PUC

Series: Monographs in Computer Science and Computer Applications

Nº 13/76

A LINEAR TIME SORT ALGORITHM APPLICABLE TO DATA BASE MANAGEMENT

by

G. C. Magalhães

and

A. L. Furtado

Departamento de Informática

Pontifícia Universidade Católica do Rio de Janeiro

Rua Marquês de São Vicente 225 — ZC 19

Rio de Janeiro — Brasil

Series: Monographs in Computer Science
and Computer Applications

Nº 13/76

A LINEAR TIME SORT ALGORITHM APPLICABLE TO DATA BASE MANAGEMENT*

by

G. C. Magalhães

and

A. L. Furtado

Series Editor: Roberto Lins de Carvalho October 1976

This research has been supported in part by FINEP under contract Nº CT/370.

SETOR DE DOCUMENTAÇÃO E INFORMAÇÃO
CÓDIGO / REGISTRO 2970
DATA 14 , 3 , 77
DEPT.º DE INFORMÁTICA

M 2422 ex. 2
DEPARTAMENTO DE INFORMÁTICA
SETOR DE DOCUMENTAÇÃO
E INFORMAÇÃO

Copies may be requested from:

Rosane Teles Lins Castilho
Head, Setor de Documentação e Informação
Depto. de Informática - PUC-RJ
R. Marques de São Vicente, 209 - GÁVEA
20.000 - RIO DE JANEIRO - BRASIL

ACKNOWLEDGEMENTS

The authors would like to thank M. Stonebraker and D. Tsichritzis for providing some useful comments.

ABSTRACT:

A sorting algorithm suitable for data base management applications, especially with relational data bases, is described. It is shown that it sorts a relation in time essentially linear in the number of n-tuples in the relation. The assumed data structures are described, as well as certain strategies for the convenient handling of secondary storage.

KEY WORDS:

Sorting, relational data bases, inverted files.

RESUMO:

Um algoritmo de classifcação adequado para aplicações de bancos de dados, especialmente bancos de dados relacionais, é descrito. Mostra-se que classifica uma relação em tempo essencialmente linear no número de n-tuplas da relação.
As estruturas de dados requeridas são descritas, assim como certas estratégias para a manipulação conveniente da memória secundária.

PALAVRAS CHAVE:

Classificação, bancos de dados relacionais, arquivos invertidos.

CONTENTS

1 - INTRODUCTION.. 1

2 - STATEMENT OF THE PROBLEM........................... 2

3 - THE ALGORITHM....................................... 5

4 - AN ITERATIVE VERSION................................ 6

5 - APPLICATION TO DATA BASES.......................... 12

6 - CONCLUSION... 15

REFERENCES... 16

1- INTRODUCTION

Sorting is a crucial feature in many data base systems. For relational data bases [1] it has been shown that, as far as the number of comparisons are concerned, the computational complexity of several relational algebra operations is dominated by the complexity of the sort [2,3,4].

The paper shows that, wherever a certain (fairly common) organization is adopted, a relation can be sorted in time essentially linear in the number of n-tuples in the relation. The techniques reported here are being investigated in the HYADES data base implementation project, conducted at PUC/RJ; in particular, their use for executing the relational algebra operations has been studied [5].

The sorting algorithm to be described can be related to radix sort [6,7,8]. It is not claimed that the algorithm is optimal; however it is suitable for data base management problems, performing very well within certain operational constraints.

2- STATEMENT OF THE PROBLEM

Let $\hat{T}$ be a table with m rows and n columns. Equivalently $\hat{T}$ can be viewed as a set of m n-tuples. No assumptions will be made about the type and range of the values of the table entries (components of the n-tuples).

