HASHING METHODS AND RELATIONAL ALGEBRA OPERATIONS

Kjell Bratbergsengen

Department of Computer Science, Norwegian Institute of Technology
University of Trondheim, N-7034 Trondheim-NTH, Norway

Abstract

This paper present algorithms for relational algebra and set operations based on hashing. Execution times are computed and performance is compared to standard methods based on nested loop and sort-merge. The algorithms are intended for use on a monoprocessor computer with standard disks for data base storage. It is indicated however that hashing methods are well suited to multi processor or especially multi machine database machines. The relational algebra operations described in this paper are under implementation in TECHRA (TECH84), a database system especially designed to meet the needs of technical applications, like CAD systems, utility maps, oil field exploration, etc.

1. INTRODUCTION

The algebra operations we consider in this paper is projection, equi-join, division, union, difference, intersection and aggregate functions. Selection and inequi-joins are not considered. The basic problem of all these operations is finding records with the same "key". In projection, union and difference duplicates are searched for and eventually thrown away. In intersection and join duplicates are kept. And finally in division and aggregate functions, records with equal keys are grouped together.

Efficient execution of relational algebra have stimulated creativaty of many researchers. Inventive architectures have been proposed; RAP (OZKA75), CASSM (SU75, LIPO78), CAFS (MALL79), RELACS (OLIV79), SURE (LEIL78), DIRECT (DEWI79), ASTRA (BRAT80), LUCAS (KRUZ83).

However, we are in favor of the standing taken by DeWitt and Boral in BORA83:

"Our assertion is that highly parallel highperformance database machines are predicated on the availability of mass storage technologies that have not and probably will not emerge as commercially viable products."

Relational database systems should use conventional mass storage technology, and it should be a more fruitful approach to evaluate methods and algorithms for relational algebra and set operations prior to building special purpose hardware. No hardware can make up for poorly designed algorithms. Commercially available systems to day are based on standard disks for database storage, IDM, INGRES, MIMER, ORACLE, ...

It is generally accepted that there is three main algorithms for doing relational algebra operations: nested loop, sort-merge and hashing.

This paper presents three areas for employment of hashing methods: multiple key comparison, problem partitioning and filter techniques. Multiple key comparison is used in nested loop algorithms for algebra operations. Sorting puts data into a higher degree of order than necessary for doing algebra operations. All algebra and set operations except select and in-equi joins turns out to end up in "how to bring together records with some equal key" as the basic problem. For the sole purpose of doing relational algebra, there is no use in having those keys in ascending or descending order. And still sorting seems to be the standard method used in for example INGRES and IDM, see BITT83. Also textbooks state that "Sort merge is best for natural join" and "We conclude that we cannot now improve on sort-merge for calculating the natural join", MERR84 pages 177 and 181.

Proceedings of the Tenth International Conference on Very Large Data Bases.

Singapore, August, 1984

Methods based on hashing have many merits, and seems to be much underrated. One exception is in a recent article written by Valduriez and Gardarin, VALD84. However, they are focusing on multiprocessor database computers, and leaves out the detailed evaluation of hashing methods in a monoprocessor system.

The algorithms discussed in this paper are last resort algorithms, i.e. we are not considering optimization based on indices or preordered relations.

In the following we will develope several timing formulas, and the following notation is used:

A	- first or only operand
B	- second operand, A is always smaller or equal to B
R	- resulting table
nA,nB,nR	- number of records in table A, B and R
n	- number of records in general.
lA,lB,lR	- record length in table A, B and R
l	- record length i general.
VA,VB,VR	- data volume in bytes.
V	- operand volume, in general.
b	- block size in bytes.
s	- number of page in workspace.
ta	- total transport time for one page
tr	- rotational time
tc	- time for comparing two records.
th	- time for computing one hash address
tm	- time to move one record in working storage
te	- time for putting one record into hashed workspace, te=th+tm.
w	- number of split or merge stages.
F	- number of elements in filter.
d	- filter density, no. of 1-elements/F
G	- number of records in a table after filtering.
N	- number of subfiles in sort, partitions in partitioning methods.
M	- workspace, M=s*b. In addition to workspace, we always need one buffer of size b for either input or output.

2. ALGORITHMS BASED ON NESTED LOOP

The basic algorithm for doing relational algebra operations is based on nested loop. Nested loop is efficient when at least one of the operands are small, i.e. not many times larger then the working space. Thruout the paper we will mostly use join to demonstrate and compare the different methods.

The basic algorithm for nested loop join is:

- Read A or as much as possible of A into workspace. Attributes not part of join key or result records are not stored in workspace. To reduce search time in workspace, records are organized into short lists. Which list number is given by a hash formula on the join key.

- For each record in B check if it has the same joinkey as any record stored in workspace. If match; concatenate A and B-records and add result to result relation.

- Repeat this process for all parts of A.

The execution time for nested loop join is:

```
Tnlj = vA/b*ta+nA*te      + (A into hash lists   )
VA/M*(VB/b*ta+nB*(th+tc))+ (Test B on hash lis t)
RV/b*ta + nR*tm             (Write result records)

Tnlj =
     (VA+VR)*ta/b+VA/M*(VB*ta/b+nB*(th+tc))+nR*tm
```

As it can be seen from this formula, execution time is depending on available workspace M=s*b, and page size. Note that one simplification has been done; page transport time is constant, although it increases with page size. The fixed portion of page transfer time is considered dominating. However, the formula clearly tells us: the larger b the better. Double buffering should also be considered.

An important quality of the nested loop algorithm is its ability to exploit unequal operand volumes. This makes it better then sorting methods in many practical situations.

3. SORTING METHODS

Sorting is a safe method, but never the best, except when tables are ordered in advance or when the result have to be sorted on operation key.

