node is not found, or the delete value within the bounding node is not found, the delete returns unsuccessful Otherwise, the item is removed from the node If deleting from the node causes an underflow, then the greatest lower bound for this node is borrowed from a leaf If this causes a leaf node to become empty, the leaf node is deleted and the tree is rebalanced If there is no leaf to borrow from, then the node (which must be a leaf) is allowed to underflow

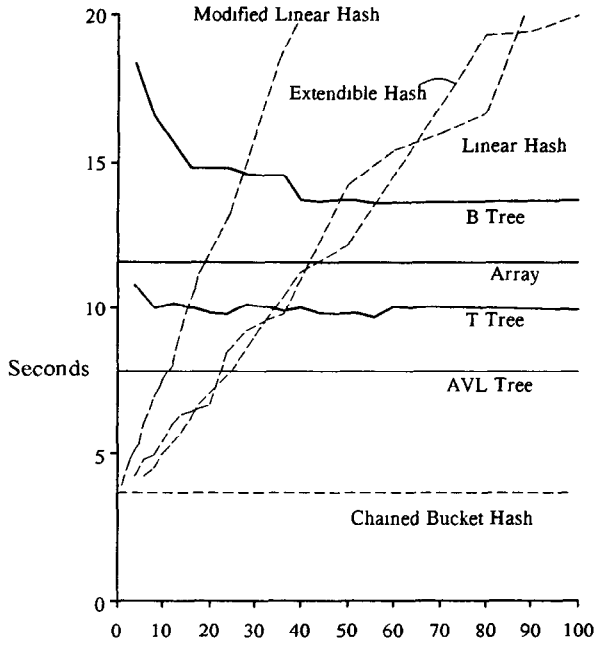
3 2 2 The Index Tests

Each index structure (arrays, AVL Trees, B Trees, Chained Bucket Hashing, Extendible Hashing, Linear Hashing, Modified Linear Hashing, and T Trees) was tested for all aspects of index use creation, search, scan, range queries (hash structures excluded), query mixes (intermixed searches, inserts and deletes), and deletion Each test used index structures filled with 30,000 unique elements (except for create, which inserted 30,000 elements) The indices were configured to run as unique indices — no duplicates were permitted The index structures were constructed in a "main memory" style, that is, the indices held only tuple pointers instead of actual key values or whole tuples We summarize the results of three of the tests from [LeC85] searching, a query mix of searches and updates, and storage cost measurements In order to compare the performance of the index structures in the same graphs, the number of variable parameters of the various structures was reduced to one — node size In the case of Modified Linear Hashing, single-item nodes were used, so the "Node Size" axis in the graphs refers to the average overflow bucket chain length Those structures without variable node sizes simply have straight lines for their execution times The graphs represent the hashing algorithms with dashed lines and the order-preserving structures with solid lines

Search

Graph 1 shows the search times of each algorithm for various node sizes 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 linked list of single-item nodes, so each data reference requires traversing a pointer This overhead is noticeable when the chain becomes long (Recall that



Graph 1 — Index Search

"Node Size" is really average chain length for Modified Linear Hashing)

Query Mix

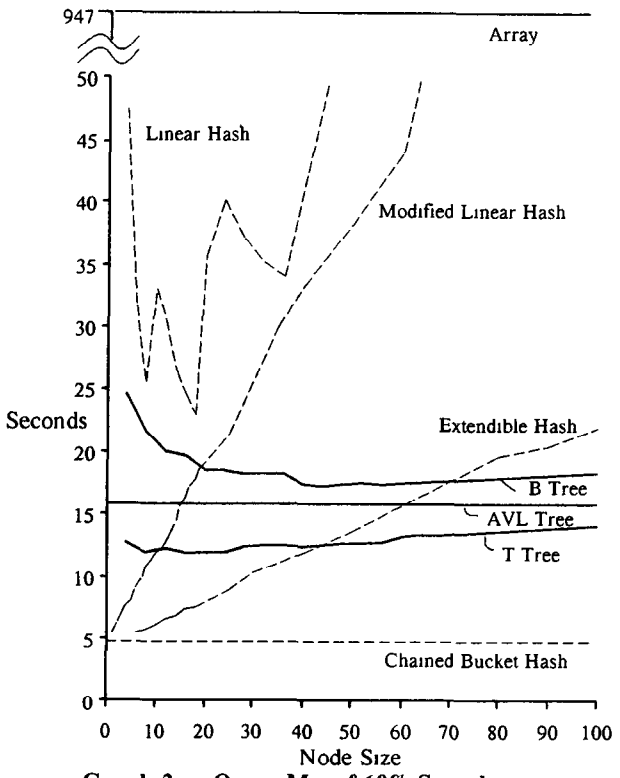
The query mix test is most important, as it shows the index structures in a normal working environment Tests were performed for three query mixes using different percentages of interspersed searches, inserts and deletes

1) 80% searches, 10% inserts, 10% deletes
2) 60% searches, 20% inserts, 20% deletes
3) 40% searches, 30% inserts, 30% deletes

The query mix of 60 percent searches, 20 percent inserts and 20 percent deletes (Graph 2) was representative of the three query mix graphs 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 similar 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, was much slower because, trying to maintain a particular storage utilization (number of data bytes used / total number of data bytes available), it did a significant amount of data reorganization even though the number of elements was relatively constant As for the array index, its performance was two orders of magnitude worse than that of the other index structures because of the large amount of data movement required to keep the array in sorted order (Every update requires moving half of the array, on the average)

Storage Cost

Space considerations preclude the inclusion of the storage results graph, but we summarize them here 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



