

COMPUTING JOINS OF RELATIONS

Leo R. Gotlieb

Department of Computer Science
University of Toronto
Toronto, Canada M5S 1A7

Little has appeared on the problems of efficiently storing, accessing, and manipulating relations. These are of considerable interest, both to implementors and those seeking comparison with other database models. In this paper we examine how one relational operation, the join, can be efficiently computed. Algorithms are presented, and evaluated with respect to storage use and computing time.

Keywords: joins, relational systems, data bases

CR Categories: 4.33 (Data Base), 4.9

0. INTRODUCTION

In [1] Codd proposed a database model in which data and operations on them were viewed in terms of mathematical relations. Although the model has been developed to the extent that relational systems now exist, little has appeared on the problems involved in efficiently storing, accessing, and manipulating relations. These are of considerable interest, both to implementors, and those seeking comparison with other database models.

One important relational operation is the join, basically a way of combining two binary relations to form a ternary one so that the information in the originals is preserved. Codd originally defined joins in very general terms [1], but so far the natural join of two relations on a single domain has seemed powerful enough that other kinds are not needed. In this paper we examine how this join can be efficiently computed.

The extent to which a relational operation can be made efficient depends on the level at which it is to be implemented; this in turn depends on the design of the whole system. If it is being built from scratch, then there is a good deal of freedom in optimizing the performance of operations considered important. If it is to be constructed on top of another system (e.g., a hierarchical one [4]), then relational operations will be implemented by primitives which may not quite "fit" relational requirements. Thus there will be some sacrifice of possible optimization in favour of other advantages, which may include elegance, convenience, reliability, etc.

Here, the concern is with low-level implementation--given files representing relations, what is the best way of computing the join? We shall try to optimize for a particular storage structure; this may suggest whether matters can be improved by modifying the representation.

Joining two relations is similar to intersecting two sets: in both operations a relationship between each element in the first set and each one in the second must be established. For N -element sets, this potentially requires $O(N^2)$ comparisons, where $O(N)$ here means "directly proportional to N ". The process can be speeded up to $O(N \log N)$ by ordering one or both sets and taking advantage of the order to do fast searching; an $O(N)$ dependency is achievable through hashing. (All log's here are to base 2.) One of the purposes of this paper is to determine the extent to which the theoretical efficiency is obtainable, since in practice, space and time costs must be considered in addition to internal operations.

The analysis is theoretical, in that no experiments have been carried out. Section 1 is a basic survey of the problem; section 2 goes into more detail. Algorithms are presented and evaluated with respect to storage use and computing time. It is assumed that the reader is familiar with the basic relational concepts and terminology. [1,2,3]

1. BASIC CONSIDERATIONS

1.1 Notation

The notation follows that of Codd [3]. A relation is essentially a table whose rows represent entities, and whose columns represent attributes. In strict relational terminology, attributes take on values from an attribute space, or domain; thus a relation is a subset of tuples from the Cartesian product $D_1 * D_2 * \dots * D_n$, where $*$ indicates cross product, and some of the domains D_i may be the same. Here we shall identify domain with attribute, and use domain (attribute) names to identify columns of the relation. A relation composed of n -tuples has degree n ; its size will be defined to be the number of tuples, or rows, denoted $\#R$.

To clarify the above, consider relation R with domains A and B, written $R(A, B)$. A and B name columns of R. The attribute or domain space of A is the set of values which entries of A may assume. #A denotes the size of this set. $R[A]$ refers to the set of values from the domain space actually present in the A-column. This is also called the projection of R on A. # $R[A]$ denotes the set size.

Small letters indicate rows, or tuples of a relation; r is a row of R. If A is a domain of R, $r[A]$ is the A-value of row r. (r, s) denotes the tuple formed by concatenating r from R and s from S.

Let ϕ be a mathematical relation (normally one of $=, \neq, <, \leq, >, \geq$, and unless otherwise stated, $=$). The ϕ -join of relations R and S on their respective domains A and B is written $R[A \phi B]S$, and defined to be $\{(r, s): r \text{ is in } R, s \text{ is in } S, \text{ and } r[A] \phi s[B]\}$. R and S are in First Normal Form (none of their domains is itself a relation) and may be of any degree. The only requirement for $R[A \phi B]S$ to exist is that A and B be ϕ -comparable, i.e., for all x in A and y in B, $x \phi y$ is either true or false.

