Implementation Techniques for Main Memory Database Systems

David J. DeWitt[1]
Randy H. Katz[2]
Frank Olken[3]
Leonard D. Shapiro[4]
Michael R. Stonebraker[2]
David Wood[2]

[1] Computer Sciences Department, University of Wisconsin
[2] EECS Department, University of California at Berkeley
[3] CSAM Department, Lawrence Berkeley Laboratory
[4] Department of Computer Science, North Dakota State University

This research was partially supported by the National Science Foundation under grants MCS82-01860, MCS82-01870, by the Department of Energy under contracts #DE-AC02-81ER10920 and #W-7405-ENG-48, and by the Air Force Office of Scientific Research under Grant 83-0021.

ABSTRACT

With the availability of very large, relatively inexpensive main memories, it is becoming possible keep large databases resident in main memory. In this paper we consider the changes necessary to permit a relational database system to take advantage of large amounts of main memory. We evaluate AVL vs. B+-tree access methods for main memory databases, hash-based query processing strategies vs. sort-merge, and study recovery issues when most or all of the database fits in main memory. As expected, B+-trees are the preferred storage mechanism unless more than 80-90% of the database fits in main memory. A somewhat surprising result is that hash based query processing strategies are advantageous for large memory situations.

Key Words and Phrases: Main Memory Databases, Access Methods, Join Algorithms, Access Planning, Recovery Mechanisms

1. Introduction

Throughout the past decade main memory prices have plummetted and are expected to continue to do so. At the present time, memory for super-minicomputers such as the VAX 11/780 costs approximately $1,500 a megabyte. By 1990, 1 megabit memory chips will be commonplace and should further reduce prices by another order of magnitude. Thus, in 1990 a gigabyte of memory should cost less than $200,000. If 4 megabit memory chips are available, the price might be as low as $50,000.

With the availability of larger amounts of main memory, it becomes possible to contemplate the storage of databases as main memory objects. In fact, IMS Fast Path [DATE82] has supported such databases for some time. In this paper we consider the changes that might be needed to a relational database system if most (or all) of a relation(s) is (are) resident in main memory.

In Section 2, the performance of alternative access methods for main memory database systems are considered. Algorithms for relational database operators in this environment are presented and evaluated in Section 3. In Section 4, we describe how access planning will be affected by the availability of large amounts of main memory for query processing. Section 5 discusses recovery in memory resident databases. Our conclusions and suggestions for future research are contained in Section 6.

2. Access Methods for Memory Resident Databases

The standard access method for data on disk is the B+-tree [COME79], providing both random and sequential key access. A B+-tree is specially designed to provide fast access to disk-resident data and makes fundamental use of the page size of the device. On the other hand, if a keyed relation is known to reside in main memory, then an AVL (or other binary) tree organization may be a better choice. In this section we analyze the performance of both structure for a relation R with the following characteristics:

||R||	number of tuples in relation R
K	width of the key for R in bytes
L	width of a tuple in bytes
P	page size in bytes
4	size of a pointer in bytes

We have analyzed two cases of interest. The first is the cost of retrieving a single tuple using a random key value. An example of this type of query is:

retrieve (emp.salary) where emp.name = "Jones"

The second case analyzed is the cost of reading N records sequentially. Consider the query

retrieve (emp.salary, emp.name) where emp.name = "J*"

which requests data on all employees whose names begin with J. To execute this query, the database system would locate the first employee with a name beginning with J and then read sequentially. This second case analyzes the sequential access portion of such a command.

For both cases (random and sequential access), there are two costs that are specific to the access method:

|page reads|	the number of pages read to execute the query
|comparisons|	the number of record comparisons required to isolate the particular data of interest.

The number of comparisons is indicative of the CPU time required to process the command while the number of page reads approximates the I/O costs.

To compare the performance AVL and B+-trees, we propose the following cost function:

$$\text{cost} = Z * |\text{page-reads}| + |\text{comparisons}|$$

Since a page read consumes perhaps 2000 instructions of operating system overhead and 30 milliseconds of elapsed time while a comparison can easily be done in 200, we expect realistic values of Z to be in the range of 10 to 30. Later in the section we will use several values in this range.

Moreover, it is possible (although not very likely) that an AVL-tree comparison will be cheaper than a B+-tree comparison. The reasoning is that the B+-tree record must be located within a page while an AVL tree does not contain any page structure and records can be directly located. Consequently, we assume that an AVL-tree comparison costs Y times a B+-tree comparison for some $Y < 1$.

From Knuth [KNUT73], we can observe that in an $||R||$-tuple AVL tree approximately

$$C = \log_2||R|| + 0.25 \text{ comparisons}$$

are required to find a tuple in a relation. Without any special precautions each of the C nodes to be inspected will be on a different page.[1] Hence, the number of pages accessed is approximately C. The

[1] If a paged binary tree organization is used instead, the fanout per node will be slightly worse than the B-tree. Furthermore,

AVL structure will occupy approximately

$$S = \left\lceil \frac{||R|| * (L+8)}{P} \right\rceil \text{ pages}$$

Here $\lceil X \rceil$ denotes the smallest integer larger than X. If |M| pages of main memory are available, and if $|M| < |S|$, and if a random replacement algorithm is used, the number of page faults to find a tuple in a relation will be approximately:

$$faults = C * (1 - \frac{|M|}{S})$$

Consequently the cost of a random access by key is:

$$cost(AVL) = Z*C*(1 - \frac{|M|}{S}) + Y*C$$

Next we derive the approximate cost for a random access to a tuple using a B+-tree. According to YAO [YAO78], B-tree nodes are approximately 69 percent full on the average. Hence, the fanout of a B+-tree is approximately

$$A = \frac{.69*P}{K+4}$$

The number of leaf nodes will be about

$$D = \frac{||R|| * L}{.69*P} \text{ data pages}$$

The height of a B+-tree index is thereby

$$height = \left\lceil \frac{\log_2 D}{\log_2 A} \right\rceil$$

The number of comparisons required to locate a tuple with a particular value is:

$$C' = \lceil \log_2 ||R|| \rceil$$

The number of pages which the tree consumes is about

$$S' = D + \left\lceil \frac{D}{A} \right\rceil + \left\lceil \frac{1}{A} \right\rceil \left\lceil \frac{D}{A} \right\rceil + \left\lceil \frac{1}{A} \right\rceil \left\lceil \frac{1}{A} \right\rceil \left\lceil \frac{D}{A} \right\rceil + \cdots = D \sum_{r=0}^{\infty} \left\lceil \frac{1}{A} \right\rceil^r$$