Graph 2 — Query Mix of 60% Searches

cost First, we consider the fixed values the AVL Tree storage factor was 3 because of the two node pointers it needs for each data item, and Chained Bucket Hashing had a storage factor of 2 3 because it had one pointer for each data item and part of the table remained unused (the hash function was not *perfectly* uniform) Modified Linear Hashing was similar to Chained Bucket Hashing for an average hash chain length of 2, but, for larger hash chains, the number of empty slots in the table decreased and eventually the table became completely full Finally, Linear Hashing, B Trees, Extendible Hashing and T Trees all had nearly equal storage factors of 1 5 for medium to large size nodes Extendible Hashing tended to use the largest amount of storage for small nodes sizes (2,4 and 6) This was because a small node size increased the probability that some nodes would get more values than others, causing the directory to double repeatedly and thus use large amounts of storage As its node size was increased, the probability of this happening became lower

3 2 3 Index Study Results

Table 1 summarizes the results of our study of main memory index structures We use a four level rating scale (poor, fair, good, great) to show the performance of the index structures in the three categories An important thing to notice about the hash-based indices is that, while 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, the version of Modified Linear Hashing tested here used single-item nodes, so there was 4 bytes of pointer overhead for each data item) As for the other two hash-based methods Chained Bucket Hashing had good search and update performance, but it also had fairly high storage costs, and it is only a static structure, and finally, Linear Hashing is just too slow to use in main memory Among the hash-based methods tested, Modified Linear Hashing provided the best overall performance

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 reasonable search 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 uniformly well in all of the tests

Data Structure	*Search*	*Update*	*Storage Cost*
Array	good	poor	good
AVL Tree	good	fair	poor
B Tree	fair	good	good
T Tree	good	good	good
Chained Bucket Hash	great	great	fair
Extendible Hash	great	great	poor
Linear Hash	great	poor	good
Mod Linear Hash	great	great	fair/good

Table 1 — Index Study Results

3 3 Join

Previous join studies involving large memories have been based on the large buffer pool assumption [Sha86], [DKO84], [DeG85] (Others have studied hash joins as well in a normal disk environment [Bab79], [VaG84], [Bra84], but their results are less applicable here) Three main join methods were tested in [DeG85] Nested Loops with a hashed index, Sort Merge [BlE77], and three hashing methods, Simple Hash, Hybrid Hash and GRACE Hash [DKO84] The results showed that when both relations fit in memory, the three hash algorithms became equivalent, and the nested loops join with a hash index was found to perform just as well as the other hash algorithms (and outperformed Sort Merge) They also studied the use of semijoin processing with bit vectors to reduce the number of disk accesses involved in the join, but this semijoin pass is redundant when the relations are memory resident The variety of join relation compositions (e g , sizes, join selectivities, join column value distributions) used in their study was small, and may not completely reflect all possibilities (performance-wise)

In this study, we examine the performance of a number of candidate join methods for the MM-DBMS We use a wide selection of relation compositions so as to evaluate the algorithms under a wide variety of possible conditions

3 3 1 Relation Generation

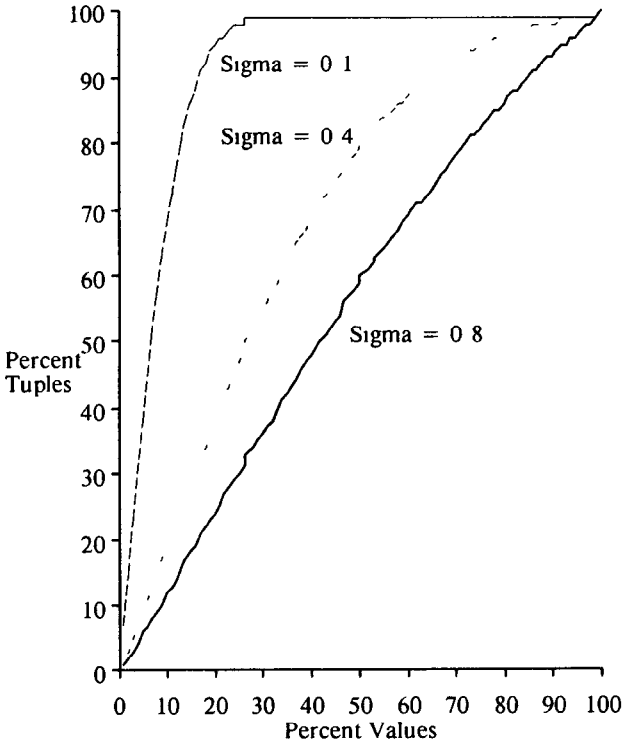
In order to support our intent to test a variety of relation compositions, we constructed our test relations so that we could vary several parameters The variable parameters were

(1) The relation cardinality (|R|)

(2) The number of join column duplicate values (as a percentage of |R|) and their distribution

(3) The semijoin selectivity (the number of values in the larger relation that participate in the join, expressed as a percentage of the larger relation)

In order to get a variable semijoin selectivity, the smaller relation was built with a specified number of values from the larger relation To get a variable number of duplicates, a specified number of unique values were generated (either from a random number generator or from the larger relation), and then the number of occurrences of each of these values was determined using a random sampling procedure based on a truncated normal distribution with a variable standard deviation Graph 3 shows the three duplicate distributions used for the tests — a skewed distribution (where the standard deviation was 0 1), a moderately skewed distribution (the 0 4 curve in the graph), and a near-uniform distribution (the 0 8 curve in the graph)



Graph 3 — Distribution of Duplicate Values

The results for the 0 4 and 0 8 cases were similar, so results are given here only for the two extreme cases

3.3 2 The Join Algorithms

For memory resident databases, all of the hash-based algorithms tested in [DeG85] were found to perform equally well Therefore, the hash-based nested loops algorithm is the only hash-based algorithm that we examine here For our tests, we implemented and measured the performance of a total of five join algorithms Nested Loops, a simple main-memory version of a nested loops join with no index, Hash Join and Tree Join, two variants of the nested loops join that use indices, and Sort Merge and Tree Merge, two variants of the sort-merge join method of [BIE77] We briefly describe each of these methods in turn Recall that relations are always accessed via an index, unless otherwise specified, an array index was used to scan the relations in our tests