The cost of doing sort-merge join is: initial sort and multiway merge of both operands, and finally merging the two operands. The sort process itself can be optimized (see for instance KNUT73 or BRAT73). The following calculations are based on internal sort i.e. quicksort or similar methods for the initial sort phase and fixed s-way merge for the merge phase. Internal sort gives gives initial subfiles of length M (available workspace) rather than an average of 2*M when the reservoir sorting method is employed. However, using the reservoir method on variable length records involves dynamic memory allocation, and the possible gain from longer subfiles is likely to disappear in memory management processing.

$$Tsort = 2*ta*V/b + (tc*\log_2(M/l)+tm)*n + w * (2*V/b + n*(tm + tc*\log_2 s)$$

w is the average number of merge stages, and w is found from the number of initial subfiles, N and the number of available input buffers, s.
N = trunc((V-1)/M) + 1

If N is 1 we have one subfile and the merge phase

is skipped. If we do merge at all, then all data must be read at least once. When N is larger then s, a balanced tree merge pattern gives:

$$w = q - \text{trunc}((s^{**}q-N)/(s-1))/N, \qquad (1)$$
$$q = \text{trunc}(\log_s(N-1)) - 1$$

To summarize:

if N=1	:	w=0.0
if 1 < N <= s	:	w=1.0
if N > s	:	w=w(s,N), see (1)

Tsmj=
TsortA+TsortB+ta*(VA+VB+VR)/b+(nA+nB)*tc+nR*tm

The size of R depends on the selectivity factor.

The problems with sort-merge join is:

1. Initial sort takes one extra pass of all data, and it is CPU-demanding.

2. Sorting never takes advantage of different operand sizes.

4. HASHING USED TO PROBLEM PARTITIONING

Nested loop is the best algorithm when at least one operand is contained in working space. Our strategy now is to partition the larger task into smaller subtasks where the nested loop algorithm can be employed. This is a classic application of the divide and conquer principle, see for instance BENT80.

The problem remaining is problem partitioning. Again hashing comes to help. Two potentially equal joinkey values certainly must give the same hash value. Fig. 1 gives an example of the join operation. The problem is divided into 3 subtasks, using joinkey mod 3 as the hashing algorithm.

The partitioning process requires at least one input buffer and s output buffers where s is the number of partitions. It should be noted that double input buffering probably will pay off. The output buffers should also be kept large because it is important to keep the number of accesses down. Without double buffering, and with a fixed page size b, the partitioning cost will be:

$$Tp = 2*ta*V/b + n*(th+tm)$$

The number of partitions is now limited to s, the number of output buffers.

In general, the number of partitions is:
N = trunc((VA-1)/M) + 1, and there is a trade off between block size, split factor and the number of split stages. The problem is analog to finding the optimum merge pattern in sort merge.

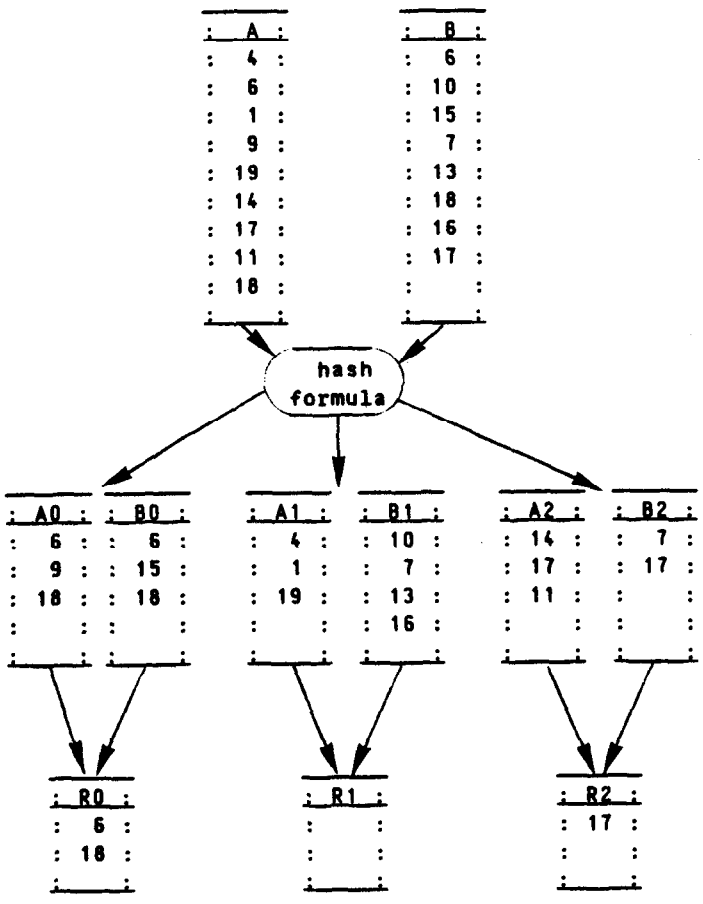


Fig. 1: Table partitioning using hashing on joinkeys. Subtable is found from the hashing formula: subtable := joinkey mod 3.

Comparing sorting and partitioning, without further analysis we can conclude:
1) Partitioning does not require initial sort and thus saves one pass of the data and CPU time, and
2) Computing a hash value should be quicker than selecting the smallest key among s keys as is required during an s-way merge.

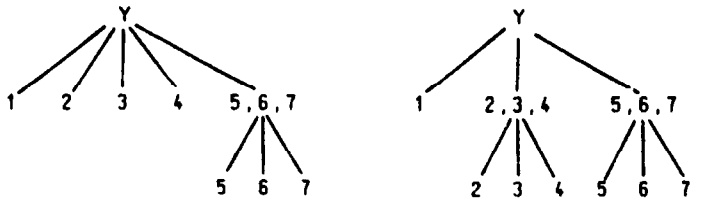
For all two-operand relational set- and algebra operations (except difference) the number of partitions is always given by the smallest operand. This helps a lot when operand sizes differ. Sorting is not able to exploit this, the number of subfiles after initial sort is trunc((V-1)/M)+1) and they have to be merged separatly for both operands.

Finding an optimum split pattern

Assumptions: Input and output buffers have the same size. There is no double buffering or I/O overlapping. The optimum split pattern is a balanced tree of some kind. Block transport time is constant; i.e. not depending on block size.