1.2 Choosing the Algorithm

The appeal of the relational approach lies in its logical view of data; the concept of a database as a collection of tables is elementary, but it enables the casual user to extract detailed information in a natural way. This picture is less helpful to the implementor, for the definition of a relation says little about how it might be efficiently represented and manipulated. In particular consider computing the join of two relations.

The "obvious" way, which stems right from the definition, is the following:

```

for each r in R
do
  for each s in S
  do
    if  $r[A] \phi s[B]$  then form  $(r, s)$ .

```

Whether or not this is suitable for an actual implementation depends on several factors.

Relation Size

Consider joining two size N relations by the above method. If they are stored externally but can fit into core, the cost is roughly $2 * (\text{cost of reading in a size N relation}) + N^2 * (\text{cost of a logical comparison})$. (The choice of logical compare as a measure of compute time is discussed below.) Reads are much more expensive than internal operations, but at some value of N, the N^2 term will start to dominate the total. Where relations are smaller than this critical size, the N^2 contribution may be acceptable because the read cost is critical; once they get larger, it will pay to find an algorithm which does less internal computing.

Storage Representation

Storage representation influences the choice of algorithm in two ways. First, a particular method may simply not work with a given storage organization--for example a binary search cannot be used on a (sorted) linked list. Second, data structure as well as relation size can affect the efficiency threshold at which one algorithm becomes preferable to another. For example, call relations stored as sets of tuples containing actual data values primary. Using these, we can define a class of indirect or derived relations, whose domains consist of pointers, and whose rows specify concatenations of tuples from various primary relations [4]. (Note that such relations are not in First Normal Form, as their domains are really relations themselves.) It can be expected that join algorithms for primary relations will have to be modified to work for derived ones, and that this will result in some loss of efficiency, because of the level of indirectness introduced.

Nature of the Join Domains

The characteristics of the domains being joined on will affect the selection and performance of an algorithm. For example, when a domain is a key, equality searches on it can be stopped when the required value is found; this reduces the running time of the above method by about half. If the domain is ordered and amenable to a binary search, even better improvement is possible.

It is worth knowing something about the attribute space of a domain. If it is small compared to relation size, then tuples of the same value can be efficiently collected for

join purposes. Knowing how domain values are distributed is also useful--if the spread is fairly uniform, a fast sorting method such as address calculation can be used to speed up the join.

Workspace Available

Workspace can be used to speed internal processing and keep the number of I/O operations down. As an example, suppose two size N relations are to be joined on a domain which can only assume d possible values, $d \ll N$. The relations may not fit into core, but there could be room for $2*d$ buckets, which hold a total of $2*N$ row pointers. The first relation is read in and partitioned, i.e., i is placed in bucket k if the join domain of the i -th tuple assumes the k -th value in the attribute space. This is repeated for the second relation. The derived form of the join can be constructed directly from the two bucket sets. Each relation is scanned exactly once, and most of the internal processing involves pairing tuples from corresponding buckets. This degree of efficiency is hard to achieve unless a reasonable amount of internal workspace is available. (For very large relations we will have to keep workspace on secondary storage as well.)

1.3 Evaluation Factors

For evaluation we adopt a two-level storage model. Core, or main memory, is fast but expensive; secondary storage is cheap but access is slow. The performance of an algorithm will be evaluated according to the amount of core used (in characters, or bytes), computing time, and I/O time. The first criterion needs no explanation.

Computing time will be measured by the number of logical comparisons performed. This estimate may appear unrealistic, as it ignores the cost of data movement--in particular that involved in assembling the result. On the other hand, where the join is concerned, it does focus on that part of the computation which determines which tuples to pair up. Moreover, as is true with sorting, most of the (internal) data movement involved in computing the join occurs in direct proportion to the number of compares executed.

In any large system, relations will be stored externally, so read time is a factor in algorithm performance. To measure it we count the number of characters read. Once located, a character can be transferred from secondary to main store in time t_r . If external memory consists of disks, each read incurs an average seek time (t_s) and average latency time (t_l) cost. The cost of reading in a tuple of a relation w characters wide is therefore $t_s + t_l + w*t_r$. Files are organized so that when K tuples are read, only one unit of t_s and one of t_l are charged. The latency time can be eliminated if K is a multiple of track size. Similar considerations apply to disk writes.