The pure Nested Loops join is an $O(N^2)$ algorithm It uses one relation as the outer, scanning each of its tuples once For each outer tuple, it then scans the entire inner relation looking for tuples with a matching join column value The Hash Join and Tree Join algorithms are similar, but they each use an index to limit the number of tuples that have to be scanned in the inner relation The Hash Join builds a Chain Bucket Hash index on the join column of the inner relation, and then it uses this index to find matching tuples during the join The Tree Join uses an existing T Tree index on the inner relation to find matching tuples We do not include the possibility of building a T Tree on the inner relation for the join because it turns out to be a viable alternative only if the T tree already exists as a regular index — if the cost to build the tree is included, a Tree Join will *always* cost more than a Hash Join, as a T tree costs more to build and a hash table is faster for single value retrieval [LeC85] On the other hand, we always include the cost of building a hash table, because we feel that a hash table index is less likely to exist than a T Tree index The cost of creating a hash table with 30,000 elements is about 5 seconds in our environment [LeC85]

The merge join algorithm [BIE77] was implemented using two index structures, an array index and a T Tree index For the Sort Merge algorithm tested here, array indexes were built on both relations and then sorted The sort was done using quicksort with an insertion sort for subarrays of ten elements or less [6] For the Tree Merge tests, we built T Tree indices on the join columns of each relation, and then performed a merge join using these indices However, we do not report the T Tree construction times in our tests — it turns out that the T Merge algorithm is only a viable alternative if the indices already exist. Preliminary tests showed that the arrays can be built and sorted in 60 percent of the time to build the trees, and also that the array can be scanned in about 60 percent of the time it takes to scan a tree

3.3 3 Join Tests

The join algorithms were each tested with a variety of relation compositions in order to determine their relative performance Six tests were performed in all, and they are summarized below In our description of the tests, |R1| denotes the outer relation and |R2| denotes the inner relation

(1) *Vary Cardinality* Vary the sizes of the relations with |R1| = |R2|, 0% duplicates, and a semijoin selectivity of 100%

(2) *Vary Inner Cardinality* Vary the size of R2 (|R2| = 1-100% of |R1|) with |R1| = 30,000, 0% duplicates, and a semijoin selectivity of 100%

(3) *Vary Outer Cardinality* Vary the size of R1 (|R1| = 1-100% of |R2|) with |R2| = 30,000, 0% duplicates, and a semijoin selectivity of 100%

(4) *Vary Duplicate Percentage (skewed)* Vary the duplicate percentage of both relations from 0-100% with |R1| = |R2| = 20,000, a semijoin selectivity of 100%, and a skewed duplicate distribution

(5) *Vary Duplicate Percentage (uniform)* Vary the duplicate percentage of both relations from 0-100% with |R1| = |R2| = 20,000, a semijoin selectivity of 100%, and a uniform duplicate distribution

(6) *Vary Semijoin Selectivity* Vary the semijoin selectivity from 1-100% with |R1| = |R2| = 30,000 and a duplicate percentage of 50% with a uniform duplicate distribution

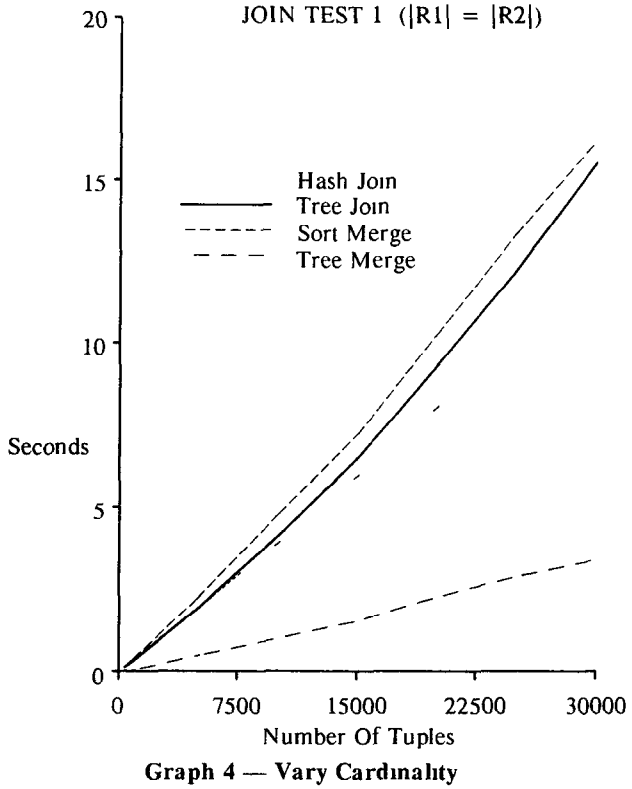
[6] We ran a test to determine the optimal subarray size for switching from quicksort to insertion sort, the optimal subarray size was 10

3 3 4 Join Test Results

We present the results of each of the join tests in this section The results for the Nested Loops algorithm will be presented separately at the end of the section, as its performance was typically two orders of magnitude worse than that of the other join methods