To a first approximation S' is

$$S' = D * \frac{A}{A-1}$$

Again the number of page faults is approximately

---

paged binary trees are not balanced and the worst case access time may be significantly poorer than in the case of a B-tree.

$$faults = (height + 1) * (1 - \frac{|M|}{S'})$$

As a result the cost of a B+-tree access by key is:

$$cost(B+-tree) = Z*(height + 1)*(1 - \frac{|M|}{S'}) + C'$$

An AVL-Tree will be the preferred structure for case 1 if

$$\text{DIFF} = \text{cost(B+-tree)} - \text{cost(AVL-Tree)} > 0$$

If we assume that $C = C' = \log_2||R||$ and rearrange the terms in the inequality, then an AVL-Tree will be preferred if:

$$(1-Y)*\log_2||R|| > Z*\log_2||R||*(1-\frac{|M|}{S}) - Z*(height + 1)*(1-\frac{|M|}{S'})$$

Note that if L $>>$ 8 then S $\simeq$ 0.69 * S'. Define $H = \frac{height + 1}{\log_2||R||}$. Some simplification yields:

$$\frac{|M|}{S} > \frac{Z*(1-H) + Y - 1}{Z} * \frac{1-H}{1.45}$$

Obviously, if $|M| > S$, then AVL trees are the preferred structure regardless of the values of H, Y, and Z. In this situation, the entire AVL-Tree is resident in main memory and there are no disk accesses. Since both data structures require the same number of comparisons and the AVL comparisons are cheapers, then the AVL-Tree is guaranteed to have lower cost. If $|M| < S$ then AVL trees will be preferred if the value of $\frac{|M|}{S}$ is larger than the value of min(|M|/S) shown as in Table 1. As can be seen, essentially all of a relation has to be resident in main memory before an AVL tree is the preferred structure. For rea-

Table 1 – Minimum Residency Factor For Random Access

Z	Y	H	min (|M|/S)
10	.5	.1	.91
10	.5	.2	.87
10	.5	.3	.82
10	.75	.1	.94
10	.75	.2	.90
10	.75	.3	.86
15	.75	.1	.96
15	.75	.2	.91
15	.75	.3	.86

sonable values of H, Y and Z, at least 80 percent and sometimes more than 90 percent of a relation must be main memory resident.

We turn now to sequential access. For an AVL-Tree, the cost of reading N records sequentially is N comparisons and N page reads, i.e.:

$$seq\text{-}cost(AVL) = Y*N + N*Z*(1-\frac{|M|}{S})$$

On the other hand, N records in a B+-Tree will occupy

$$Q = \left\lceil \frac{(N*L)/(1.38*P)}{L/(.69*P)} \right\rceil \text{ data pages}$$

and consequently:

$$seq\text{-}cost(B+\text{-}Tree) = N + Q*Z*(1-\frac{|M|}{S'})$$

An AVL-Tree will be preferred if:

$$\frac{|M|}{S} > \frac{Z(1-H')+(Y-1)}{Z*(1-H'/1.45)}$$

where $H' = \frac{Q}{N}$. It appears that reasonable values for H' are similar to reasonable values for H; hence, Table 1 also applies to sequential access.

In both random and sequential access, a very high percentage of the tree must be in main memory for an AVL-Tree to be competitive. Hence, it is likely to be a structure of limited general utility and B+-Trees will continue to remain the dominant access method for database management systems.

3. Algorithms for Relational Database Operations

3.1. Introduction

In this section we explore the performance of alternative algorithms for relational database operations in an environment with very large amounts of main memory. Since many of the techniques used for executing the relational join operator can also be used for other relational operators (e.g. aggregate functions, cross product, and division), our evaluation efforts have concentrated on the join operation. However, at the end of the section, we discuss how our results extend to these other algorithms.

After introducing the notation used in our analysis, we present an analysis of the familiar sort-merge [BLAS77] join algorithm using this notation. Next we analyze a multipass extension of the

simple hashing algorithm. The third algorithm described is an algorithm that has been proposed by the Japanese 5th generation project [KITS83], and is called GRACE. In the first phase, the join of two large relations is reduced to the join of several small sets of tuples. During the second phase, the tuple sets are joined using a hardware sorter and a sort-merge algorithm. Finally, we present a new algorithm, called the Hybrid algorithm. This algorithm is similar to the GRACE algorithm in that it partitions a join into a set of smaller joins. However, during the second phase, hashing is used instead of sort merge.

In the following sections we develop cost formulas for each of the four algorithms and report the result of analytic simulations of the four algorithms. Our results indicate that that the Hybrid algorithm is preferable to all others over a large range of parameter values.

3.2. Notation and Assumptions

Let R and S be the two relations to be joined. The number of pages in these two relations is denoted |R| and |S|, respectively. The number of tuples in R and S are represented by ||R|| and ||S||. The number of pages of main memory available to perform the join operation is denoted as |M|. Given |M| pages of main memory, $\{M\}_R$, $\{M\}_S$ specify the number of tuples from R and S that can fit in main memory at one time.

We have used the following parameters to characterize the performance of the computer system used:

comp	time to compare keys
hash	time to hash a key
move	time to move a tuple
swap	time to swap two tuples
IO_{SEQ}	time to perform a sequential IO operation
IO_{RAND}	time to execute a random IO operation

To simplify our analysis we have made a number of assumptions. First, we have assumed that $|R| \leq |S|$. Next, several quantities need to be incremented by slight amounts to be accurate. For example, a hash table or a sort structure to hold R requires somewhat more pages than |R|, and finding a key value in a hash table requires, on the average, somewhat more than one probe. We use "F" to denote any and all of these increments, so for example a hash table to hold R will require |R|*F pages. To simplify cost calculations, we have assumed no overlap of CPU and IO processing. We have also ignored the

cost of reading the relations initially and the cost of writing the result of the join to disk since these costs are the same for each algorithm.