In order to compare internal operations against those for input/output, performance expressions will be evaluated using times for a System 370/165 with 3330 disk drive. For this system, $t_s = 30$ msec., $t_l = 8.4$ msec., and $t_r = .0012$ msec.¹ (approximately). The time for a logical comparison will be taken as that required (on the average) to compare two 8-character fields--about .0012 msec.² This is slower than comparison for word size integers, but domains will frequently be wider than 8 bytes, so it represents a compromise value. These figures are intended for comparison purposes, and are only approximately indicative of actual running time.

2. ALGORITHMS FOR JOINS

We begin by assuming that relations are implemented in a "natural" way. A relation is stored on disk as a file of records. Each record represents a row, or tuple, and its fields are the domain (attribute) values. The i -th tuple of a relation is defined to be the i -th record of the corresponding file. It can be read in using a direct access method. When a relation (or part of one) is brought into core, it is accessible as a one-level structure array (cf. PL/I). For example, if the i -th tuple of $R(A,B)$ has been read in, it is referred to by $R(i)$; its A -domain value would be $R.A(i)$.

Relations are not assumed to be ordered, but there may be key lists (that is, directories) on various domains. These are lists of ordered pairs of the form (domain value, i), where "domain value" serves as the key, and i is an index to a row of a relation. The

¹ Source: IBM System/370 Systems Summary, Form GA22-7001-3 (4-th edition) p. 7-12.

² Source: IBM System/370 Model 165 Functional Characteristics Form GA22-6935-0 (p. 34). This figure is derived using a store to store logical compare (CLC) and assuming that the not-equal condition is raised, on the average, at the 4-th byte.

list is usually sorted on the key to permit fast searching, which could alternately be accomplished by hashing. In most cases, relations will be so large that key lists on them will not fit into core. They are then stored as files (of two-field records), accessible through the access method.

Sometimes it is useful to implement key lists with pointers. When the list is being constructed, some type of search-tree organization is particularly attractive as it allows fast insertion and eliminates the need for data movement [6]. If domain values are repeated, the (domain value, index) pairing described above is modified so that the key points to a list of indices. In this case, the key list is just the collection of inverted lists on all the domain values present.

A binary relation of tuple pointers provides a natural way of representing the derived form of the join. $R[A \bowtie B]S$ is represented by $RSAB(R_i, S_i)$, where R_i and S_i are domains of integers. $RSAB.R_i(h) = j$ and $RSAB.S_i(h) = k$ means that the j -th row of R and the k -th row of S concatenate to form a row of the join. Conversely, when the join is being computed and (r, s) is found to be in it, the row "(index of r in R , index of s in S)" is added to $RSAB$. $RSAB$ is initially empty, and tuples are placed in it using the primitive ADD . The cost of actually assembling this relation is not considered, as it represents an add-on term which does not depend on the choice of algorithm. We can imagine that $RSAB$ is built up in a buffer, which is written out on disk when full.

R and S are relations of size N and width w . K is a block size representing the portion of a relation read into core; for convenience it is assumed to evenly divide relation ($list$) size.

The "obvious" method of computing the join is presented first, and then improved on.

2.1 Algorithm 1 ("Obvious" Method) $RSAB(R, S)$, the derived form of $R[A \bowtie B]S$, is to be computed.

1. (Loop on h) Perform steps 2-3 for $h = 0, K, 2K, \dots, (N/K-1)K$; then terminate. (If $K = N$, both relations fit into core.)
2. (Read a block of R) Read $R(h+1)$ through $R(h+K)$.
3. (Loop on i) Perform steps 4-5 for $i = 0, K, 2K, \dots, (N/K-1)K$.
4. (Read a block of S) Read $S(i+1)$ through $S(i+K)$.
5. (Loop on p) Perform step 6 for $p = 1, 2, \dots, K$.
6. (Connect tuples and put them in the join) Do for $q = 1, 2, \dots, K$: If $R.A(h+p) \neq S.B(i+q)$ then $ADD(RSAB, (h+p, i+q))$.

Cost

Core used: $2Kw$ bytes.