Test 1 — Vary Cardinality

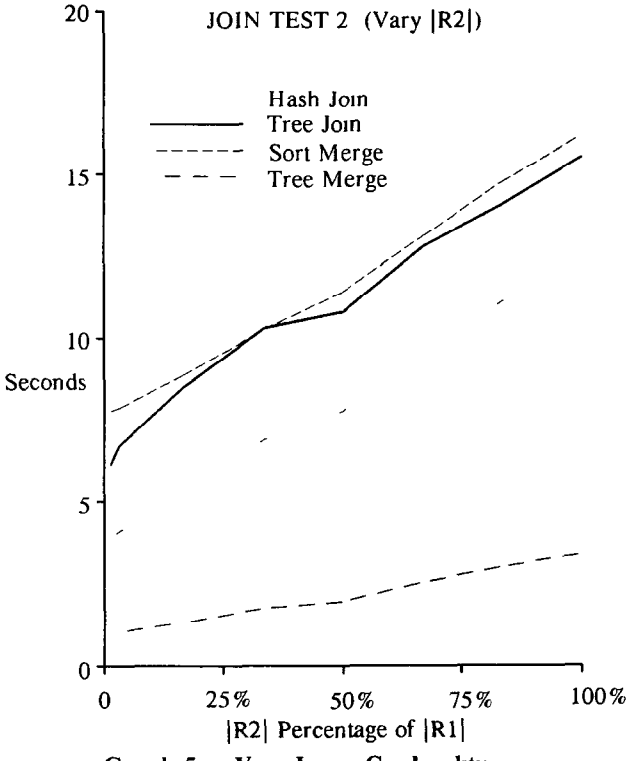
Graph 4 shows the performance of the join methods for relations with equal cardinalities The relations are joined on keys (i e , no duplicates) with a semijoin selectivity of 100% (i e , all tuples participate in the join) If both indices are available, then a Tree Merge gives the best performance It does the least amount of work, as the T Tree indices are assumed to exist, and scanning them in order limits the number of comparisons required to perform the join The number of comparisons done is approximately (|R1| + |R2| * 2), as each element in R1 is referenced once and each element in R2 is referenced twice (The presence of duplicates would increase the number of times the elements in R2 are referenced) If it is not that case that both indices are available, it is best to do a Hash Join It turns out that, in this case, it is actually faster to build and use a hash table on the inner relation than to simply use an existing T Tree index A Hash table has a fixed cost, independent of the index size, to look up a value The number of comparisons done in a Hash Join is approximately (|R1| + (|R1| * k)) where k is the fixed lookup cost, whereas the number of comparisons in a Tree Join is roughly (|R1| + (|R1| * Log_2(|R2|))) The value of k is much smaller than Log_2(|R2|)) but larger than 2 Finally, the Sort Merge algorithm has the worst performance of the algorithms in this test, as the cost of building and sorting the arrays for use in the merge phase is too high ((|R1| * Log_2(|R1|)) + (|R2| * Log_2(|R2|)) + (|R1| + |R2|))



Graph 4 — Vary Cardinality

Test 2 — Vary Inner Cardinality

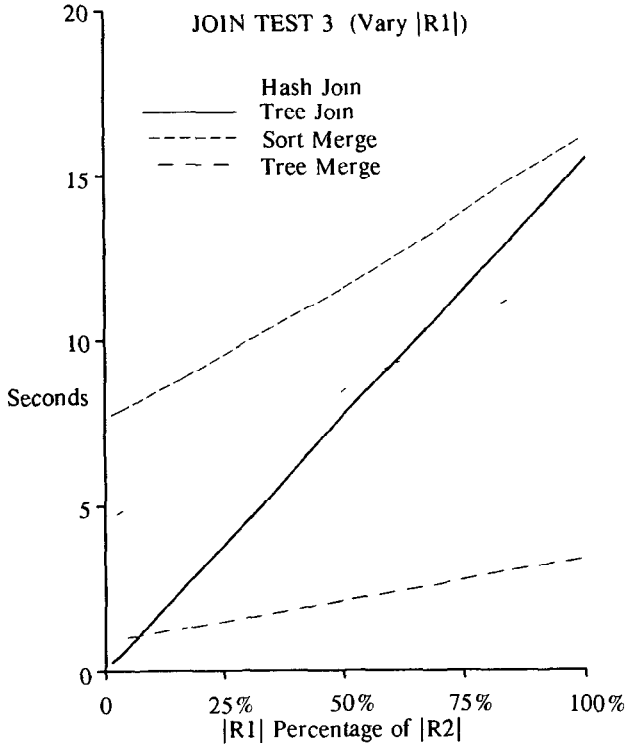
Graph 5 shows the performance of the join methods as R2's cardinality is varied from 1-100% of the cardinality of R1 In this test, R1's cardinality is fixed at 30,000, the join columns were again keys (i e , no duplicates), and the semijoin selectivity was again 100% The results obtained here are similar to those of Test 1, with Tree Merge performing the best if T Tree indices exist on both join columns, and Hash Join performing the best otherwise In this test, each of the the index joins were basically doing |R1| searches of an index of (increasing) cardinality |R2|