In any sorting or hashing algorithm, the implementor must make a decision as to whether the sort structure or hash table will contain entire tuples or only Tuple IDs (TIDs) and perhaps keys. If only TIDs or TID-key pairs are used, there is a significant space savings since fewer bytes need to be manipulated. On the other hand, every time a pair of joined tuples is output, the original tuples must be retrieved. Since these tuples will most likely reside on disk, the cost of the random accesses to retrieve the tuples can exceed the savings of using TIDs if the join produces a large number of tuples. Fortunately, we can avoid making a choice as the decision affects our algorithms only in the values assigned to certain parameters. For example, if only TID-key pairs are used then the parameter measuring the time for a move will be smaller than if entire tuples are manipulated.

Three algorithms (Sort-merge, GRACE, and Hybrid hash) are much easier to describe if they require at most two passes. Hence we assume the necessary condition $\sqrt{|S|*F} \leq |M|$. For example, if F = 1.2, then |M| is only 1,000 pages (4 megabytes at 4K bytes/page), and |S| (and |R|, since $|R| \leq |S|$) can be as large as 800,000 pages (3.2 gigabytes)!

3.3. Partitioning a Relation by Hash Values

If |M| < |R|*F, each of the hashing algorithms described in this paper requires that R and/or S be partitioned into distinct subsets such that any two tuples which hash to the same value lie in the same subset. One such partitioning is into the sets R_x such that R_x contains those tuples r for which h(r) = x. We call such a partition *compatible* with h.

A general way to create a partition of R compatible with h is to partition the set of hash values X that h can assume into subsets, say $X_1,...,X_n$. Then, for i = 1,...,n define R_i to be all tuples r such that h(r) is in X_i. In fact, every partition of R compatible with h can be derived in this general way, beginning with a partition of the hash values. The power of this method is that if we partition both R and S using the same h and the same partition of hash values, say into $R_1,...,R_n$ and $S_1,...,S_n$, then the problem of joining R and S is reduced to the task of joining R_1 with S_1, R_2 with S_2, etc. [BABB79, GOOD81].

In order for the hash table of each set of R tuples to fit in memory, $|R_i| * F$ must be $\leq |M|$. This is not easily guaranteed. For example, how can one choose a partition of R, compatible with h, into two sets of equal size? One might try trial and error: Begin by partitioning the set of hash values into two sets X_1 and X_2 of equal size and then consider the sizes of the two corresponding sets of tuples R_1 and R_2. If the R-sets are not of equal size then one changes the X sets to compensate, check the new R-sets again, etc. Despite the apparent difficulties of selecting the sets X_1, X_2, ..., there are two mitigating circumstances. Suppose that the key distribution has a bounded density and that the hash function effectively randomizes the keys. If the number of keys in each partition is large, then the central limit theorem assures us that the relative variation in the number of keys (and hence the number of tuples) in each partition will be small. Furthermore, if we err slightly we can always apply the hybrid hash join recursively, thereby adding an extra pass for the overflow tuples.

3.4. Sort-Merge Join Algorithm

The standard sort-merge algorithm begins by producing sorted runs of tuples which are on the average twice as long as the number of tuples that can fit into a priority queue in memory [KNUT73]. This requires one pass over each relation. During the second phase, the runs are merged using an n-way merge, where n is as large as possible (since only one output page is needed for each run, n can be equal to |M|-1). If n is less than the number of runs produced by the first phase, more than two phases will be needed. Our assumptions guarantee that only two phases are needed.

The steps of the sort-merge join algorithm are:

(1) Scan S and produce output runs using a selection tree or some other priority queue structure. Do the same for R. A typical run will be approximately $\frac{2*|M|}{F}$ pages long [KNUT73]. Since the runs of R have an average length of $\frac{2*|M|}{F}$ pages, there are $\frac{|R|*F}{2*|M|}$ such runs. Similarly, there are $\frac{|S|*F}{2*|M|}$ runs of S. Since S is the larger relation, the total number of runs is at most $\frac{|S|*F}{|M|}$. Therefore, all the runs can be merged at once if $|M| \geq \frac{|S|*F}{|M|}$, or $|M| \geq \sqrt{|S|*F}$, and we have assumed |M| to be at least $\sqrt{|S|*F}$ pages. Thus all runs can be merged at once.

(2) Allocate one page of memory for buffer space for each run of R and S. Merge runs from R and S concurrently. When a tuple from R matches one from S, output the pair.

The cost of this algorithm (ignoring the cost of reading the relations initially and the cost of writing the result of the join) is:

$$(||R||\log_2\frac{\{M\}_R}{F} + ||S||\log_2\frac{\{M\}_S}{F}) * (\text{comp} + \text{swap})$$ Insert tuples into priority queue to form initial runs

$$+ (|R| + |S|) * IO_{SEQ}$$ write initial runs

$$+ (|R| + |S|) * IO_{RAND}$$ Reread initial runs

$$+ (||R||\log_2\frac{||R||}{\{M\}_R/F} + ||S||\log_2\frac{||S||}{\{M\}_S/F}) * (\text{comp} + \text{swap})$$ Insert tuples into priority queue for final merge

$$+ (||R|| + ||S||) * \text{comp}$$ Join results of final merge.

This cost formula holds only if a tuple from R does not join with more than a page of tuples from S.

3.5. Simple-Hash Join Algorithm

If a hash table containing all of R fits into memory, i.e. if $|R| * F \leq |M|$, the simple-hash join algorithm proceeds as follows: build a hash table for R in memory and then scan S, hashing each tuple of S and checking for a match with R (to obtain reasonable performance the hash table for R should contain at least TID-key pairs). If the hash table for R will not fit in memory, the simple-hash join algorithm fills memory with a hash table for part of R, then scans S against that hash table, then it continues with another part of R, scans the remainder of S again, etc.

The steps of the simple-hash join algorithm are:

(1) Let P = min(|M|, |R|*F). Choose a hash function h and a range of hash values so that $\frac{P}{F}$ pages of R-tuples will hash into that range. Scan the (smaller) relation R and consider each tuple. If the tuple hashes into the chosen range, insert the tuple into a P-page hash table in memory. Otherwise, write the tuple into a new file on disk.