Comparisons: $N^2(2 + 1/K + 1/K^2 + 1/NK)$. Indexing accounts for N/K compares in step 1, N^2/K^2 in 3, N^2/K in 5, and N^2 in 6. In addition there is one $\neq$ compare for each of the N^2 executions of step 6.

I/O: Step 2 accounts for N/K reads; step 4 accounts for N^2/K^2 of them. At each read, K tuples are brought in, for a total of $(N/K + N^2/K^2) * (t_s + t_l + K^*w*t_r)$.

S/370 Time: To get an idea of which term is most important, note that $2N^2$ is the main term in the number of compares, and that the dominating term in the left factor of the I/O time expression is $(N/K)^2$. Substituting for t_l , t_s and t_r gives $(N/K)^2 * (38.4 + .0012*w*K + .0012*2*K^2)$ msec.³

Remarks

The comparisons start to dominate the 370 time expression when K gets above 100 (assuming relation width is $O(100)$). In other words, expensive as reads are, it pays to reduce the number of compares, even for very small relations. If one of A or B is a key, and if the join criterion is equality, then an exit can be made from the loop in step 6 on a tuple match and a return made to step 3. This saves about half the total number of operations (and so would be worth doing) but does not change the above conclusion. To significantly reduce the number of comparisons, searches on one of the relations must be speeded up.

³ Sometimes it is possible to overlap compute and I/O operations. This depends on the system and the algorithm. In such cases, that total time is closer to the maximum of compute time and I/O time than it is to the sum of the two.

In steps 5 and 6 above, a tuple in R is compared to every one in S. This can be avoided if there is a key list on B organized to permit fast searching. Suppose that INV_S is such a list. A and B may be key domains; if they have repeated values, let the size of the attribute space be $M < N$.⁴ M is assumed small enough that INV_S , the header of the inverted lists, can fit into core. We are concerned with two cases in particular--when A and B are both key domains, and when neither of them are.

2.2 A, B Not Keys

In this case each tuple in R must be paired with those in one of the M inverted lists on B. Since it is expensive to read in a list for each r of R, the tuples are collected according to A-value and then paired (effectively constructing M inverted lists on A).

Let 1 $INV_S(1:M)$, 2 Bval, 2 Blist be the header list. For $1 \leq i \leq M$, $INV_S.Blist(i)$ is the address of the externally stored list of indices of tuples s in S such that $s[B] = INV_S.Bval(i)$. FIND is a routine for searching INV_S ; $FIND(INV_S, b) = i$ if b is in $S[B]$ and $INV_S.Bval(i) = b$, 0 otherwise. The organization of INV_S for efficient searching is discussed under "Cost".

Algorithm 2 (Compute $R[A=B]S$, the equi-join, using a collection of inverted lists on B; neither A nor B is a key domain).

* Suppose there is space for K rows of R, $M \cdot J$ pointers, and two header lists, $INV_S(1:M)$, and $MATCH(1:M)$. $MATCH(i)$ is the file address of an initially empty list of row indices (referring to R). It is assembled in the process of the algorithm by placing indices in one of M sequentially allocated lists, each holding J indices. The lists are paired so that they "grow" towards each other as they are filled; when a list pair is full, the whole area is written onto disk, the i-th list being added to that at $MATCH(i)$. $MATCH$ and INV_S are initially in core. The algorithm has two parts, and it is assumed that the workspace from the first part can hold, for any i, the $MATCH(i)$ and $INV_S.Blist(i)$ lists during the second part.

1. (Loop on h) Perform steps 2-3 for $h = 0, K, \dots, (N/K-1) \cdot K$; then goto step 4.
2. (Read a block of R) Read $R(h+1)$ through $R(h+K)$.
3. (Partition R) Do for $i = 1, 2, \dots, K$: Set $L = FIND(INV_S, R.A(h+i))$, and add $h+i$ to the list for $MATCH(L)$.
4. (Empty in-core lists; then loop on j) Write out the remaining list contents; then perform steps 5-6 for $j = 1, 2, \dots, M$; then terminate.
5. Read in the lists pointed to by $INV_S.Blist(j)$ and $MATCH(j)$.
6. (Connect tuples and place them in RSAB) For each index p in $INV_S.Blist(j)$ do: for each index q in $MATCH(j)$ do: $ADD(RSAB, (p, q))$.