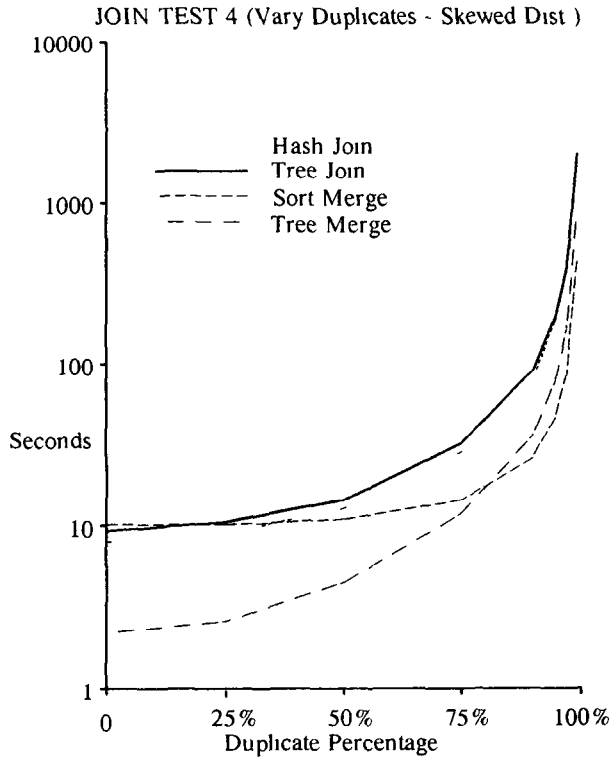
Graph 5 — Vary Inner Cardinality

Test 3 — Vary Outer Cardinality

The parameters of Test 3 were identical to those of Test 2 except that |R1| was varied instead of |R2| The results of this test are shown in Graph 6 The Tree Merge, Hash Join, and Sort Merge algorithms perform much the same as they did in Test 2 In this case, however, the Tree Join outperforms the others for small values of |R1|, beating even the Tree Merge algorithm for the smallest |R1| values This is intuitive, as this algorithm behaves like a simple selection when |R1| contains few tuples Once |R2| increases to about 60% of |R1|, the Hash Join algorithm becomes the better method again because the speed of the hash lookup overcomes the initial cost of building the hash table, both of which combined are cheaper than the cost of many T Tree searches for large values of |R1| Note if a hash table index already existed for R2, then the Hash Join would be faster than the Tree Join (recall that building the hash table takes about 5 seconds)



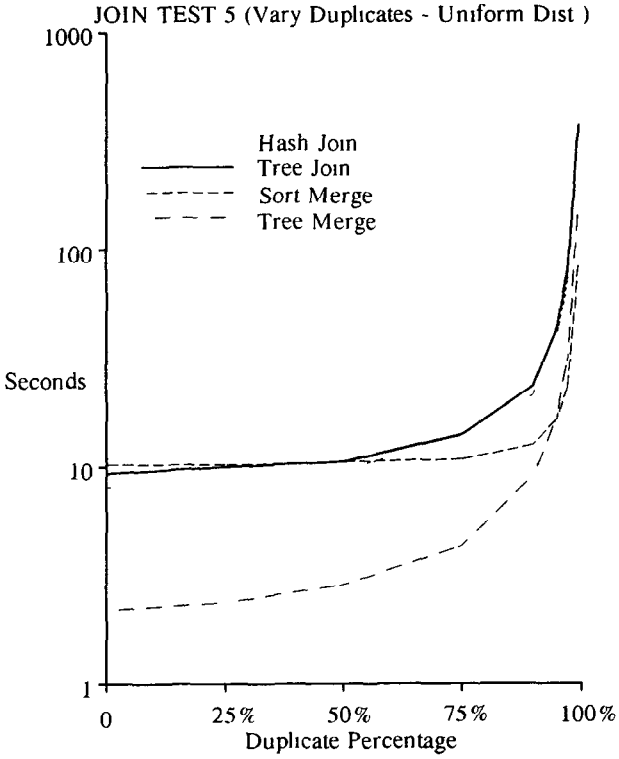
Graph 6 — Vary Outer Cardinality



Graph 7 — Vary Duplicate Percentage (skewed)

Test 4 — Vary Duplicate Percentage (skewed)

For test 4, |R1| and |R2| were fixed at 20,000, the semijoin selectivity was kept at 100%, and the duplicate percentage for both relations was varied from 1 to 100% The results of this test are shown in Graph 7 The duplicate distribution was skewed, so there were many duplicates for some values and few or none for others (The duplicate percentages of the two relations were different in this test — a result of the relation construction procedure In order to achieve 100 percent semijoin selectivity, the values for R2 were chosen from R1, which already contained a non-uniform distribution of duplicates Therefore, number of duplicates in R2 is greater than that of R1 The duplicate percentages in Graph 7 refer to R1) Once the number of duplicates becomes significant, the number of matching tuples (and hence result tuples) becomes large, resulting in many more tuples being scanned The Sort Merge method is the most efficient of the algorithms for scanning large numbers of tuples — once the skewed duplicate percentage reaches about 80 percent, the cost of building and sorting the arrays is overcome by the efficiency of scanning the relations via the arrays, so it beats even Tree Merge in this case Although the number of comparisons is the same, as both Tree Merge and Sort Merge use the same Merge Join algorithm, the array index can be scanned faster than the T Tree index because the array index holds a list of contiguous elements whereas the T Tree holds nodes of contiguous elements joined by pointers Test results from [LeC85] show that the array can be scanned in about 2/3 the time it takes to scan a T Tree The Index Join methods are less efficient for processing large numbers of elements for each join value, so they begin to lose to Sort Merge when the skewed duplicate percentage reaches about 40 percent



Graph 8 — Vary Duplicate Percentage (uniform)

Test 5 — Vary Duplicate Percentage (uniform)

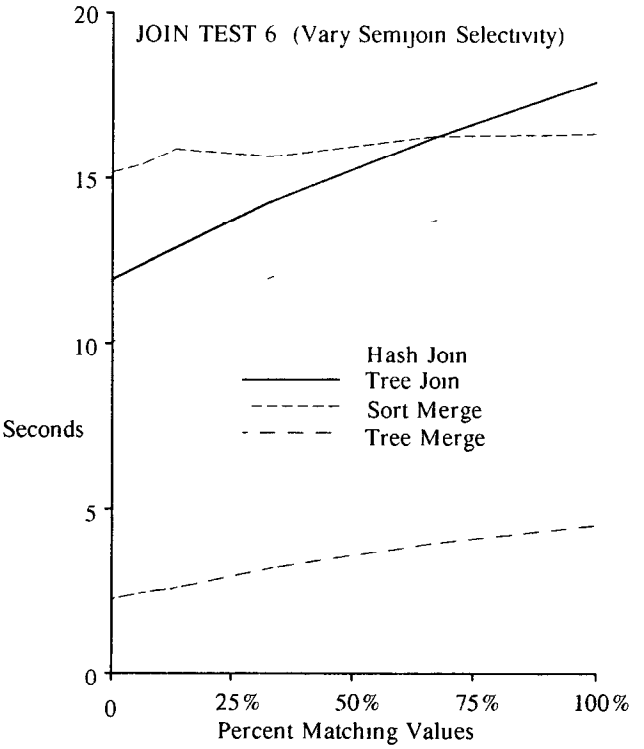
Test 5 is identical to Test 4 except that the distribution of duplicates was uniform The results of Test 5 are shown in Graph 8