(2) Scan the larger relation S and consider each tuple. If the tuple hashes into the chosen range, check the hash table of R-tuples in memory for a match and output the pair if a match occurs. Otherwise, write the tuple to disk. Note that if key values of the two relations are distributed similarly, there will be $\frac{P}{F} * \frac{|S|}{|R|}$ pages of the larger relation S processed in this pass.

(3) Repeat steps (1) and (2), replacing each of the relations R and S by the set of tuples from R and S that were "passed over" and written to disk in the previous pass. The algorithm ends when no tuples from R are passed over.

The algorithm requires $\left\lceil \frac{|R|*F}{|M|} \right\rceil$ passes to execute. We denote this quantity by A. Also note that on the ith pass, i = 1, ..., A-1, $||R|| - i * \frac{\{M\}_R}{F}$ tuples of R are passed over. The cost of the algorithm is:

$\|\|R\|\|$ * (hash + move)	Place each tuple of R in a hash table
+ $\|\|S\|\|$ * (hash + comp*F)	Check a tuple of S for a match.
$+ ((A-1)*\|\|R\|\| - \frac{A*(A-1)}{2} * \frac{\{M\}_R}{F})$ * (hash+ move)	Hash and move passed-over tuples in R.
$+ ((A-1)*\|\|S\|\| - \frac{A*(A-1)}{2} * \frac{\{M\}_S}{F})$ * (hash+ move)	Hash and move passed-over tuples in S.
$+ ((A-1)*\|R\| - \frac{A*(A-1)}{2} * \frac{\|M\|}{F}) * 2*IO_{SEQ}$	Write and read passed-over tuples in R.
$+ ((A-1)*\|S\| - \frac{A*(A-1)}{2} * \frac{\|M\|}{F} * \frac{\|S\|}{\|R\|}) * 2 * IO_{SEQ}$	Write and read passed-over tuples in S

3.6. GRACE-Hash Join Algorithm

As outlined in [KITS83], the GRACE-hash join algorithm executes as two phases. The first phase begins by choosing an h and partitioning the set of hash values for h into |M| sets, corresponding to a partition of R and S into |M| sets each, such that R is partitioned into sets of approximately equal size. No assumptions are made about set sizes in the partition of S. The algorithm uses one page of main memory as an output buffer for each of the |M| sets in the partition of R and S. During the second phase of the algorithm, the join is performed using a hardware sorter to execute a sort-merge algorithm on each pair of sets in the partition. To provide a fair comparison between the different algorithms, we have used hashing to perform the join during the second phase. The algorithm proceeds as follows:

(1) Scan R. Using h, hash each tuple and place in the appropriate output buffer. When an output buffer fills, it is written to disk. After R has been completely scanned, flush all output buffers to disk.

(2) Scan S. Using h, hash each tuple and place in the appropriate output buffer. When an output buffer fills, it is written to disk. After S has been completely scanned, flush all output buffers to disk.

Steps (3) and (4) below are repeated for each set R_i, $1 \leq i \leq |M|$, in the partition for R, and its corresponding set S_i.

(3) Read R_i into memory and build a hash table for it.

We pause to check that a hash table for R_i can fit in memory. Assuming that all the sets R_i are of equal size, since there are |M| of them, $|R_i|$ will equal $\frac{|R|}{|M|}$ pages. The inequality $|R_i| * F \leq |M|$ is equivalent to $\sqrt{|R| * F} \leq |M|$, and we have assumed that $\sqrt{|S| * F} \leq |M|$.

(4) Hash each tuple of S_i with the same hash function used to build the hash table in (3). Probe for a match. If there is one, output the result tuple, otherwise proceed with then next tuple of S_i.

The cost of this algorithm is:

(||R|| + ||S||) * (hash + move)	Hash tuple and move to output buffer
+ (|R| + |S|) * IO_{RAND}	Write partitioned relations to disk
+ (|R| + |S|) * IO_{SEQ}	Read partitioned sets
+ ||R|| * (hash + move)	Build hash tables in memory
+ ||S|| * (hash + comp*F)	Probe for a match

3.7. Hybrid-Hash Join Algorithm

In our hybrid-hash algorithm, we use the large main memory to minimize disk traffic. On the first pass, instead of using all of memory as a buffer as is done in the GRACE algorithm, only as many pages (B, defined below) as are necessary to partition R into sets that can fit in memory are used. The rest of memory is used for a hash table that is processed at the same time that R and S are being partitioned.

Let $B = max(0, \frac{|R| * F - |M|}{|M| - 1})$. There will be B+1 steps in the hybrid-hash algorithm. First, choose a hash function h and partition R into $R_0, \cdots, R_B$, such that a hash table for R_0 has $|M| - B$ pages, and $R_1, \ldots, R_B$ are of equal size.

Before describing the algorithm we first show that a hash table for R_i will fit into memory. Assuming that all sets R_i are of equal size, we denote $|R_i|$ by p. We must show that:

$$p * F \leq |M| \qquad \text{(a)}$$

Since R_0 is chosen so that a hash table for it fits into $|M| - B$ pages of memory, we have:

$$|R_0| * F = |M| - B \qquad \text{(b)}$$

Since the sum of all the R_i-sets is R, we have

$$|R| = B * p + |R_0| \qquad \text{(c)}$$

If a hash table for all of R fits into memory, we can choose B = 0 and be done with it. So henceforth we assume $|M| < |R| * F$. Thus, $B = \frac{|R| * F - |M|}{|M| - 1}$. If we solve (c) for p and substitute (b) in the result we get:

$$p = \frac{|R|}{B} - \frac{|R_0|}{B} = \frac{|R|}{B} - \frac{|M| - B}{F * B} \qquad \text{(d)}$$

Now we multiply (d) by F and simplify to get:

$$p * F = \frac{|R| * F - |M|}{B} + 1 \qquad \text{(e)}$$

Finally, we substitute for B in (e) to get (a), which was our goal. Thus we have demonstrated that a hash table for R_i fits into memory.