Cost

Assume that value occurrences in A and B are uniformly distributed over the common attribute space, and that their physical distribution over R is fairly random as well. The lists $INV_S.Blist(i)$ and $MATCH(i)$ thus contain about N/M indices for each i.

Core used: Suppose pointers (and indices) take 4 bytes, and let v be the width of A and B (w is the width of R). $M \cdot (v+4) + 4 \cdot M$ characters are needed to hold INV_S and $MATCH$. Steps 2-3 require $K \cdot w$ bytes for blocks of R + $4 \cdot M \cdot J$ bytes for the M lists; by assumption this total is at least equal to the $8 \cdot N/M$ bytes needed for the two lists in steps 5-6. The core requirement is therefore $K \cdot w + M \cdot (v + 8 + 4 \cdot J)$ bytes.

Comparisons: Let $F(n)$ be the average number of compares taken by FIND on an n-element table. Step 3 accounts for $N \cdot F(N)$ comparisons. If INV_S is ordered on Bval, FIND can be a binary search, making $F(n)$ $O(\log n)$. Another possibility is to organize INV_S as a chained scatter table (perhaps using some extra space to reduce the chance of collisions). Since searches are always successful here, $F(n) \leq 2$ ([5], pp. 513-517). Suppose hashing is used; then step 3 takes about $2 \cdot N$ compares. Looping accounts for N/K in step 1 and with the uniformity assumption, about N^2/M^2 .

⁴ This is a simplification. There could be repeated values in a domain even when the space is much larger than relation size, but this makes timing difficult to estimate--we then have to guess how many values in $R[A]$ coincide with ones in $S[B]$, what the multiplicity is, etc.

in step 6. List bookkeeping in step 3 accounts for N comparisons. The total is therefore about $N^2/M^2 + N*(3 + 1/K)$. This is surprising--with the fast searching one would expect an $O(N)$ dependency, but when both domains have repeated values, forming the cross product of sets of tuples which agree on the join domain dominates the search comparisons. Note however, that comparisons involved in looping are generally less expensive than those required for matching tuples. First, they are usually done in registers--on the 370/165 this means a speed increase by a factor of more than 10 over the 1.2 micro-sec. store-to-store compare we have been using as a standard. Second, looping compares are associated with counting, and many computers have an instruction to perform both these functions in one operation.

I/O: Step 2 accounts for $N/K*(t_s + t_1 + K*w*t_r)$ sec. In steps 3-4, a total of N 4-byte pointers are written out. Given the uniformity assumptions, the M lists should fill at a fairly equal rate, with variations being handled by the pairing scheme mentioned above. The in-core list area (capacity $M*J$) will therefore be written out about $N/(M*J)$ times. If the M list areas on disk are suitably organized on a cylinder, each write costs $t_s + t_1 + M*J*4*t_r$ sec. Step 5 is executed M times, and on each occasion two lists of about N/M pointers are brought in, taking $2*M*(t_s + t_1 + 8*N/M*t_r)$ sec. in all. Total I/O time is therefore $N*((1/K + 1/MJ + 2*M/N)*(t_s + t_1) + (w+20)*t_r)$ sec.

S/370 Time: About $N * (38.4*(1/K + 1/MJ + 2*M/N) + .0012*(w+20) + .0012*N/M^2)$ msec., if the number of comparisons is taken as N^2/M^2 .

Remarks

Algorithm 2 is a definite improvement over Algorithm 1, where I/O time is $O(N^2)$. Comparisons have been reduced by a factor of M^2 , and being associated with looping, they are "cheaper" than those of Alg. 1. The uniformity assumption makes timing easy to estimate, but it is not strictly necessary. With linked allocation, the MATCH lists can be assembled without regard to the rate at which they fill. Step 5 represents a simplification in that the two lists are assumed to fit into core. If they do not, step 6 must be expanded to take the cross product in blocks, similar to the way whole relations were processed in Algorithm 1.