(Note that the duplicate percentages of R1 and R2 are the same here, because R2 was created with a uniform distribution of R1 values) Here, the Tree Merge algorithm remained the best method until the duplicate percentage exceeded about 97 percent because the output

of the join was much lower for most duplicate percentages When the duplicate percentages were low (0-60 percent), the join algorithms had behavior similar to that of earlier tests Once the duplicate percentage became high enough to cause a high output join (at about 97 percent), Sort Merge again became the fastest join method

Test 6 — Vary Semijoin Selectivity

In the previous tests, the semijoin selectivity was held constant at 100% In Test 6, however, it was varied, and the results of this test are shown in Graph 9 For this test, |R1| = |R2| = 30,000 elements, the duplicate percentage was fixed at 50% in each relation with a uniform distribution (so there were roughly two occurrences of each join column value in each relation), and the semijoin selectivity was varied from 1-100% The Tree Join was affected the most by the increase in matching values, a brief description of the search procedure will explain why When the T Tree is searched for a set of tuples with a single value, the search stops at any tuple with that value, and the tree is then scanned in both directions from that position (since the list of tuples for a given value is logically contiguous in the tree) If the initial search does not find any tuples matching the search value, then the scan phase is bypassed and the search returns unsuccessful When the percentage of matching values is low then, most of the searches are unsuccessful and the total cost is much lower than when the majority of searches are successful A similar case can be made for the Hash Join in that unsuccessful searches sometimes require less work than successful ones — an unsuccessful search may scan an empty hash chain instead of a full one The increase in the Tree Merge execution time in Graph 9 was due mostly to the extra data comparisons and the extra overhead of recording the increasing number of matching tuples Sort Merge is less affected by the increase in matching tuples because the sorting time overshadows the time required to perform the actual merge join



Graph 9 — Vary Semijoin Selectivity

3 3 5 Join Test Result Summary

If the proper pair of tree indices is present, the Tree Merge join method was found to perform the best in almost all of the situations tested It turned out never to be advantageous to build the T Tree indices for this join method, however, as it would then be slower then the other three methods In situations where one of the two relations is missing a join column index, the Hash Join method was found to be the best choice There are only two exceptions to these statements

(1) If an index exists on the larger relation and the smaller relation is less than half the size of the larger relation, then a Tree Join (T Tree index join) was found to execute faster than a Hash Join In this situation, the tuples in the smaller relation can be looked up in the tree index faster than a hash table can be built and scanned This would also be true for a hash index if it already existed

(2) When the semijoin selectivity and the duplicate percentage are both high, the Sort Merge join method should be used, particularly if the duplicate distribution is highly skewed A Tree Merge join is also satisfactory is this case, but the required indices may not be present If the indices must be built, then the Tree Merge join will be more costly than the Hash Join for duplicate percentages less then 60 in the skewed case and 80 in the uniform case

It should be mentioned that only equijoins were tested Non-equijoins other than "not equals" can make use of ordering of the data, so the Tree Join should be used for such (<, ≤, >, ≥) joins

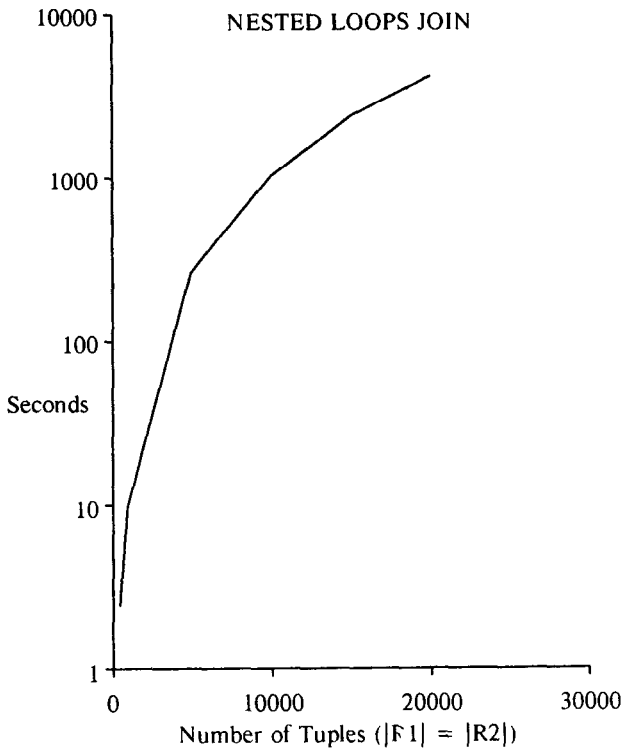
As mentioned earlier, we also tested the nested loops join method Due to the fact that its performance was usually several orders of magnitude worse than the other join methods, we were unable to present them on the same graphs Graph 10 shows the cost of nested loops join for a portion of Test 1, with |R1| = |R2| varied from 1,000 to 20,000 It is clear that, unless one plans to generate full cross products on a regular basis, nested loops join should simply never be considered as a practical join method for a main memory DBMS

The precomputed join described in Section 2 1 was not tested along with the other join methods Intuitively, it would beat each of the join methods in every case, because the joining tuples have already been paired Thus, the tuple pointers for the result relation can simply be extracted from a single relation