If the file has to go into 7 subfiles

(partitions), fig. 2 shows two possible patterns:



Alternative 1 Alternative 2

Fig. 2: Alternative split patterns. Assuming the same amount of workspace available in both alternatives, the block size in alternative 1 is less than in alternative. 2.

As for merging the average heigth of a balanced tree is w(s,N), see formula (1).

As for sorting there is some integer logic involved. If the number of partitions; N is one or less, partitioning does not make sense, hence w=0. If partitioning takes place at all, we have to scan all records at least once.

```
if  N <= 1        : w=0
if  1 < N <= s    : w=1
if  N > s         : w=w(s,N), see formula (1)
```

The sheer transportation in partitioning takes:

```
Tt = 2*w*ta*V/b
```

Given a certain workspace M, we can vary the number of output buffers s, to find a value on s which minimizes Tt. A large s will reduce w, but at the same time the number of blocks will increase, and the increased number of page transfers will cost time. We can illustrate these effects by computing the transportation times for the files shown in fig. 2.

	Alternative 1	Alternative 2
Block size	M/5	M/3
w(s,N)	10/7	13/7
Total time	100/7*ta*V/M	78/7*ta*V/M

The larger page size of alternative 2 is best although only 3/7 of the total data volume is scanned twice in alternative 1, and 6/7 of the total volume is scanned twice in alternative 2.

In general we can find the best s by derivation of Tt with respect to s. Substituting for b=M/s and $w=\log_s (V/M)$ we get Tt as a function of s. We use an approximation for w to make Tt a continous function.

```
Tt = (2 * ta * V * ln (V/M) / M) * (s/ln(s))
```

s/ln(s) has its minimum value for s=e=2.71. s must be an integer, and the best value of s is 3. Under the current assumptions the available workspace should always be used as 3 buffers.

Increasing M reduces Tt by reducing the number of partitions, but if M is made larger than 3 tracks the block transportation time ta is no longer constant, but will probably jump to ta' = ta+tr. tr is rotation time. Taking CPU-time into account also tends to increase the optimum number of buffers because CPU-time depends on w, and w decreases with larger s.

Taking all "expenses" into account the time to partition a file takes:

```
Tp = w * (2*ta*V/b + n*(th+tm))
```

The time to do a join or in fact any two-operand operation is:

```
Tphj = w * 2*ta*(VA+VB)/b      + (split A and B)
       w * (nA+nB)*(th+tm)     + (    - " -     )
 (VA+VB)*ta/b+nA*te+nB*tc      + (Nested loop   )
 VR*ta/b+nR*tm                   (Write result  )
```

The size of R depends on selectivity and record length of R-records.

Time for projection is:

```
Tphp =
  w * (2*ta*VA/b + nA*(th+tm))+    (Split A      )
  ta*VA/b + te*nA              +   (Nested loop  )
  ta*VR/b + tm*nR                  (Write result)
```

Aggregate functions

Aggregate functions based on grouping easily lend themselves to partitioning methods. The table is partitioned based on the grouping attribute(s). Then each subtable is handled separatly. A hash-list on the grouping attribute is established, and the actual aggregate function is performed at the same time. The only special problem with aggregate functions is that it is hard to estimate the required workspace necessary to hold the resulting subtable. The maximum size is one record in every group, which in most cases gives far to many groups.

Optimization of disk usage

Sorting requires a certain sequence in reading pages from files. It is possible to prefetch pages, this requires more buffers. Hashing gives more freedom, pages within one bucket (subfile) can be read in any sequence. The prefetch mechanism can be simpler or at least give the same efficiency with fewer buffers.

Problems with variable bucket size

All our calculations so far have not taken into account the problem with variable bucket size. We have an overflow problem similar to the overflow

problem in hashed file allocation. The bucket size is normally very large, several hundred records, and using fill factors 0.7 to 0.9 the overflow percentage will be low. The following table gives the percentage of buckets having more records than bucket size. These values are obtained by simulation.

bucket	fill factor								
size	0.70	0.75	0.80	0.85	0.90	0.95	1.00	1.05	1.10
40	0	2	5	11	22	33	44	57	70
80	0	0	1	5	14	28	46	64	77
160	0	0	1	4	10	24	46	71	85
320	0	0	1	4	7	20	48	73	87

There is another alternative, however: We may operate with larger bucket sizes than s*b, the workspace. On the positive side we can save data transportation during partition, on the negative side we will have to scan the second operand several times during the nested loop phase. The cost of this approach can be analyzed: Let x be the bucket size in number of work space areas. The time to perform a join counting only transportation time is:

$$Tx = (2*wx*(VA+VB) + VA + x*VB + VR) * ta/b$$

$$Nx = VA/(x*s*b) \text{ and } wx = \log_s(Nx)$$

We would like to find a value of x which minimizes Tx. Derivation of Tx on x and equating this to 0 gives:

$$2 * (VA+VB) = x * VB * \ln(s)$$

or $$x = 2 * (VA + VB) / (VB * \ln(s))$$

When VA equal VB and s is small we have:

s	2	3	4	6	8	10
x	6	3	3	2	2	2

For larger s and VB larger than VA, x comes closer to 1 and even lower. However, 1 is the lowest valid value on x.

5. APPLICATION OF HASH FILTERS

Application of hash filters is described in BABB79 and VALD84 among other places. Hash filters are used to eliminate non-candidate records from the rest of the process. Reducing data volume as early as possible in heavy operations might influence the total processing time considerably.

The filter itself is stored as bit vector. Let F be the number of elements in the filter. n is the number of keys in the table for which the filter is built. When the keys are randomly distributed an average of G = F * (1-exp(-n/F) bits are set in the filter. Bit h is set in the filter if h=hash(operationkey).