Now we continue with the algorithm. Allocate B pages of memory to output buffer space, and assign the other $|M| - B$ pages of memory to a hash table for R_0. We pause again to check that there are enough pages in memory to hold the output buffers, i.e. that $B \leq |M|$. If we substitute for B in the inequality $B \leq |M|$ and simplify, we get $\sqrt{|R| * F} \leq |M|$, which is true since we have assumed that that $\sqrt{|S| * F} \leq |M|$.

The steps of the hybrid-hash algorithm are:

(1) Assign the ith output buffer page to R_i *for* $i = 1,...,B$. Scan R. Hash each tuple with h. If it belongs to R_0, place it in memory in the hash table for R_0. Otherwise it belongs to R_i for some $i > 0$, so move it to the ith output buffer page. When this step has finished, we have a hash table for R_0 in memory, and $R_1,...,R_B$ are on disk. The partition of R corresponds to a partition of S compatible with h, into sets $S_0,...,S_B$.

(2) Assign the ith output buffer page to S_i *for* $i = 1,...,B$. Scan S, hashing each tuple with h. If the tuple is in S_0, probe the hash table in memory for a match. If there is a match, output the the result tuple. If there is no match, toss the tuple. Otherwise, the hashed tuple belongs to S_i for some $i > 0$, so move it to the ith output buffer page. Now $R_1,...,R_B$ and $S_1,...,S_B$ are on disk.

Repeat steps (3) and (4) for i = 1,...,B.

(3) Read R_i and build a hash table for it in memory. We have already shown that a hash table for it will fit in memory.

(4) Scan S_1, hashing each tuple, and probing the hash table for R_1, which is in memory. If there is a match, output the result tuple, otherwise toss the S tuple.

For the cost computation, denote by q the quotient $\frac{|R_0|}{|R|}$, namely the fraction of R represented by R_0. To calculate the cost of this join we need to know the size of S_0, and we estimate it to be q*|S|. Then the fraction of R and S sets remaining on the disk is 1-q. The cost of the hybrid-hash join is:

(||R|| + ||S||)*hash	Partition R and S
+ (||R||+ ||S||)*(1-q)*move	Move tuples to output buffers
+ (|R|+ |S|)*(1-q)*IO_{RAND}	Write from output buffers
+ (||R||+ ||S||)*(1-q)*hash	Build hash tables for R and find where to probe for S
+ ||S||*F*comp	Probe for each tuple of S
+ ||R||*move	Move tuples to hash tables for R
+ (|R|+ |S|)*(1-q)*IO_{SEQ}	Read sets into memory

3.8. Comparison of the 4 Join Algorithms

In Figure 1 we have displayed the relative performance of the four join algorithms. The vertical axis is execution time in seconds. The horizontal axis is the ratio of $\frac{|M|}{|R|*F}$. Note that above a ratio of 1.0 all algorithms have the same execution time as at 1.0, except that sort-merge will improve to approximately 900 seconds, since fewer IO operations are needed. The parameter settings used are shown in Table 2. We have assumed that there are at least $\sqrt{|S|*F}$ pages in memory. For the values specified in Table 2, this corresponds to $\frac{|M|}{|R|*F} = 0.009$.

In generating these graphs we have used the cost formulas given above with one exception. The IO_{RAND} term used in the cost formula for hybrid hash should be replaced by IO_{SEQ} in the case that there is only one output buffer. There is only one output buffer whenever $|M| \geq \frac{|R|*F}{2}$ (0.5 on the horizontal axis of Figure 1). The abrupt discontinuity in the performance of the hybrid hash algorithm at 0.5 occurs because when memory space decreases slightly, changing the number of output buffers from one to two, the IO time is suddenly calculated as a multiple of IO_{RAND} instead of IO_{SEQ}. Even when there

Table 2 -- Parameter Settings Used

comp	time to compare keys	3 microseconds
hash	time to hash a key	9 microseconds
move	time to move a tuple	20 microseconds
swap	time to swap two tuples	60 microseconds
IO_{SEQ}	sequential IO operation time	10 milliseconds
IO_{RAND}	random IO operation time	25 milliseconds
F	universal "fudge" factor	1.2
|S|	size of S relation	10,000 pages
|R|	size of R relation	10,000 pages
||R||/|R|	number of R tuples/page	40
||S||/|S|	number of S tuples/page	40

are only two or three buffers, IO_{RAND} is probably too large a figure to use to measure IO cost, but we have not made that change. This is what causes our graphs to indicate that simple hash will outperform hybrid hash in a small region; in practice hybrid hash will probably always outperform simple hash.