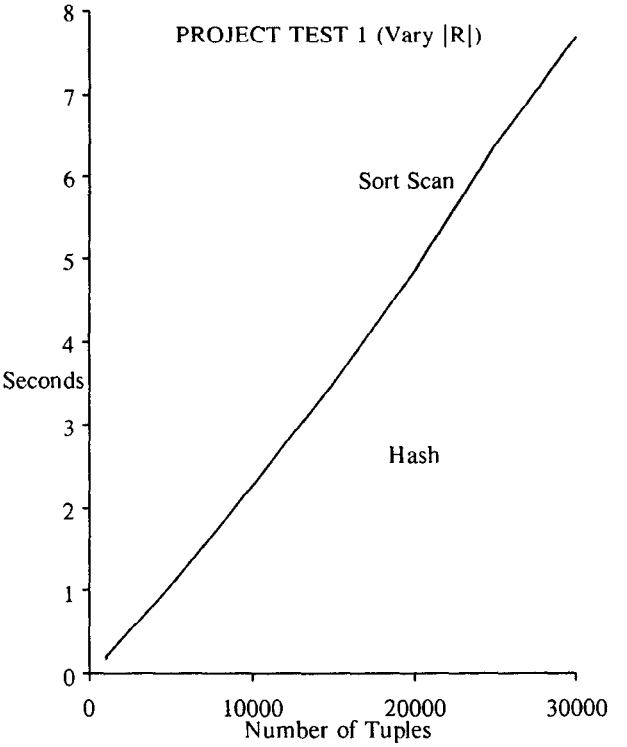
3 4 Projection

In our discussion of the MM-DBMS in Section 2, we explained that much of the work of the projection phase of a query is implicitly done by specifying the attributes in the form of result descriptors Thus, the only step requiring any significant processing is the final operation of removing duplicates For duplicate elimination, we tested two candidate methods Sort Scan [BBD83] and Hashing [DKO84] Again, we implemented both methods and compared their performance

In these tests, the composition of the relation to be projected was varied in ways similar to the those of the join tests — both the relation cardinality and its duplicate percentage were varied Since preliminary tests showed that the distribution of duplicates had no effect on the results, we do not vary the distribution in the tests presented here



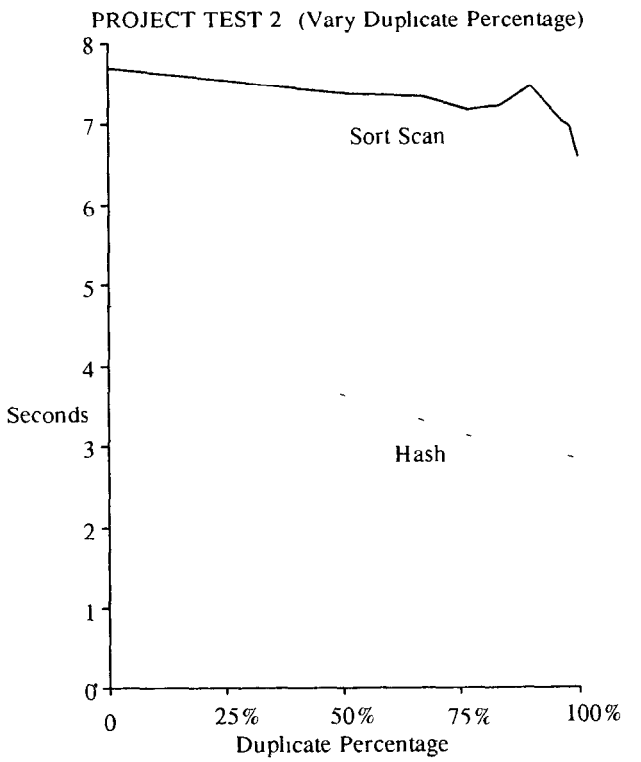
Graph 10 — Nested Loops Join



Graph 11 — Vary Cardinality

Graph 11 shows the performance of the two duplicate elimination algorithms for relations of various sizes For this test, no duplicates were actually introduced in the relation, so the starting size of the relation and its final size were the same The insertion overhead in the hash table is linear for all values of |R| (since the hash table size was always chosen to be |R|/2), while the cost for sorting goes as O(|R| log |R|) As the number of tuples becomes large, this sorting cost dominates the performance of the Sort Scan method In addition, these tests were performed using single column relations — the number of comparisons is much higher in the sort process, and this cost would only be exacerbated if more columns participated in the projection Thus, the Hashing method is the clear winner in this test

Graph 12 shows the results for a relation with 30,000 elements but a varying number of duplicates As the number of duplicates increases, the hash table stores fewer elements (since the duplicates are discarded as they are encountered) The Hashing method is thus able to run faster than it would with all the elements (since it has shorter chains of elements to process for each hash value) Sorting, on the other hand, realizes no such advantage, as it must still sort the entire list before eliminating tuples during the scan phase The large number of duplicates does affect the sort to some degree, however, because the insertion sort has less work to do when there are many duplicates — with many equal values, the subarray in quicksort is often already sorted by the time it is passed to the insertion sort



Graph 12 — Vary Duplicate Percentage

4 Conclusions and Future Work

In this paper, we have addressed query processing issues and algorithms for a main memory database management system We sketched an architecture for such a system, the MM-DBMS architecture, pointing out the major differences between disk-based databases and memory resident databases We then addressed the problem of processing relational queries in the MM-DBMS architecture, studying algorithms for the selection, join, and projection operations

A number of candidate algorithms were implemented for each operation, and their performance was experimentally compared We found that, for selection, the T Tree provides excellent overall performance for queries on ordered data, and that Modified Linear Hashing is the best index structure (of those examined) for unordered data For joins, when a precomputed join does not exist, we found that a T Tree based merge join offers good performance if both indices exist, and that hashing tends to offer the best performance otherwise A main memory variant of the sort merge algorithm was found to perform well for high output joins Finally, it was shown that hashing is the dominant algorithm for processing projections in main memory