We will discuss the use of hash filters in two-operand operations, and we will use join as an example. The filter is set up based on the operand having the least number of distinct keys. In practice this can not be known in advance, and the table with the least number of tuples is taken. Then the B-operand is read thru this filter, and B is reduced to B'. The volume of B' is:

$$VB' = VB * (1-\exp(-nA/F)$$

If A is passed more times, a similar filter is made based on B and this is used when reading A. Then VA' = VA * (1-exp(nB/F). This process might be repeated several times, and when a new and independant hash formula is used for every pass, the data volume is reduced correspondingly. The figure 3 shows this for splitting. We assume there is a fixed amount of storage avaiable for the filter. During splitting the tables are reduced by a factor of s for every pass, hence the filter density d=n/F decreases, and d is a function of s and the pass number p, d(s,p) = n/(s**(p-1)*F).

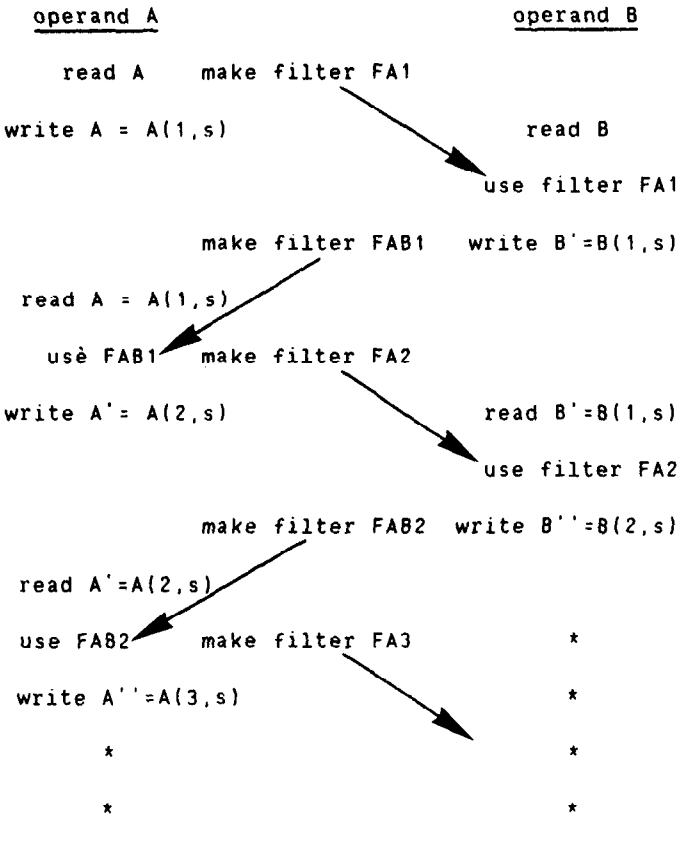


Fig. 3: Application of filters in a two operand relational operation.

After w passes:

```
B(w,s)   =   B   *  (1-exp(-nA/F))
                 *  (1-exp(-nA/(s*F)))
                 *  (1-exp(-nA/(s**2*F)))  *
             .... *  (1-exp(-nA/(s**(w-1)*F)
```

$$B(w,s) = B * \prod_{k=0}^{w-1} 1 - \exp(-nA/(F*s^{**}k))$$

Similarly:

```
A(w,s)   =   A   *  (1-exp(-nB/F))
                 *  (1-exp(-nB/(s*F)))
                 *  (1-exp(-nB/(s**2*F)))  *
             .... *  (1-exp(-nB/(s**(w-2)*F)))
```

$$A(w,s) = A * \prod_{k=0}^{w-2} 1 - \exp(-nB/(F*s^{**}k))$$

Both functions are extremely rapidly decreasing with increasing w and s. The total transportation volume for some values of d and w is tabulated below.

	filter density d								
w	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0
1	3.2	3.3	3.3	3.4	3.5	3.5	3.6	3.6	3.6
2	4.4	4.6	4.8	5.0	5.2	5.4	5.6	5.8	5.9
3	4.3	4.7	5.0	5.3	5.6	5.9	6.2	6.4	6.7
4			5.0	5.3	5.6	5.9	6.2	6.5	6.9
5						5.9	6.2	6.6	6.9

Table 1: Number of table reads or writes using hash filters in two operand operations. w is number of stages, both files have the same size. Each stage requires 4.0 reads or writes without filters.

In this calculation we have not counted for the records actually going into the resulting table. How many records would remain is depending on join selectivity. However, not many stray records will be found in the tables after two or three passes thru the filter.

Optimum filter efficiency.

Given a certain filter space F and a number of keys n, there is a choice of how to use the filter space. Should it be used as one filter with density d = n/F, or should it be partitioned into s filters each of density d = s*n/F. The filtering effect of s different filters of density d(s) is

```
g(s) = (1-exp(-s*n/F)) ** s
```

This function is tabulated below.

	Total filtering effect, 1 to 8 filters							
n/F	1	2	3	4	5	6	7	8
0.1	.10	.03	.02	*.01	.01	.01	.01	.01
0.2	.18	.11	*.09	.09	.10	.12	.14	.16
0.3	.26	*.20	.21	.24	.28	.34	.40	.47
0.4	.33	*.30	.34	.41	.48	.57	.64	.72
0.5	*.39	.40	.47	.56	.65	.74	.81	.86
0.6	*.45	.49	.58	.68	.77	.85	.90	.94
0.7	*.50	.57	.68	.78	.86	.91	.95	.97
0.8	*.55	.64	.75	.85	.91	.95	.97	.99
0.9	*.59	.70	.81	.90	.95	.97	.99	.99
1.0	*.63	.75	.86	.93	.97	.99	.99	1.00

Table 2: Total filtering effect as a function of total filter density n/F, and number of filters. The optimum number of serial filters is starred. For lower densities many serial filters gives the best effect.

For densities d=0.5 or higher, one filter with minimum density is best. For low densities several serial filters is best. The best combinations are starred in table 2.

Hash filters and projection

Hash filters are not easily adapted to projection. Normally rather few records will be removed, and the effect then is small.

Hash filters and sorting

When sorting is used in two-operand algebra operations filters are readily used. Best effect is attained when both filters are established during initial sort phase. Then the B operand is reduced before it enters the initial sort phase. At best the A operand can be reduced after the first merge stage. Further reduction as in partitioning is not feasible.

