The Performance of a Relational Interface to a Codasyl Database

P. M. D. Gray and P. Esslemont*
Department of Computing Science, University of Aberdeen, King's College, Old Aberdeen, AB9 2UB

A program generator (ASTRID) has been built which generates Fortran code to access records stored in a Codasyl database which satisfies a query written in a generalised relational algebra. The Join operation can be done either by using cost pointers or else by sort-merge. Also projection and grouping operations will require sorts to find and remove duplicate values if they cannot be accessed in sorted order.

A novel method of combining grouping and sorting operations via a sparse array technique is presented. Comparisons are made between this method and separate sorting followed by grouping; timings are given to show the significant advantage in practice.

A number of programs have been generated and run using different access paths, with and without sorts. Comparisons between timings and monitored page charges are made and heuristics suggested for predicting the best method. These are shown to be different from those used by Selinger et al. for the System R relational database.

1. INTRODUCTION

This paper describes an experimental investigation into the performance of programs which retrieve data from a Codasyl database and use a significant amount of sequential access. These are typical of university database usage, where databases grow monotonically and results are obtained by comparing large numbers of sets of data and searching for particular combinations of values that are not always easy to find. The programs have all been generated automatically by the ASTRID system from queries formulated in relational algebra according to Codd's relational model. Since this does not presume any particular access path it is possible to generate several alternative programs for the same query using different access paths through the data. One program may use a direct-access hashing technique followed by a sort. Another may just use nested loops following Codasyl set occurrences, but involving a scan of more records. Another may scan all the data pages in an area so as to reduce the number of page changes. It is important to try to compare these methods in practice on a real database so as to look for heuristics that will reject bad methods and select good ones. Once found these could be embodied in the generator.

We have not tried to build an accurate cost-prediction algorithm for reasons explained in Section 4. Pizarro attempted to build a cost model for a Codasyl system but got very variable results, especially on traversing large sets. Workers with relational databases such as PRTV and System R have had more success in predicting run-times, provided they knew the selectivities accurately. However, they have often had to guess these with arbitrary factors like 0.1 and 0.25 being introduced. Nevertheless, these have been sufficient to rank available methods in very nearly the right order.

Another problem addressed is the handling of run-time sorts. We decided to use our own run-time sort package, which could be closely integrated with the generated code. We do not claim it is as fast as a manufacturer's package for really large sorts; the advantage comes from using it in situations where intermediate sorts are required and most of the sorting can be done in store, using a moderate-sized direct-access file for overflows. This happens in two ways with complex relational queries. First, it occurs when we cannot use a natural Codasyl access path to perform joins, and so we do this operation by sort-merge (as used also by System R). Secondly, it happens where a grouping function is used over data items that are not retrieved a whole group at a time but in some unrelated sequence. Instead of just sorting these items into the right groups we have extended our run-time sort package to combine the group function computation with the sort and so save a lot of time, as explained in Section 5.

It is interesting to compare these techniques with those used by ordinary programmers. Probably they would avoid queries that could not be answered by following cost access paths. The advantage of using sorts is that they make it possible to answer any query in the extended relational algebra used for ASTRID. However, the sort is a double-edged weapon and one must be careful only to use it where necessary and not, e.g. to generate programs that sort the whole database before starting. This is the basic reason for seeking a method that will select good access paths.

2. THE MATERNITY DATA BASE

The results from this study were obtained using a database of maternity information, from which patient names and identification had been removed, for confidentiality. The database contains about 30,000 records of various types. A Bachman diagram of the part under investigation is shown in Fig. 1. The structure is of interest because it includes both a record hierarchy (PATIENT-EPISODE-ABNEPI) and a 'V' structure with a many-many relationship (ABNORM-EPISODE), and this is fairly typical of the structures used in universities.

With reference to Fig. 1, there are about 2000 patient records, each owning between 0 and 33 episode records. Each episode also refers to between 0 and 3 operation records (ABNEPI) which contain one of 100 different operation codes (abencode).

* Now at SCICON, Berners Street, London.
3. Queries and Traversals Considered

The study was carried out on three related queries which are given below as Q1, Q2 and Q3. The queries are given in relational algebra with infix keywords. The group_by operator is explained in Ref. 2. It acts as a generalised projection operation, grouping tuples with the same values of given attributes (e.g. patno) and deriving results as functions of attributes over all these tuples (e.g. afreq = frequency of detection of abnocode 28 over all visits by the given patient). In query Q1 the result relation RES just has two attributes, patgrp and freq, while the intermediate relation AB2E has five, namely patno, visitino, abnocode, abnactn and patgrp; the other three are removed by the projection implicit in group_by.

(*)Q1*) AB28 := ABNFINDBDING selected_on[abnocode = 28] group_by[patno creating afreq := count()]
AB2E := AB28 extended_by[patgrp := patno div 10] RES := AB2E group_by[patgrp creating freq := sum(afreq)]

(*)Q2*) as Q1 but with ABNFINDBDING selected_on[abnocode > 27]

(*)Q3*) AB27 := ABNFINDBDING selected_on[abnocode > 27]
AB2E := AB27 extended_by[patgrp := patno div 10] RES := AB2E group_by[patgrp creating freq := count()]

The first two queries use data from 1053 and 7636 ABNEPI records respectively out of 13065 possible; the results are fed into grouping functions to give a table of 190 tuples as a result. The third query produces the same results as the second but only involves one group-sort operation.