In the process of Alg. 2, inverted lists on A are constructed. This can be avoided by reading in $INV_S.Blist(i)$ and doing the pairing of step 6 whenever the i -th in-core list associated with $MATCH(i)$ gets full. I/O time is increased, however; an N/M element list is read in each time a J -element list fills. Since there are N elements in all, lists will fill about N/J times, for a read time of $N/J*(t_s + t_1 + N/M*4*t_r)$ (replacing the $N*((1/MJ + 2*M/N)*(t_s + t_1) + 20*t_r)$ in the above I/O expression). The data transfer time is now $O(N^2)$, and the seek and latency time will be considerably greater, unless $J \geq N/2M$ --unlikely since $J*M$ is a term in the core requirement expression. Therefore, unless there is a severe shortage of disk space, it saves, not costs, to construct the inverted lists on A.

2.3 A,B Keys

Suppose A and B have no repeated values. INV_S now has N elements, which it is assumed, makes it too large for core; it takes the form 1 $INV_S(1:N)$, 2 Bval, 2 row, where "row" is an integer row index, rather than a header list as before. As noted in the last paragraph of 2.2, it is very expensive to repeatedly read in a list, or portion thereof, during processing; some arrangement of tuples of R which minimizes scans of INV_S is called for. Given that INV_S is ordered on Bval, Algorithm 3 does this by partitioning R into A-value ranges determined by the end-values of the blocks of INV_S that can fit into core, and then doing in-core joins on pairs of blocks.

Algorithm 3 (Compute $R[A=B]S$ using an ordered key list on B; A and B are key domains.)

Suppose INV_S can be read in blocks of N/M elements, where M here is a parameter which reflects available workspace. There are M ranges, each determined by the maximum and minimum values per block; these are represented in an in-core array $RANGE(0:M)$. $RANGE(0) = INV_S.Bval(1)$, and $RANGE(i) = INV_S.Bval(i*N/M)$ for $1 \leq i \leq M$. FIND functions as in Alg. 2, but works on portions of $INV_S.FINDR$, similar to FIND, searches $RANGE$: $FINDR(RANGE,a) = i$ if $RANGE(i-1) < a \leq RANGE(i)$, 1 if $a = RANGE(0)$, and -1 otherwise. In Alg. 2 it was enough to place a tuple index in one of the M lists according to the A-value of that tuple; here the value must be stored as well. The function of MATCH is assumed by $PART(1:M)$. $PART(i)$ points to a list of ordered pairs (a,p) , where $R.A(p) = a$ and $FIND(RANGE,a) = i$. $PART$ is stored and assembled the same way MATCH was. There is room for $M*J$ pairs in the list area before it must be written out.

1. (Initialize RANGE) Initialize RANGE as described above, reading in the appropriate elements of INV_S.
2. (Loop on h) Perform steps 3-4 for $h = 0, K, \dots, (N/K-1)*K$; then goto step 5.
3. (Read a block of R) Read $R(h+1)$ through $R(h+K)$.
4. (Partition R according to RANGE) Do for $i = 1, 2, \dots, K$: Set $L = \text{FINDR}(\text{RANGE}, R.A(h+i))$. If $L > -1$ add $(R.A(h+i), h+i)$ to the PART(L) list.
5. (Empty in-core lists; then loop on j) Write out the remaining list contents; then perform steps 6-7 for $j = 1, 2, \dots, M$; then terminate.
6. (Process the lists) Read in the PART(j) list; let $i = (j-1)*N/M$; Read in $\text{INV_S}(i+1)$ through $\text{INV_S}(i+N/M)$.
7. (Connect tuples and place them in RSAB) For each pair (a,p) in the PART(j) list do: Set $L = \text{FIND}(\text{INV_S}(i+1:i+N/M), a)$; If $L > 0$ ADD(RSAB, (p, $\text{INV_S}.\text{row}(L)$)).

Cost

Assume that the A-values which fall between $\text{RANGE}(0)$ and $\text{RANGE}(M)$ are uniformly distributed and that their physical distribution over R is fairly random. Each PART(j) list will then have at most N/M pairs. Again, this is done mainly to estimate timing.

Core used: (v is the width of A,B.) $M*(v+4)$ to hold RANGE and PART + $K*w$ for blocks of R + $M*J*(v+4)$ for the M in-core lists. All but PART can be relinquished after step 5 to hold the two lists in steps 6-7 ($2*N/M * (v+4)$ bytes). The core required is therefore $K*w + M*(J*(v+4) + v + 8)$ bytes.