General usage of filters

Filters should be established as early as possible, and could be used to reduce operands at a very early stage in the process. This is a general technique, and is not discussed further in this paper.

6. COMPARISON OF METHODS

General assumptions

Reading and writing to disk takes the same time, there is no cache memory for the disk. There is no overlapping of IO and processing, although this should be sought in real implementations. We have much higher estimates on CPU consumption than for instance used in DEWI83. All our estimates are based on a monoprocessor computer with

a capacity similar to VAX750.

To test whether two keys are equal takes computing of a hashvalue and performing the actual comparison. The establishing of hashlists involves moving records from input buffer to the hashlist. Records are of variable length which complicates the initial sort phase in the sort merge process. Attributes are not of fixed length only, comparing two records also involves some record format evaluation.

Values used in the following computations:

ta	= 17.0	ms	block transportation time, includes positioning.
te	= 0.5	ms	establish record in hashlist
th	= 0.25	ms	test if record is in hashlist
tc	= 0.25	ms	compare two records in workspace.
tm	= 0.1	ms	move one record
l	= 100	B	number of bytes per record.

Space for filters is not counted with the workspace. The size of the result table depends of selectivity and comes in addition to all times shown in the following comparisons.

The different methods are named:

- NL - nested loop join
- SM - sort merge join
- PH - partition hash join
- PHF - partition hash with filter join.

Two important cases are analyzed: 1) Both operands have the same size (even operands), and 2) Operand B has a volume 9 times larger than A (uneven operands).

Nested loop versus sort-merge

Figures 4 and 5 show join execution times as a function of operand volume for all methods discussed in this paper. As seen in fig.4 the nested loop algorithm is better than any method for small volumes. Another important effect is clearly demonstrated: When one operand becomes much larger than the other; nested loop takes advantage of this, but it is a disadvantage for sort-merge.

Nested loop is hopelessly inadequate for large operand volumes. The nested loop method is much depending on available workspace. This can be seen in fig. 10. Computations indicate that nested loop is better than sort merge when operand volume is less than 4 to 12 times the workspace. This is for equally sized operands. Generally it seems that nested loop is better than sort merge when the smallest operand is less than 3 to 8 times the workspace.

As seen especially in fig. 10 sort-merge is not clever to take advantage of larger workspace. This was indeed unexpected, but the computation time increases with more buffer space, so much that there is almost no gain after a certain size. If we use a faster processor the operation becomes IO-bound and the picture will change.

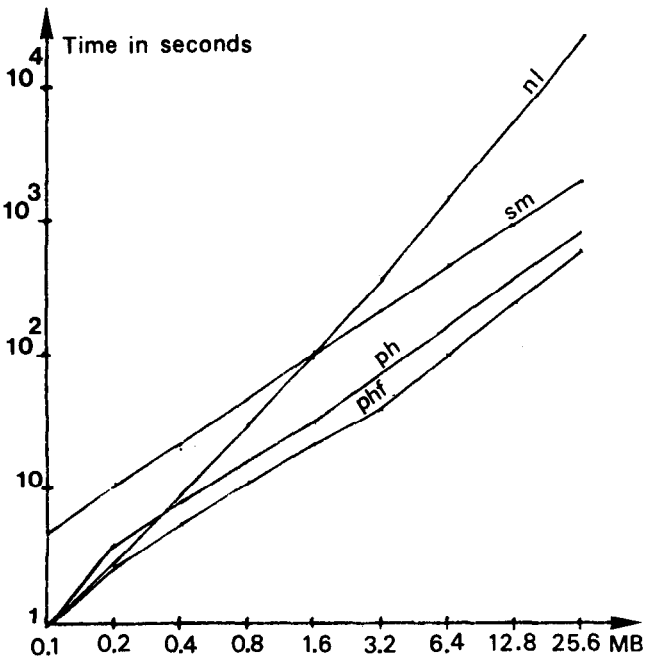


Fig. 4 Operation times as a function of total operand volume. Buffer size is 4K and the number of buffers is 16, giving a total workspace of 64 KB. Even operands.

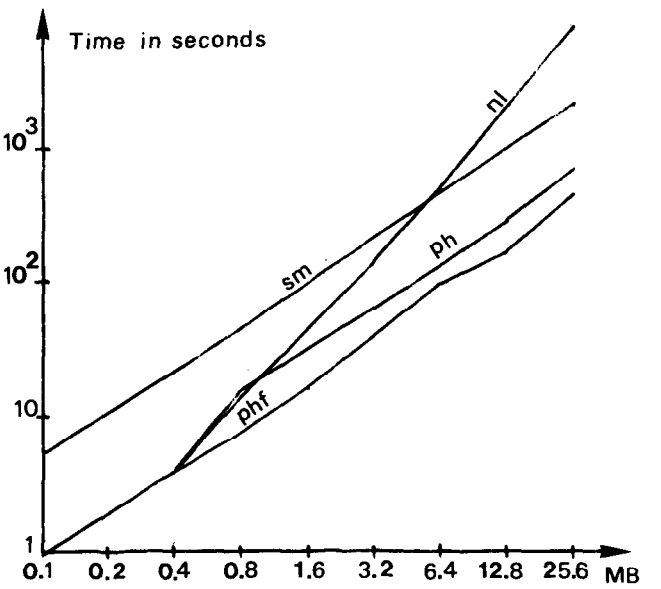


Fig. 5 Operation times as a function of total operand volume. Buffer size is 4K and the number of buffers is 16, giving a total workspace of 64 KB. Uneven operands, VB=9*VA.

Sort merge versus partitioning

Partitioning always performs better than sort-merge. The difference seems to grow with larger workspace, and to shrink with larger operand volumes. This becomes more clear when we look

separately on IO-time and CPU-time. For even sized operands Tsmj is proportional to V(3+2*w) where V=VA+VB, and Tphj is proportional to V(1+2*w). The quotient q=Tsmj/Tphj is (3+2*w)/(1+2*w). w can take values 0, 1 and is continous above 1.0. A brief tabulation of the quotient gives:

w	0	1	2	3	4	5	6
q	3	1.66	1.40	1.29	1.22	1.18	1.15

w have the same value for both operands and also in both methods.