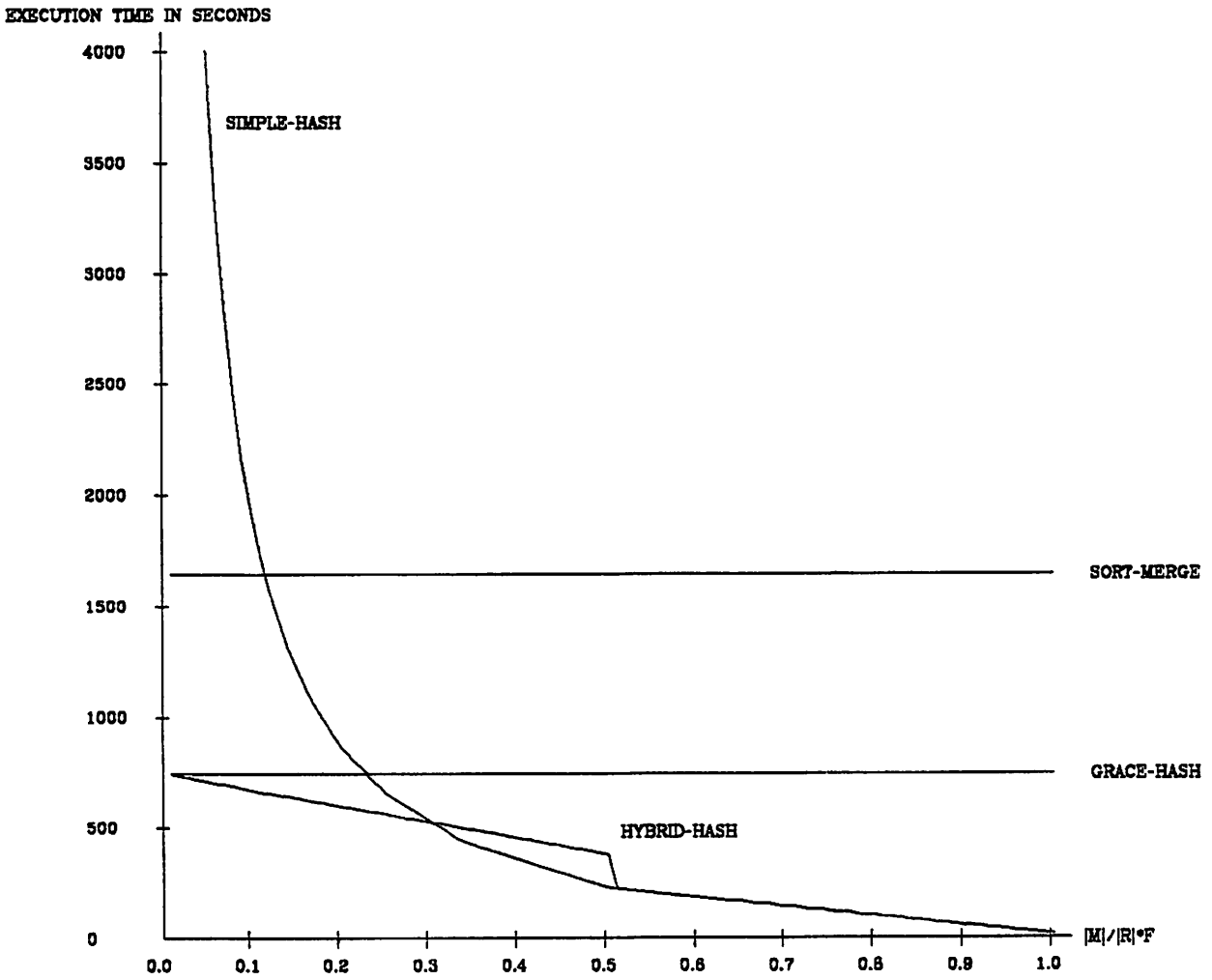
We have generated similar graphs for the range of parameter values shown in Table 3. For each of these values we observed the same qualitative shape and relative positioning of the different algorithms as shown in Figure 1. Thus our conclusions do not appear to depend on the particular parameter values that we have chosen.

3.9. Algorithms for Other Relational Operations

While we have not analyzed algorithms for the remaining relational operations such as aggregate function and projection with duplication elimination, we can offer the following observations. For aggregate functions in which related tuples must be grouped together (compute average employee salary

Table 3 -- Other Parameter Settings Tested

comp	1 to 10 microseconds
hash	2 to 50 microseconds
move	10 to 50 microseconds
swap	20 to 250 microseconds
IO_{SEQ}	5 to 10 milliseconds
IO_{RAND}	15 to 35 milliseconds
F	1.0 to 1.4
|S|	10,000 to 200,000 pages
||R||	100,000 to 1,000,000 tuples



PERFORMANCE OF THE 4 JOIN ALGORITHMS

Figure 1

by manager), the result relation will contain one tuple per group. If there is enough memory to hold the result relation, then the fastest algorithm will be a one pass hashing algorithm in which each incoming tuple is hashed on the grouping attribute. If there is not enough memory to hold the result relation (probably a very unlikely event as who would ever want to read even a 4 million byte report), then a variant of the hybrid-hash algorithm described for the join operator appears fastest. This same hybrid-hash algorithm appears to be the algorithm of choice for the projection operator as projection with duplicate elimination is very similar in nature to the aggregate function operation (in projection we are grouping identical tuples while in an aggregate function operation we are grouping tuples with an identical partitioning attribute).

4. Access Planning and Query Optimization

In the classic paper on access path selection by Selinger [SELI79], techniques are developed by choosing the "best" processing strategy for a query. "Best" is defined to be the plan that minimizes the function W*|CPU| + |I/O| where |CPU| is the amount of CPU time consumed by a plan, |I/O| is the number of I/O operations required for a plan, and W is a weighting factor between CPU and I/O resources. Choosing a "best" plan involves enumerating all possible "interesting" orderings of the operators in query, all alternative algorithms for each operator, and all alternative access paths. The process is complicated by the fact the order in which tuples are produced by an operator can have a significant effect on the execution time of the subsequent operator in the query tree.

The analysis presented in Section 3 indicates that algorithms based on hashing (the hybrid-hash algorithm in the case of the join operator and the simple-hash algorithm to process projection and aggregate function operators) are the fastest algorithms when a large amount of primary memory is available. Since the performance of these algorithms is not affected by the input order of the tuples and since there is only one algorithm to choose from, query optimization is reduced to simply ordering the operators so that the most selective operations are pushed towards the bottom of the query tree.

5. Recovery in Large Memory Databases

5.1. Introduction and Assumptions

High transaction processing rates can be obtained on a processor with a large amount of main memory, since input/output delays can be significantly reduced by keeping the database resident in memory. For example, if the entire database is resident in memory, a transaction would never need to access data pages on disk.

However, keeping a large portion of the database in volatile memory presents some unique challenges to the recovery subsystem. The in-memory version of the database may differ significantly from its latest snapshot on disk. A simple recovery scheme would proceed by first reloading the snapshot on disk, and then applying the transaction log to bring it up to date. Unless the recovery system does more than simple logging during normal transaction processing, recovery times would become intolerably long using this approach.

Throughout this section, we will assume that the entire database fits in main memory. In such an environment, we need only be concerned with log writes. A "typical" transaction writes 400 bytes of log data (40 bytes for transaction begin/end, 360 bytes for old values/new values),[2] which takes 10 ms (time to write one 4096 byte page without a disk seek). We also assume that a small portion of memory can be made stable by providing it with a back-up battery power supply.