The representation considered here for $\hat{T}$ consists of:

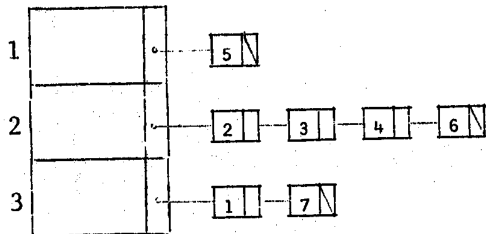
a. an m by n pointer array T; each entry of which points to the appropriate element in the arrays below;

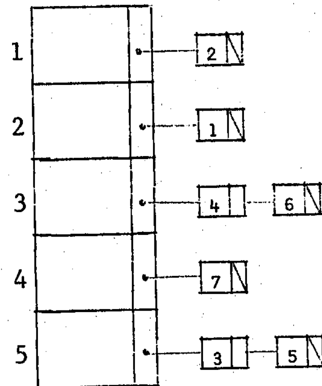
b. arrays $C_1, C_2, \ldots, C_m$ with two columns. In each array, the first column stores uniquely and in collating sequence the values in the corresponding columns of the given table $\hat{T}$; the second column provides headers to the lists below;

c. lists with two fields, to be called occurrence lists. The name field of an occurrence list-cell attached to an element of C contains a backpointer from C to a row of T; the other field points to the next list-cell.

An example is given in Fig.1. Values are not shown for the C's since they are not needed for this discussion; thus all the represented entries are pointers, noting that by pointers one does not mean necessarily physical addresses:

T			
1	3	2	3
2	2	1	1
3	2	5	4
4	2	3	3
5	1	5	2
6	2	3	4
7	3	4	3





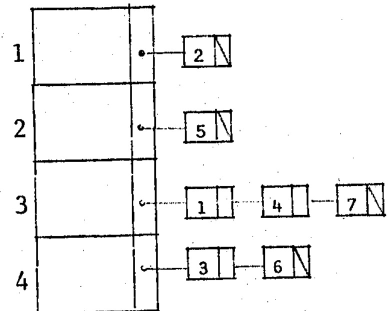


Fig.1. Representation of $\hat{T}$

From the fact that the C's are ordered and possess occurrence lists pointing to the rows of T it follows that this representation shows $\hat{T}$ sorted, separately, on each column. In the example, the sorted sequence of n-tuple identifiers (row numbers) are:

for C_1 - (5,2,3,4,6,1,7)
for C_2 - (2,1,4,6,7,3,5)
for C_3 - (2,5,1,4,7,3,6)

The problem is now to obtain $\hat{T}$ sorted on k columns, $k \leq n$, indicated by a sequence of column numbers $S = (s_1, s_2, \ldots, s_k)$ where $1 \leq s_j \leq n$.

3- THE ALGORITHM

The algorithm takes as input a list ℓ_0 of all the n-tuple identifiers. It then traverses sequentially C_{s_1}. Each attached occurrence list is considered; if it has only one tuple identifier in common with ℓ_0, this tuple identifier is placed in the output list; if there is more than one tuple identifier the algorithm calls itself recursively with the common tuple identifiers as input to try to resolve the tie using C_{s_2} ,and so forth.

A more detailed description is given in a notation close to "pidgin ALGOL" [6].

```
procedure SORT (ℓ,i):
    for j←1 step 1 until | C[S_i][j] | while |ℓ| ≠ 0 do
        begin
            t ← ℓ ∩ list(C[S_i][j]);
            ℓ ← ℓ-t;
            if |t| ≠ 0 then
                if i=k or |t| = 1 then
                    output ← t
                else SORT (t,i+1)
        end

    SORT (ℓ0,1);
```

It can be shown that, as it stands the algorithm has a quadratic worst case. However it can be improved, the main observation being that no use has been made inside the procedure of the pointer array T. In particular an iterative version, using T, will be described next.

4- AN ITERATIVE VERSION

The iterative version works in a minor - to - major manner. It traverses sequentially C_{s_k} and its occurrence lists, going through T, for re-ordering *in situ* the occurrence lists of $C_{s_{k-1}}$; after this is done the latter lists show $\hat{T}$ ordered with respect to (s_{k-1}, s_k); they are then used to re-arrange the lists of $C_{s_{k-2}}$, and so on, until the lists of C_{s_1} show $\hat{T}$ ordered with respect to the entire sequence S.

The occurrence lists can be regarded as queues, and an auxiliary pointer array P (with $\max_j (|C_{s_j}|)$ elements) is used to indicate the next list-cell to be updated in each queue; the preceding cells at a given stage, have already been updated (they already "belong" to the queue, whilst the others have still to be over-written).