In light of these results, query optimization in MM-DBMS should be simpler than in conventional database systems, as the cost formulas are less complicated [SAC79] The issue of clustering and projection for size reduction has been removed from consideration, thereby simplifying the choice of algorithms (Projection may be needed to reduce the number of duplicate entries in a temporary result, but it is never needed to reduce the *size* of the result tuples, because tuples are *never* copied, only pointed to) There are three possible access paths for selection (hash lookup, tree lookup, or sequential scan through an unrelated index), three main join methods (precomputed join, Tree Merge join, and Hash Join) and one method for eliminating duplicates (Hash) Moreover, the choice of which algorithm is simplified because there is a more definite ordering of preference a hash lookup (exact match only) is always faster than a tree lookup which is always faster than a sequential scan, a precomputed join is always faster than the other join methods, and a Tree Merge join is nearly always preferred when the T Tree indices already exist

5 References

[AHU74] A Aho, J Hopcroft and J Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley Publishing Company, 1974

[AHK85] A Ammann, M Hanrahan and R Krishnamurthy, Design of a Memory Resident DBMS, *Proc IEEE COMPCON*, San Francisco, February 1985

[Bab79] E Babb, Implementing a Relational Database by Means of Specialized Hardware, *ACM Transactions on Database Systems 4*,1 (March 1979), 1-29

[BBD83] D Bitton, H Boral, D DeWitt and W Wilkinson, Parallel Algorithms for the execution of Relational Database Operations, *ACM Transactions on Database Systems 8*,3 (September 1983), 324

[BlE77] M Blasgen and K Eswaran, Storage and Access in Relational Databases, *IBM Systems Journal 16*,4 (1977)

[Bra84] K Bratbergsengen, Hashing Methods and Relational Algebra Operations, *Proc of 10th Int Conf on Very Large Data Bases*, Singapore, August 1984, 323

[Com79] D Comer, The Ubiquitous B-Tree, *Computing Surveys 11*,2 (June 1979)

[Dat81] C J Date, *An Introduction to Database Systems*, Addison-Wesley, 1981 (3rd Ed)

[Dat85] C J Date, *An Introduction to Database Systems*, Addison-Wesley, 1985 (4th Ed)

[DKO84] D 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

[DeG85] D DeWitt and R Gerber, Multiprocessor Hash-Based Join Algorithms, *Proc of 11th Int Conf on Very Large Data Bases*, Stockholm, Sweden, August 1985

[Eic86] M Eich, MMDB Recovery, Southern Methodist Univ Dept of Computer Sciences Tech Rep # 86-CSE-11, March 1986

[ElB84] K Elhardt and R Bayer, A Database Cache for High Performance and Fast Restart in Database Systems, *ACM Transactions on Database Systems 9*,4 (December 1984), 503-526

[FNP79] R Fagin, J Nievergelt, N Pippenger and H Strong, Extendible Hashing - A Fast Access Method for Dynamic Files, *ACM Transactions on Database Systems 4*,3 (Sept 1979)

[Fis86] M Fishetti, Techology '86 Solid State, *IEEE Spectrum 23*,1 (January 1986)

[GLV83] H Garcia-Molina, R J Lipton and J Valdes, A Massive Memory Machine, Princeton Univ EECS Dept Tech Rep # 315, July 1983

[HoT85] S Horwitz and T Teitelbaum, Relations and Attributes A Symbiotic Basis for Editing Environments, *Proc of the ACM SIGPLAN Notices Conf on Language Issues in Programming Environments*, Seattle, WA, June 1985

[IBM79] IBM, *IMS Version 1 Release 1 5 Fast Path Feature Description and Design Guide*, IBM World Trade Systems Centers (G320-5775), 1979

[Knu73] D Knuth, *Sorting and Searching*, Addison-Wesley, 1973

[LeC85] T Lehman and M Carey, A Study of Index Structures for Main Memory Database Management Systems, UW CS Tech Rep # 605, July 1985 (A revised version has been submitted for publication)

[LeR85] M Leland and W Roome, The Silicon Database Machine, *Proc 4th Int Workshop on Database Machines*, Grand Bahama Island, March 1985

[Lin84] M Linton, Implementing Relational Views of Programs, *Proc of the ACM Software Eng Notes/SIGPLAN Notices Software Eng Symp on Practical Software Development Environments*, Pittsburgh, PA, April 1984

[Lit80] W Litwin, Linear Hashing A New Tool For File and Table Addressing, *Proc of 6th Int Conf on Very Large Data Bases*, Montreal, Canada, October 1980

[SAC79] P Selinger, M Astrahan, D Chamberlin, R Lorie and T Price, Access Path Selection in a Relational DBMS, *Proc ACM SIGMOD Conf*, June 1979

[Sha86] L D Shapiro, Join Processing in Database Systems with Large Main Memories, *ACM Transactions on Database Systems*, 1986 (to appear)

[Sno84] R Snodgrass, Monitoring in a Software Development Environment A Relational Approach, *Proc of the ACM Software Eng Notes/SIGPLAN Notices Software Eng Symp on Practical Software Development Environments*, Pittsburgh, PA, April 1984

[VaG84] P Valduriez and G Gardarin, Join and Semijoin Algorithms for a Multiprocessor Database Machine Transactions on Database Systems, *ACM Transactions on Database Systems 9*,1 (March 1984), 133

[War81] D H D Warren, Efficient Processing of Interactive Relational Database Queries Expressed in Logic, *Proc of 7th Int Conf on Very Large Data Bases*, Cannes, Fance, September, 1981