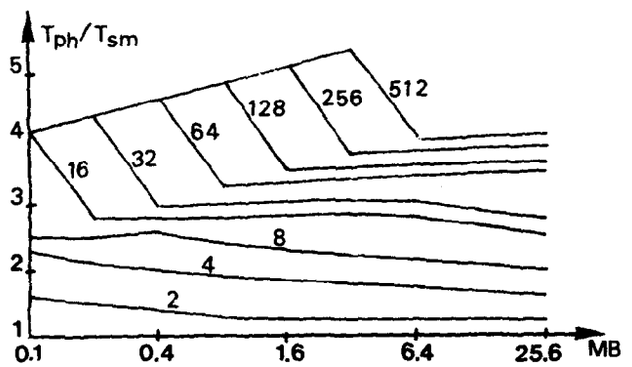


Fig. 6 Partition/sort-merge quotient as a function of workspace and operand volume. Workspace in no. of buffers, varying from 2 to 512. Page size is 4KB. Even operands, total operand size is 6.4 MB.

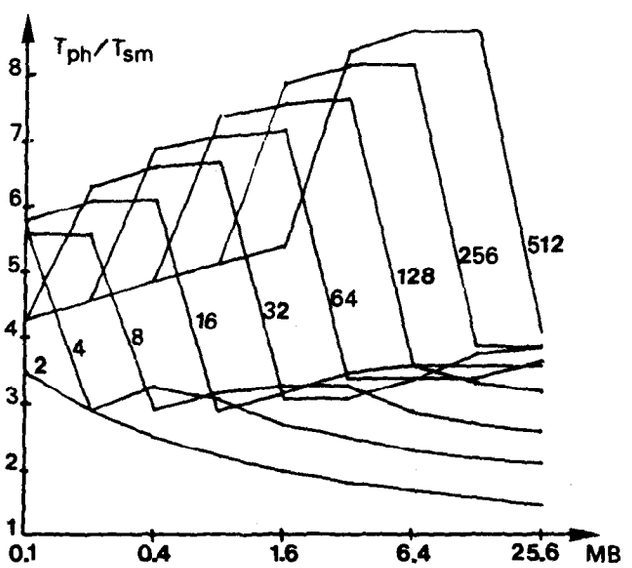


Fig. 7 Partition/sort-merge quotient as a function of workspace and operand volume. Workspace in no. of 4KB buffers, varying from 2 to 512. Uneven operands, total operand volume is 6.4 MB.

CPU-time spent on each record grows with w for both methods. For sorting it also grows with $O(\log_2 M)$ in initial sorting and with $O(\log_2 s)$ during merge. When each comparison takes about 250 microseconds this can add considerably to the processing time. Variable processsing time in a 1MB buffer and 4KB pages amounts to about 5ms per record. Using a large workspace does not add to processing cost for partitioning. On the other side, sorting can even take longer time when more workspace is used. This is seen in fig.10.

Effect of unequal size of operands

Both nested loop and partitioning takes advantage of uneven operands. This is demonstrated in figs. 8 and 9. However, nested loop is much more sensitive to available workspace and uneven operand volumes than partitioning. These effects must be considered in the selection of method for a given task. When even sized operands are joined; nested loop is always chosen when VA is less then 3*M i.e. V<6*M. When operands differ the crossover point moves. For the case shown in fig. 9, nested loop is the preferred method when VA is less than 2*M, i.e. V<20*M as VA=V/10.

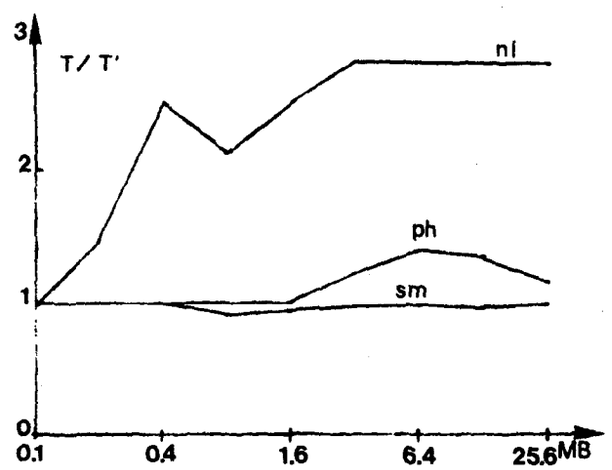


Fig. 8 Improvement factor, operation time for even operands over operation time for uneven operands. When operands are uneven VB=9*VA.

Effect of hash filters

The effect of hash filters is an improvement from 0 to about 4, best effect for few buffers in workspace. The effect is not much affected by differences in operand sizes, the good characteristics of partitioning is retained. For really large volumes the filtering effect is reduced because of high filter density. Instead of using all workspace for page buffers, it should be considered to use more space to reduce filter density. It should be noted that the filtering effect is reduced if selectivity is high, i.e. a large proportion of records will be included in the result table.

Effect of available workspace

Workspace is perhaps the most valuable resource. The nested loop algorithm has a processing time inversely proportional to available workspace.