A more detailed description of the algorithm follows. At each step of the main loop the lists of some C_a are used to re-order the lists of C_b. The current-cell-to-update pointer array P is initialized with a copy of the second column of C_b (the * instead of a row-subscript indicates that an entire column is taken). From list-cell q of C_a one goes to a tuple r of T and from it to a list-cell of C_b via P_v; the contents of the latter list-cell are changed to contain the tuple identifier r, and P_v in turn is advanced to point to the next list-cell.

```
begin
  for i ← k step - 1 until 2 do
    begin
      a ← S[i];
      b ← S[i-1];
      P ← C_b[*,2];
      for u←1 step 1 until |C_a| do
        for each cell q of C_a[u] do
          begin
            r ← name[q];
            v ← T[r,b];
            name[P[v]] ← r;
            P[v] ← next[P[v]]
          end
    end
  end
```

Fig.2. displays, for one step of the algorithm, the situation just before the change in the occurrence list of $C_h[v]$ is performed.

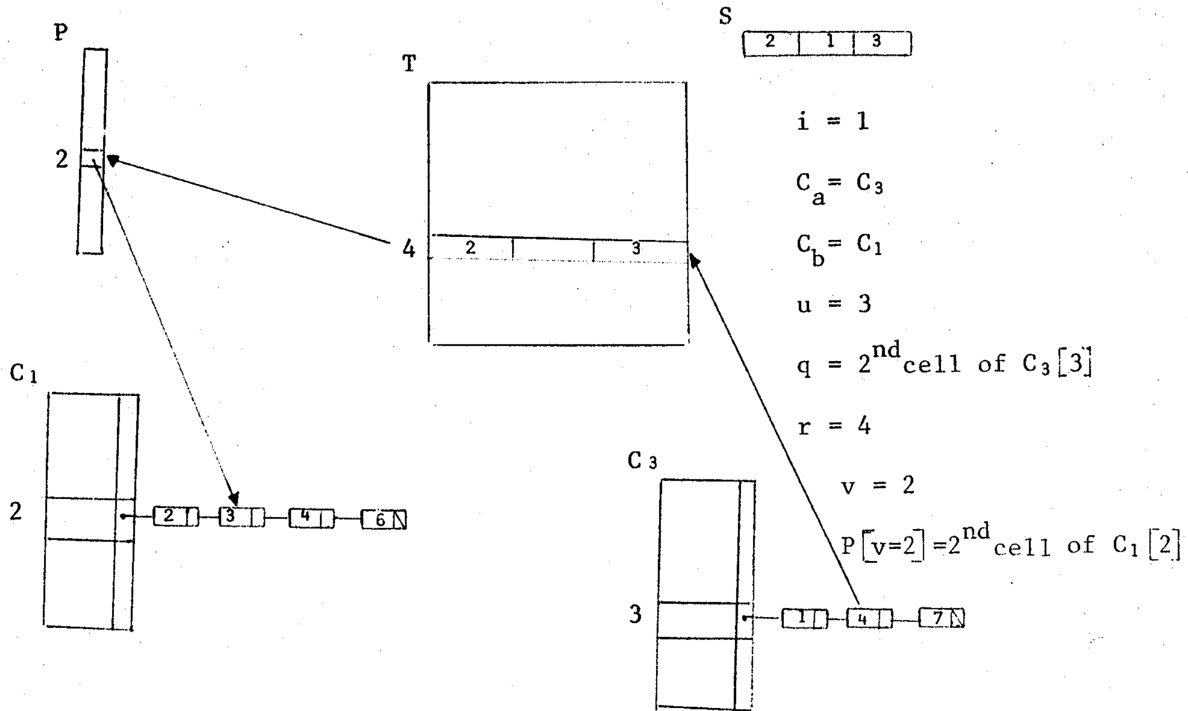


Fig.2. Step of the algorithm - before update

The change involves the name field of a cell in the occurrence list, and the queue pointer in P which is to be advanced (Fig.3).

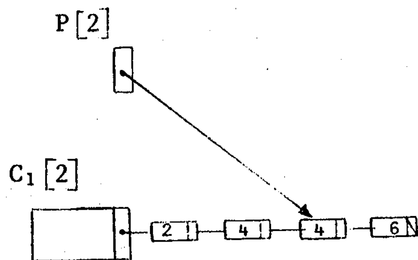


Fig.3. Step of the algorithm - after update

In the sequel it is shown that the algorithm is correct and that it works in time $O((k-1).(\max_j(|C_{s_j}|)+m))$. The argument is similar to the proof for the radix sort algorithm.