5.2. Limits to Transaction Throughput

In conventional logging schemes, a transaction cannot commit until its log commit record has been written to stable storage. Most transactions have very simple application logic, and perform three to four page reads and writes. While transactions no longer need to read or write data pages if the database is memory resident, they still need to perform at least one log I/O. Assuming a single log device, the system could commit at most 100 transactions per second (1 second / 10 ms per commit = 100 committed transactions per second). The time to write the log becomes the major bottleneck.

A scheme that amortizes this log I/O across several transactions is based on the notion of a

[2] These are ballpark estimates, based on the example banking database and transactions in Jim Gray, "Notes on Database Operating Systems," IBM RJ2188(30001), (February 23, 1978).

pre-committed transaction. When a transaction is ready to complete, the transaction management system places its commit record in the log buffer. The transaction releases all locks *without waiting for the commit record to be written to disk.* The transaction is delayed from committing until its commit record actually appears on disk. The "user" is not notified that the transaction has committed until this event has occurred.[3]

By releasing its locks before it commits, other transactions can read the pre-committed transaction's dirty data. Call these *dependent transactions.* Reading uncommitted data in this way does not lead to an inconsistent state as long as the pre-committed transaction actually commits *before* its dependent transactions. A pre-committed transaction does not commit only if the system crashes, never because of a user or system induced abort. As long as records are sequentially added to the log, and the pages of the log buffer are written to disk in sequence, a pre-committed transaction will have its commit record on disk before its dependent transactions.

The transactions with commit records on the same log page are committed as a group, and are called the *commit group.* A single log I/O is incurred to commit all transactions within the group. The size of a commit group depends on how many transactions can fit their logs within a unit of log write (i.e., a log buffer page). Assuming the "typical" transaction, we could have up to ten transactions per commit group. The transaction throughput can be increased by another order of magnitude, to 1000 transactions per second (1 second / 10 ms to commit 10 transactions = 1000 transactions committed per second).

The throughput can be further increased by writing more than one log page at a time, by partitioning the log across several devices. Since more than one log I/O can be active simultaneously, the recovery system must maintain a topological ordering among the log pages, so the commit record of a dependent transaction is not written to disk before the commit record of its associated pre-committed transaction. The roots of the topological lattice can be written to disk simultaneously.

To maintain the ordering, and thus the serialization of the transactions, the lock table of the concurrency control component must be extended. Associated with each lock are three sets of transactions: active transactions that currently hold the lock, transactions that are waiting to be granted the

[3] The notion of group commits appears to be part of the unwritten database folklore. The System-R implementors claim to have implemented it. To our knowledge, neither the idea nor the implementation details has yet appeared in print.

lock, and pre-committed transactions that have released the lock but have not yet committed. When a transaction is granted a lock, it becomes dependent on the pre-committed transactions that formerly held the lock. The dependency list is maintained in the transaction's descriptor in the active transaction table. When a transaction becomes pre-committed, it moves from the holding list to the pre-committed list for all of its locks (we assume all locks are held until pre-commit), and the committed transactions in its dependency list are removed. In becoming pre-committed, the transaction joins a commit group. The commit groups of the remaining transactions in its dependency list are added to those on which its commit group depends. A commit group cannot be written to disk, and thus commit, until all the groups it depends on have previously been committed.

For recovery processing, a single log is recreated by merging the log fragments, as in a sort-merge. For example, to roll backwards through the log, the most recent log page in each fragment is examined. The page with the most recent timestamp is processed first, it is replaced by the next page in that fragment, and the most recent log page of the group is again determined. By a careful buffering strategy, the reading of log pages from different fragments can be overlapped, thus reducing recovery time.

5.3. Checkpointing the Database

An approach for reducing recovery time is to periodically checkpoint the database to stable storage [GRAY81]. Checkpointing limits recovery activities to those transactions that are active at the checkpoint or who have begun since the last checkpoint. System-R, for example, takes an action consistent checkpoint, during which no storage system operations may be in progress (a transaction consists of several such actions, which correspond roughly to logical reads and writes of the database). Dirty buffer pool pages are forced to disk. Since the database is assumed to be large, a large number of dirty pages will need to be written to disk, making the database unavailable for an intolerably long amount of time. Consider the case of 1000 transactions per second, two dirty pages per transaction, and 30 seconds between checkpoints. In the worst case, 60,000 pages would need to be written at the checkpoint!

We would like to overlap checkpoint with transaction activity. Let Δ_{mem} be the set of pages that have been updated since the last checkpoint. Once a checkpoint begins, transaction activity can continue if updates to pages of Δ_{mem} cause new in-memory versions to be created, leaving the old versions

available to be written to disk. A checkpointed, action consistent state of the database is always maintained on disk. At a checkpoint, a portion of the state is replaced by Δ_{mem}. To guarantee that the state is updated "carefully," we use a batch update approach by first writing these pages to stable storage. We denote the batch update file by Δ_{disk}. If the system crashes while the disk state is being overwritten from memory, it can be reconstructed from the pages in Δ_{disk}.

The algorithm proceeds in two phases. In phase 1, Δ_{mem} is written to Δ_{disk}. During phase 2, the pages in Δ_{mem} are copied to their original locations on disk. For the algorithm to work, we must assume:

(1) Extra disk space is available to hold Δ_{disk}.

(2) Extra memory space is available to hold Δ_{mem}.

(3) No dirty page is ever written to disk except during a checkpoint.

Time stamps are used to determine membership in Δ_{mem}. The timestamp T_{cp} indicates when the current checkpoint began, or is zero if no checkpoint is in progress. When a transaction attempts to update a page, the page's timestamp T_{page} is compared to T_{cp}. If $T_{page} < T_{cp}$ and the page is dirty, a new version of the page is created and the in-core page table points to the new page. The update is applied to the new page. The page's timestamp is updated to reflect the latest modification.

To obtain an action consistent state for the checkpoint, the system is initially quiesced. T_{cp} is set to the current time clock to indicate that a checkpoint has begun. The active transaction list is constructed for later inclusion in the log. Transaction activity can now resume, since the old versions in Δ_{mem} can no longer be updated. Memory pages who are dirty and for which $T_{page} < T_{cp}$ are written to Δ_{disk}. After Δ_{disk} has been created, a *begin checkpoint* record is written to the log with T_{cp} and the list of active transactions, indicating that phase 1 of checkpoint is complete. The pages of Δ_{mem} are then written to their original locations on disk, making the disk state identical to the in-memory state as of T_{cp}. An *end checkpoint* record is written to the log to indicate the completion of phase 2, Δ_{disk} is removed, and T_{cp} is reset.