Comparisons: RANGE and INV_S are both ordered, so let FIND and FINDR be binary search routines. Both are executed about N times, on, respectively, M and N/M -element arrays. Explicit looping accounts for $2*N + N/K + M$ compares, and implicit looping due to in-core list management takes N more. The total is therefore $N*(\log N + 3 + 1/K) + M$.

I/O: The analysis is very similar to that for Alg. 2. We have (about) $M*(t_s + t_l + (v+4)*t_r)$ from step 1 + $N/K*(t_s + t_l + K*w*t_r)$ from step 3 + $N/MJ*(t_s + t_l + M*J*(v+4)*t_r)$ from steps 4-5 + $2*M*(t_s + t_l + N/M*(v+4)*t_r)$ from step 6. The total is $N*((1/K + 1/MJ + 2*M/N)*(t_s + t_l) + ((v+4)*(M/N + 3) + w)*t_r)$ sec.

S/370 Time: About $N*(38.4(1/K + 1/MJ + 3*M/N) + .0012*(w + 3*v + 12) + .0012*\log N)$ msec.

Remarks

M plays the same role as it does in Algorithm 2, but here it is a workspace parameter. For the same J, $J*v$ more bytes are needed to store A-domain values when partitioning R. This, and the necessity to initialize RANGE account for 3's slightly greater I/O time. Otherwise the time formulas are comparable, the difference being in the number of comparisons. If there is good reason to believe in the uniformity assumption, FIND and FINDR can be implemented as address calculation searches. This reduces the number of comparisons to $O(N)$. The same effect can be accomplished if INV_S is organized as a scatter storage table; Algorithm 3 must then be modified slightly.

2.4 An Example

To get a more concrete idea of how Algorithms 1, 2, and 3 compare consider two relations of size 10,000 and width 100 ($N = 10^4$, $w = 100$) being joined on an 8-byte domain ($v = 8$). In addition, $M = 100$ (multiplicity, in Alg. 2), $K = 100$, and $J = 10$. For these figures, the expressions for core requirements and 370/Time are evaluated.

Algorithm 1

Core: 20,000 bytes.

Time (msec.): $(10^4/100)^2 (38.4 + .0012(100)(100) + .0012(2)(100)^2) = 10^4 (38.4 + 36) = 744$ sec. (Clearly prohibitive)

Algorithm 2

Core: $10,000 + 100(8+8+4*10) = 15,000$ bytes (approx.).

Time (msec.): $10^4(38.4(1/100 + 1/1000 + 2/100) + 12/10^4(120) + 12/10^4) = 38.4(300) + 1440 + 12 = 13$ sec. (approx.).

Algorithm 3

Core: $10,000 + 100(12 + 10(12)) = 22,000$ bytes (approx.).

Time (msec.): $10^4(38.4(1/100 + 1/1000 + 3/100) + 12/10^4(136) + 12/10^4(2 \cdot \log 100))$
 $= 38.4(400) + 12(150+14) = 17$ sec. (approx.).

Note how the necessity of initializing RANGE makes Alg. 3 25% slower than Alg. 2. Also, N/M is small enough that Algorithm 3 does more compares as well.

In all cases, work space is less than 2% of relation size. In Algorithms 2 and 3, seek and latency time dominate the cost. They can be reduced by increasing K, the space available for scanning the relation, and by reading back in at once, more than one inverted list or range. In Algorithm 1, adding workspace helps somewhat, but compares still account for 240 seconds.

2.5 Other Cases

Suppose there are no key lists on domains A or B. If neither are keys, Algorithm 2 can be used with only slight modification--both R and S are partitioned according to join-domain values instead of just R. If the domains are keys, there is a question of whether one or both should be ordered. One ordered domain can be used to partially sort the other as is done in Algorithm 3; in fact totally sorting it would add to the running time by increasing the number of comparisons and the amount of data moved between core and disk in the first part, while saving only slightly in the last step. If both domains are unordered, the principle behind Alg. 3--Radix Sorting--can be employed to compute the join almost efficiently as is done there.