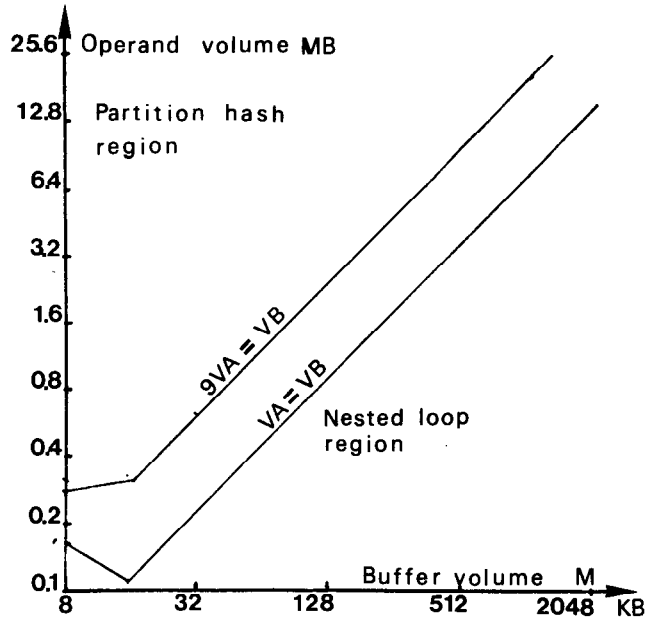


Fig. 9 Choice of method as a function of available workspace and total operand volume and relative operand size.

When one operand get room i workspace, nested loop is 3 times better than both sort-merge and partitioning. Partitioning will always benefit from more workspace. As mentioned already, sort-merge might perform slower with large workspace. This is actally the case in fig. 10. A workspace of 32 buffers gives a minimum processing time. When hash filters are used, more than minimum workspace is virtually vasted.

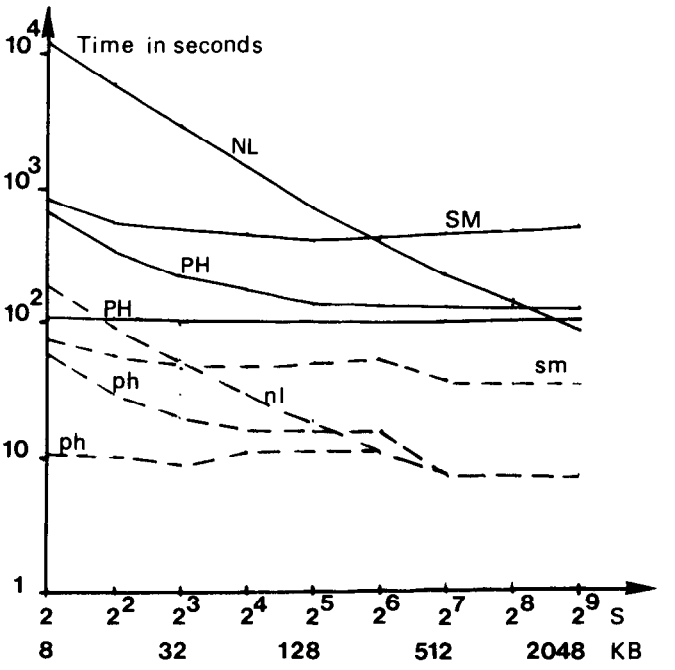


Fig. 10 Processing time as a function of available workspace. Pages are 4 KB, operands are even. Solid lines for a total operand volume of 6.4 MB, dashed lines for 0.8 MB.

Effect of increased page size

Figs. 11 and 12 give examples of the effect of increased page size. In fig. 11 the total work space is kept constant at 128 KB, and the page size is decreased from 64 KB to 0.25 KB. Large pages should be used. However, remember that page transfer time is constant in these computations. This is almost true for pages less or equal to one track. When page size is larger than one

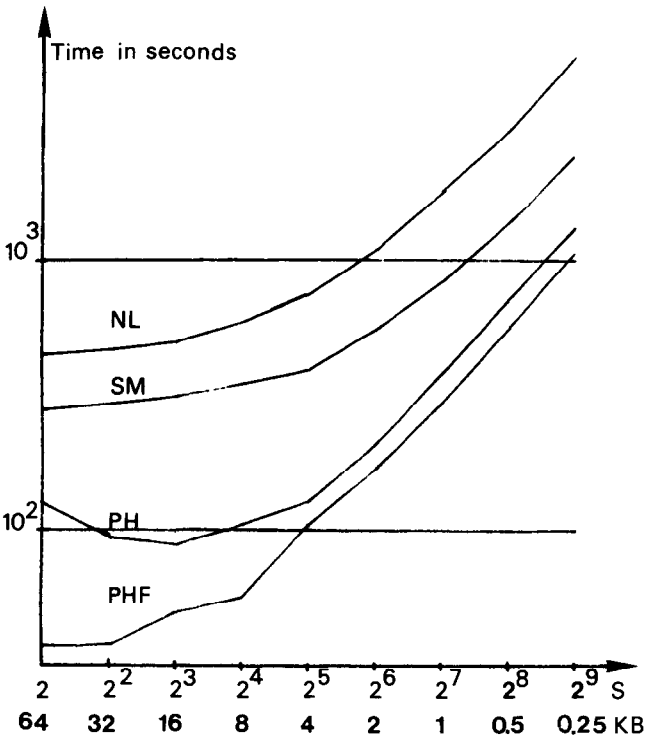


Fig. 11 Effect of increasing pagesize while total workspace is kept constant. Workspace is 128 KB. Total operand volume is 6.4 MB. Even operands.

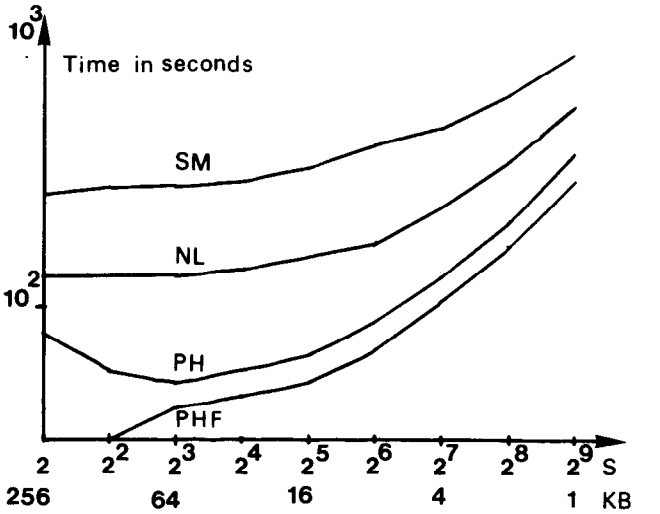


Fig. 12 Effect of increasing pagesize while total workspace is kept constant. Workspace is 512 KB. Total operand volume is 6.4 MB. Even operands.

track page transfer time should be multiplied with number of tracks necessary to store a page. Partitioning has a well defined optimum number of pages, namely 8 pages each 16 KB. This is also a practical page size. Small pages should be avoided.