The advantage of the algorithm is that checkpointing can be done in parallel with transaction activity while maintaining an action consistent state on disk. This is particularly needed in a high update

transaction environment, which can generate a large number of updated pages between checkpoints. Further, as soon as a checkpoint completes, another can commence with only a negligible interruption of service. Checkpointing proceeds at the maximum rate possible, i.e., as fast as pages can be written to disk, thus keeping the log processing time to a minimum during recovery.

5.4. Reducing Log Size

While checkpointing will reduce the time to process the log, by reducing the necessary redo activity, it does not help reduce the log size. The large amount of real memory available to us can be used to reduce the log size, if we assume that a portion of memory can be made stable against system power failures. For example, batteries can be used as a back-up power supply for low power CMOS memory chips. We further assume that such memory is too expensive to be used for all of real memory.

Partition real memory into a stable portion and a conventional, non-stable portion. The stable portion will be used to hold an *in-memory log,* which can be viewed as a reliable disk output queue for log data. Transactions commit as soon as they write their commit records into the in-memory log. Log data is written to disk as soon as a log buffer page fills up. Given the buffering of the log in memory, it may be more efficient to write the log a track at a time. In addition, multiple log writes can be directed to different log devices without the need for the bookkeeping described above. However, in the steady state, the number of transactions processed per second is still limited by how fast we can empty buffer pages by writing them to disk-based stable storage.

Stable memory does not seem to gain much over the group commit mechanism. However, the log can be significantly compressed while it is buffered in stable memory. The log entries for a particular transaction are of the form

```
Begin Transaction
<Old Value, New Value>
    . . .
<Old Value, New Value>
End Transaction
```

A transaction's space in the log can be significantly reduced if only new values are written to the disk based log (approximately half of the size of the log stores the old values of modified data -- this is only needed if the transaction must be undone). This is advantageous for space management, and also reduces

the recovery time by shortening the log.

In the conventional approach, log entries for all transactions are intermixed in the log. The log manager maintains a list of committed transactions, and removes their old value entries from log pages before writing them to disk. A transaction is removed from the list as soon as its commit record has been written to disk. A more space efficient alternative is to maintain the log on a per transaction basis in the stable memory. If enough space can be set aside to accommodate the logs of all active transactions, then only new values of committed transactions are ever written to disk.

Stable memory also assists in reducing the recovery time. To recover committed updates to pages since the last checkpoint, the recovery system needs to find the log entry of the oldest update that refers to an uncheckpointed page. A table can be placed in stable memory to record which pages have been updated since their last checkpoint, and the log record id of the first operation that updated the page. When a page is checkpointed to disk, its update status is reset. The log record id of the next update on the page is entered into the table. The oldest entry in the table determines the point in the log from which recovery should commence.

6. Conclusions and Future Research

In this paper we have examined changes to the organization of relational database management systems to permit effective utilization of very large main memories. We have shown that the B+-tree access method will remain the preferred access method for keyed access to tuples in a relation unless more than 80% - 90% of the database can be kept in main memory. We have also evaluated alternative algorithms for complex relational operators. We have shown that once the size of main memory exceeds the square root of the size of the relations being processed, that the fastest algorithms for the join, projection, and aggregate operators are based on a hashing. It is interesting to note that this result also holds for "small" main memories and "small" databases. Finally, we have discussed recovery techniques for memory-resident databases.

There appear to be a number of promising areas for future research. These include buffer management strategies (how to efficiently manage very large buffer pools), the effect of virtual memory on query processing algorithms, and concurrency control. While locking is generally accepted to the algo-

rithm of choice for disk resident databases, a versioning mechanism [REED83] may provide superior performance for memory resident systems.

7. References

[BABB79] Babb, E. "Implementing a Relational Database by Means of Specialized Hardware," ACM TODS, Vol. 4, No. 1, March 1979.

[BLAS77] Blasgen, M.W. and K.P. Eswaran, "Storage and Access in Relational Databases," IBM Systems Journal, Vol. 16, No. 4, 1977.

[CESA82] Cesarini, F. and G. Soda, "Binary Trees Paging", Information Systems, Vol. 7, No. 4, pp 337-344, 1982.

[COME79] Comer, D., "The Ubiquitous B-tree," ACM Computing Surveys, Vol. 11, No. 2, June 1979.

[DATE82] Date, C.J., "An Introduction to Database Systems," Third Edition, Addison-Wesley, 1982.

[GOOD81] Goodman, J. R., "An Investigation of Multiprocessor Structures and Algorithms for Data Base Management," Electronics Research Laboratory Memorandum No. UCB/ERL M81/33, University of California, Berkeley, May 1981.

[GRAY81] Gray, J., et. al., "The Recovery Manager of the System R Database Manager," ACM Computing Surveys, Vol. 13, No. 2, June 1981.

[KNUT73] Knuth, D., "The Art of Computer Programming: Sorting and Searching," Volume III, Addison-Wesley, Reading, MA, 1973.

[KITS83] Kitsuregawa, M. et al, "Application of Hash to Data Base Machine and its Architecture", New Generation Computing, No. 1, 1983, 62-74.

[MUNT70] Muntz, R. and R. Uzgalis, "Dynamic Storage Allocation for Binary Search Trees in a Two-Level Memory," Proceedings of the Princeton Conference on Information Sciences and Systems, No. 4, pp. 345-349, 1970.

[REED83] Reed, D., "Implementing Atomic Actions on Decentralized Data," ACM Transactions on Computer Systems, V 1, N 1, (March 1983).

[SELI79] Selinger, P.G., et. al., "Access Path Selection in a Relational DBMS," Proceedings of the 1979 SIGMOD Conference on the Management of Data, June 1979.

[YAO78] Yao, S. B., and D. DeJong, "Evaluation of Database Access Paths," Proceedings of the 1978 SIGMOD Conference on the Management of Data, May 1978.