It is done by induction on the number of executions of the outer loop; an informal outline follows:

- Basis - before the computation starts, $\hat{T}$ is already sorted with respect to (s_k), via the occurrence lists of C_{s_k}.

- Hypothesis - assume that when the loop i=w starts $\hat{T}$ is sorted with respect to $(s_w, s_{w+1}, \ldots, s_k)$, via the occurrence lists of C_{s_w}.

- Induction step - when re-constructing the occurrence lists of each element of $C_{s_{w-1}}$, the algorithm

  . places a tuple identifier r before some other r' iff r is not greater than r' with respect to $(s_w, s_{w+1}, \ldots, s_k)$; note that C_{s_w} and its (re-arranged) lists are being traversed sequentially;

  . re-installs a tuple identifier always inside the same occurrence list (of the same $C_{s_{w-1}}$ element); hence the pre-existing ordering of $\hat{T}$ with respect to $C_{s_{w-1}}$ is not disturbed;

  therefore, at the end of the loop, $\hat{T}$ is sorted with respect to $(s_{w-1}, s_w, \ldots, s_k)$.

The outer loop is executed k-1 times, and at each time all cells of the occurrence lists of C_{s_w} are visited - the number of these cells being obviously m, the number of n-tuples. The factor $\max_j(|C_{s_j}|)$ refers to the initialization of P; in reality this factor is not dominant because $m \geq \max_j(|C_{s_j}|)$.

Clearly what is being counted is the number of executions of the internal loop, which involves essentially the traversal of a path of length three.

The following remarks can be made about the algorithm:

a. If descending rather than ascending order is desired for some or all columns, it suffices to traverse the respective C's from bottom to top; the occurrence lists would still be traversed in the "natural" direction (following the next pointers).

b. While the occurrence lists are being over-written by the algorithm some tuple identifiers are temporarily unavailable in an occurrence list (cf.the absence of tuple identifier 3 in Fig.3). One of the several ways to avoid this (in case one wishes to maintain such information at all times for concurrent operation) is to interchange the name fields instead of overwriting; for doing this without a search, one can convert the entries of T into two-field entries, the second entry being a pointer to the appropriate cell in the occurrence list as shown in Fig. 4.

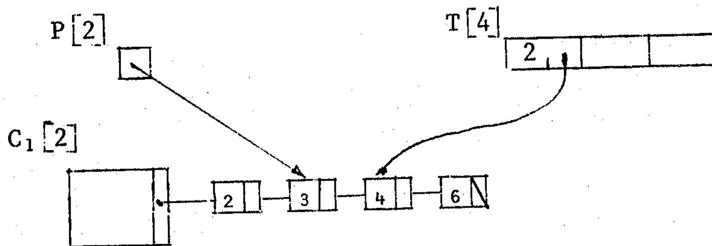


Fig.4. Pointer to occurrence added to T

The extra pointer is also useful in the situation (common in data base management) where, upon the deletion of an n-tuple, one must delete all references to it.

c. If one or more of the C's have one-cell lists for all elements (the corresponding columns being "key domains" in the data base sense), this knowledge can be used to speed-up the algorithm: it can start from the leftmost such C in S. This can be generalized to collections of C's which taken together have this property (composite keys). Recall also that in the original recursive version the algorithm will continue towards minor domains, for each subset of tuples, only while the subset has more than one tuple; consequently it takes advantage automatically of the existence of simple or composite keys.

5- APPLICATION TO DATA BASES

In data base terminology $\hat{T}$ is the extension (i.e. the set of tuples at a given instant) of an n-ary relation.

The representation used here is the data pool/inverted files organization. Keeping the C's in order can be done, for example, by using order preserving storage mapping functions [9] or index sequential schemes. Either of these storage mechanisms also speeds single pivot joins [2]; moreover the work required to create and maintain the C's is paid only once regardless of the number of times the sort algorithm is invoked.

With this representation, access paths are available to the algorithm in such a way that no searches or even comparisons are performed. This in turn is what makes the algorithm work in linear time.

One practical problem with data bases is the size of the file as compared to main store. Having this in mind, consideration is now made of what portions of the described structures are really needed there; in what follows the iterative version will be taken (cf. the algorithm and Fig.2).