In fig. 12 workspace is 512KB. Still partitioning have 8 as the optimum number of pages. It should be noted that nested loop is better then sort-merge with this workspace.

7. CONCLUSION

For small volumes nested loop methods are preferable. Partitioning based on hashing is always superior to sorting. Partitioning methods are easily extended onto multicell or parallel systems. Partitioning takes advantage of different operand sizes. Hashing leaves more opportunities for disk transfer optimizations than sorting, because blocks of a subfile might be read or written in arbitrary order. Filters are readily combined with partitioning. Partitioning takes full advantage of large workspace without degrading performance due to internal processing as in sort-merge methods. It is the opininon of the author that hashing and partitioning are grossly underrated compared to sort-merge methods. The methods does not require special purpose hardware, the only exception might be for extremely parallel and high performance systems, for instance 5th generation systems.

REFERENCES

BABB79 E. Babb "Implementing a Relational Data Base by Means of Specialized Hardware" ACM TODS vol 4, no 1 March 1979.

BENT80 J.L. Bentley "Multidimensional Divide-and-Conquer" Communication of the ACM, Vol. 23, No. 4, April 1980.

BITT83 D. Bitton, D.J. DeWitt, C. Turbyfill "Benchmarking Database Systems. A systematic Approach. Proceedings 9-th Conference on VLDB, Florence Oct. 1983.

BITT83B D. Bitton, H. Boral D.J. DeWitt and W.K. Wilkinson "Parallel Algorithms for the Execution of Relational Database Operations" ACM Transactions on Database Systems, Vol. 8, No. 3 Sept. 1983.

BITT83C D. Bitton and D.J. DeWitt "Duplicate Record Elimination in Large Data Files" ACM Transactions on Database Systems, Vol. 8, No. 2 June 1983.

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

BORA83 H. Boral and D.J. DeWitt "Database Machines: An Idea Whose Time is Past? A Critique of the Future of Database Machines" Database Machines, Springer-Verlag 1983, ed. H.-O. Leilich and M. Missikoff. Proceedings International Workshop, Munich, Sept. 1983.

BRAT73 K. Bratbergsengen :"Sortering av store datamengder", Institutt for databehandling, 1973. (In Norwegian)

BRAT80 K. Bratbergsengen :"A Neighbor Connected Processor Network for Performing Relational Algebra Operations". 5th Workshop on Computer Architecture for Non Numeric Processing. Monterey CA, March 1980

BRAT81 K. Bratbergsengen: "Design of a VLSI-chip for Moving Data in a Hypercube Network", Technical Notes, Dept. of Computer Science, Norwegian institute of Technology, 1981.

BRAT83 K. Bratbergsengen:" The Hypercube as a Line Switching Network", Technical Notes, Dept. of Computer Science, Norwegian Institute of Technology, 1981.

DEWI79 D.J. DeWitt "DIRECT - A Multiprocessor Organization for Supporting Relational Database Management Systems" IEEE Transactions on Computers, vol C-28, no 6 , June 1979.

DEWI79 D.J. DeWitt "DIRECT - A Multiprocessor Organization for Supporting Relational Database Management Systems" IEEE Transaction of Computers, June 1979.

DEWI81 D.J. DeWitt and P. Hawthorn "A Performance Evaluation of Database Machine Architectures" Proceedings 7-th VLDB Conference Cannes September 1981.

DEWI82 D.J. DeWitt and D. Friedland "Exploiting Parallelism for the Performance Enhancement of Non-numeric Applications" National Computer Conference 1982.

EPST80 R. Epstein and P. Hawthorn "Design Decisions for the Intelligent Database Machine" National Computer Conference 1980.

KNUT73 D.E. Knuth:"The Art of Computer Programming " Vol 3, Sorting and Searching, Addison-Wesley 1973.

KRUZ83 I. Kruzela "An Associative Array Processor Supporting a Relational Algebra" Doctoral Thesis, Department of Computer Engineering, University of Lund, Sweden, April 1983.

LANG78 G.G. Langdon "A Note on Associative Processors for Data Management" ACM Transactions on Database Systems Vol.3, No.2 June 1978.

LEIL78 H.O. Leilich, G. Stiege, and H.C. Zeidler "A Search Processor for Data Base Management Systems", Proc. 4th Conference on Very Large Databases, 1978.

LIPO78 G.J. Lipovski and S.Y.V. Su "Architectural Features of CASSM: A Context Addressed Segment Sequential Memory" Proc. 5th Annual Symposium on Computer Architecture, Palo Alto 1978.

MALL79 V.A.J. Maller "The Content Addressable File Store - CAFS" ICL Technical Journal Vol. 1 No. 3, November 1979.

MERR84 T.H. Merrett "Relational Information Systems" Reston Publishing Co. 1984.

OZKA75 E.A Ozkarahan, S.A. Schuster and K.C. Smith "RAP - An Associative Processor for Data Base Management" NCC 1975.

OLIV79 E.J. Oliver "RELACS An Associative Computer Architecture to Support a Relational Data Model" Ph. D. Dissertation, Syracuse University, 1979.

SU75 S.Y.V. Su and G.J. Lipovski "CASSM - A Cellular System for Very Large Databases" Proc. International Conference on Very Large Databases, Sept. 1975.

VALD84 P. Valduriez and G. Gardarin "Join and Semi-Join Algorithms for a Multiprocessor Database Machine" Report no 5, Institut de programmation, Universite P. et M. Curie, France, Jan 1983 and ACM Transactions on Database Systems Vol.9, No. 1, March 1984.

TECH84 "TECHRA Database Management System" Application Programmer's Manual, Kongsberg Våpenfabrikk, Norway, February 1984.