The idea is to do a radix sort on key lists of the two join domains, starting at the high-order byte.⁵ Working this way, the number of buckets increases by a factor of b, the base, at each pass. The key lists can be constructed on the first pass over the relations; then they can be manipulated. The sort stops when the number of keys in each bucket is such that buckets can be read into core. After the sort, corresponding buckets from the two domains are read in and joined. Uniformity is usually assumed, but buckets can be coalesced to offset skew distributions of values.

As an example, consider size N relations being joined on an alphabetic domain, and suppose there is room for K entries from each of two key lists. The number of passes required will be p, where $N/26^{**p} \geq K$. After this point, buckets can be merged to make full advantage of the core available. The join then becomes a problem of N/K in-core joins of size K blocks. This takes at most K*N comparisons (without sorting or hashing), and as seen in Sec. 2.4, they will probably not contribute significantly to total time.

3. CONCLUSIONS

Several ways of computing the join have been considered. A primary requirement for efficiency is that I/O activity be kept down; this is achieved by using as large an internal workspace as possible, and carefully organizing external files. The number of comparisons is not as important as might have been expected, but it can be prohibitive (Alg. 1). The best way to minimize compares and I/O is to adopt a "divide and conquer" strategy where possible, i.e., reduce the problem of joining large, externally stored relations to that of joining several smaller ones in core.

For more efficient joining, relations can be represented in "split" form, that is, with their domains stored separately. This eliminates reading unnecessary information on the initial scan, and enables more domain values to fit into available workspace for matching and partitioning. In the above algorithms, it would effectively increase the value of K, reducing I/O time. On the other hand, separate domain storage makes individual rows more costly to access.

Another possibility is to keep multilists, inverted lists, or sorted key lists on domains which are frequently joined on. The first two aids would not seem to save that much time--Alg. 2 takes only 25% longer if it is modified to construct two sets of inverted lists instead of one. On the other hand, when both domains are keys, ordered key lists are definitely worth having--they enable the join to be performed in a merge-type operation requiring only one pass for each list. However it may not be worth maintaining up-to-date key lists. A compromise is to keep one ordered join domain per pair of

⁵ This was suggested by M.D. MacLaren in [7] as a way of efficiently sorting large files.

frequently joined relations, but it should be remembered that, using the method suggested in Sec. 2.5, a join on two unordered keys domains can be done in time comparable to that taken by Algorithm 3.

The problem of joining derived relations (Sec. 1.2) has not been considered. These are too expensive to scan, as is done in the above algorithms, because the rows can point anywhere into the constituent primary relations. An observation which may prove useful is the following: Let $RSAB(R_i, S_i)$ be a derived relation representing $R[A=B]S$, and similarly for $XYEF(E_i, F_i)$. Suppose that relations $R[A=B]S$ and $X[E=F]Y$ are to be joined on domains C of S and G of X . Then the derived form of this join can be computed by taking the join $RSAB[S_i \phi E_i]XYEF$, where ϕ itself is the (two column) derived form of $S[C=G]X$, considered as a mathematical relation. The problem is then whether, for i and j , the truth of $i \phi j$ can be quickly determined. If so, the above algorithms can be applied, using ϕ as the match criterion. This approach may merit further investigation.

ACKNOWLEDGEMENT

I would like to thank Dennis Tsichritzis for suggesting the problem. Support from the National Research Council of Canada is gratefully acknowledged.

REFERENCES

- [1] Codd, E.F., "A Relational Model of Data for Large Shared Data Banks", CACM 13, 6 (June 1970), pp. 377-387.
- [2] Codd, E.F., "Further Normalization of the Data Base Relational Model", in Courant Computer Science Symposium 6, Data Base Systems, Randall Rustin, ed., Prentice Hall, New York, May 1971.
- [3] Codd, E.F., "Relational Completeness of Data Base Sub-languages", in Courant Symposium 6, cited in [2].
- [4] Tsichritzis, D., "On the Implementation of Relations", CSRG Technical Report 35, Computer Systems Research Group, University of Toronto, May 1974.
- [5] Knuth, D.E., "The Art of Computer Programming", Volume 3 (Sorting and Searching), Addison Wesley, 1973.
- [6] Nievergelt, J., "Binary Search Trees and File Organization", Computing Surveys 6, 3 (Sept. 1974), pp. 195-207.
- [7] MacLaren, M.D., "Internal Sorting by Radix Plus Sifting", JACM 13, 3 (July 1966), pp. 404-411.