At each execution of the main loop, one needs:

- the occurrence lists of C_a;
- the column T_{*b};
- P;
- the occurrence lists of C_b.

First of all, notice that the values, stored in the first column of the C's,are not needed. The first (list header) column is not needed also, if the occurrence lists are stored contiguously.

As to T, retrieving the single column that is needed can be facilitated by storing T column-wise (transposed file organization).

When the sorting process begins, using the last two C's in S, one may at the same time start to bring the other C's in S to faster secondary store.

There are several ways to execute the algorithm, two of which deserve mention. When main store availability is minimal, one may proceed element-by-element, traversing the path and making the changes shown in Figs.2 and 3. If a reasonably large amount of store is available, one can proceed by three stages. In the description below keep in mind that the occurrence lists of C_a are to be traversed sequentially. Hence, they can be conveniently partitioned for loading, whereas the other structures are accessed randomly:

a. take the first partition of the C_a lists and the entire column T_{*b}; place in a partitioned two-column array Q the pairs (r,v); repeat sequentially for the other partitions of the C_a lists;

b. take the first partition of Q and the entire array P (a copy of the second column of C_b); replace (r,v) by (r,P_v) in Q and update P_v; repeat sequentially for the other partitions of Q;

c. take the first partition of Q and the entire lists of C_b; perform the changes in the lists; repeat sequentially for the other partitions of Q.

If all files must be partitioned, advantage can be taken at least of one convenient property. Let F1 and F2 be two files at any of the three stages, F1 being traversed sequentially and the corresponding elements from F2 being found in a random order.

When traversing (sequentially) one partition of F1 containing $\ldots,f_1,f_1',f_1'',\ldots$, suppose that a partition of F2 has been loaded because it contains f_2 which corresponds to f_1. Suppose further that f_1' does not correspond to an element of the current F2 partition but f_1'' does; one can form

(f_1, f_2)
$(f_1', -)$
(f_1'', f_2')

in the auxiliary structure (Q) and leave the second pair to be completed later.

This means that for each partition of F_1 a partition of F_2 will have to be loaded at most once; the reader will note that the property is true even when F_2 is P, whose elements are being updated.

As seen before, the original recursive algorithm does not use T and traverses all the C's sequentially. It does not change the lists _in situ_, and it can benefit from maintaining the lists in the order according to which the n-tuples are actually stored in T (increasing tuple identifiers) which speeds up the intersection operation performed by the algorithm. However, its time can still be quadratic and it needs extra space for the recursion stack.

To summarize, it should be stressed that the performance of an implementation depends critically on wisely placing the required data on available second storage devices.

6 - CONCLUSION

With large data bases, several organizations and techniques can be needed for coping with different situations. What this research indicates is that, wherever the assumed organization is available, the fast sort algorithm is applicable.

This result reinforces the claim that increasingly more efficient implementations of relational data base systems are feasible.

REFERENCES

1 - CODD,E.F. A relational model for large shared data banks CACM, 6 : 377-387, 1970.

2 - SCHMID,H.A. & BERNSTEIN,P.A. A multi-level architecture for relational data base systems. In: International Conference on Very Large Data bases, p. 202-226.

3 - MANACHER,G.K. On the feasibility of implementing a large relational data base with optimal performance on a minicomputer. In: International Conference on Very Large Data Bases, p. 175-201.

4 - PECHERER,R.M. Efficient evaluation of expressions in a relational algebra. Berkeley, University of California, 1975 ERL-M510.

5 - FURTADO,A.L. & BRODIE,M. A data structure for fast relational algebra operations. Rio de Janeiro, PUC, Depto. de Informá tica, 1976. MCSCA 7/76.

6 - AHO, A.V.; HOPCROFT, J.E.; ULLMAN,J.D. The design and analysis of computer algorithms. Reading 1974. Addison-Wesley.

7 - KNUTH,D.E. The art of computer programming. Reading, Addison-Wesley, 1973 v.3.

8 - MARTIN, W.A. Sorting. Computing Surveys, 3 (4): 147-174, 1971.

9 - HELD, G. & STONEBRAKER, M. Storage structures and access methods in relational data base management system INGRES, In: ACM Pacific Conference, p. 148-155