For each query we consider three different access paths, corresponding to the ways into the database. The first (A) is via the system set owning all patient records and leads to a traversal of the entire database in (patno, visitino) sequence. The second (B) uses the ABNORM records as its starting point either by hashing or by a system set and gives a more selective entry. The third (C) uses the ABNEPI records as its starting point and does a basic scan of all the records in an area, thus reducing page changes.

The notion of a traversal is introduced in Refs 2 and 5. It is an abstract description of a piece of code that would generate all the tuples of a relation one by one. It can also be thought of as a description of the access path. The traversal is modified so that it generates one by one the tuples that one would expect from the result of applying Group-by or Join or Extend-by, etc. Its modifications represent the modifications one would make to the corresponding piece of code; for example an extra test inside a loop or an extra variable calculation. The object to generate code that generates the answer by visiting records in the database in turn as required but without storing or materialising any intermediate relations.

The basic traversal for method A is to visit all the different patient records owned by the system record, and for each patient to visit all the episode records, and for
each episode to visit all the ABNEPI records and find their codes as given in the ABNORM owner records. We symbolise this as

$$S(\text{Patient}) \rightarrow D(\text{Episode}) \rightarrow D(\text{Abnepi}) \rightarrow U(\text{Abnorm})$$  \hspace{0.5cm} (A1)

Here $S$ means a singular set traversal by chain of pointers (or by record keys in sequence), $D$ means go down to visit the members and $U$ means go up to find the owner. However, since the code is replicated in the Abnepi record it is not necessary to do the final $U(\text{Abnorm})$ so in general we use the traversal

$$S(\text{Patient}) \rightarrow D(\text{Episode}) \rightarrow D(\text{Abnepi})$$  \hspace{0.5cm} (A)

This traversal is modified in the case of Q1 and Q2 as follows:

- group_by(patgrp)
- $\rightarrow S(\text{Patient}) \rightarrow \text{group_by(patient)}$
- $\rightarrow D(\text{Episode}) \rightarrow D(\text{Abnepi}) \rightarrow \text{select(abncode)}$

The traversal for method B starts by using 'V' or value-based access by CALC to the Abnorn record with code 28, then visits all its Abnepi records and for each one finds the Episode and Patient owners. We symbolise this as

$$V(\text{Abnorm}) \rightarrow D(\text{Abnepi}) \rightarrow U(\text{Episode}) \rightarrow U(\text{Patient})$$  \hspace{0.5cm} (B)

When modified in the case of Q1 and 2 this becomes

- group_by(patgrp)
- $\rightarrow \text{group_by(patient)}$
- $\rightarrow V(\text{Abnorm}) \rightarrow D(\text{Abnepi}) \rightarrow U(\text{Episode}) \rightarrow U(\text{Patient})$

The traversal for C starts by a basic scan visiting all Abnepi records in the area. We symbolise this as

$$B(\text{Abnepi}) \rightarrow \text{select(abncode)} \rightarrow U(\text{Episode}) \rightarrow U(\text{Patient})$$  \hspace{0.5cm} (C)

### 4. PERFORMANCE COMPARISONS

The generated code was run on a Honeywell 66/80 with an IDS-II DBMS. The run times both in terms of Processor time and I/O time are summarised in Table 1, together with the number of records visited and the number of physical reads from the database. The times are in units of thousandths of an hour, taken from the batch job reports. The current charging algorithm weights the CPU time relative to the I/O time by at least a factor of 3, and the last column represents this. The numbers of logical and physical reads are reported by a run-time monitor incorporated in the DBMS as described in Refs 7 and 8.

We can analyse these results in terms of three factors: number of records visited, pages read, and pages used for sorting. Roughly speaking proc-time is proportional to number of records processed, while I/O time correlates with pages read and pages sorted. The use of sort pages is explained in Section 5.

#### Query 1

Here traversal B is clearly best. It uses the 'calc' selectivity to reduce the number of records visited and thus the proc-time, even though the I/O time is not much better than path C. The worst proc-time is for traversal A1, which involves a remarkable increase in I/O time over traversal A. The reason is that the Abnepi records are stored near their episode and patient owners, and the patients are randomised over the disc for hash access. However, there are only 100 Abnorn records and these are stored elsewhere. Thus in the 'innermost loop' of the program (executed 13000 times) we have a 'Find Owner' operation which is guaranteed to cause a logical page change and probably a physical page change too (actually 9010 in total!). This shows the value of duplicating the abncode in the Abnepi record so that we can avoid the expensive A traversal.

In order to use traversal B we have to sort the results, which is not in order by patient. However, the sort cost is very small because there are only 1053 tuples and we use the efficient method described in Section 5.

#### Query 2

Here traversal B is no longer best. The selectivity of the query is greatly weakened and both B and C examine similar numbers of records. In fact C examines all 13000 Abnepi records while B examines only 7600 of them, but they both have to examine a further 7600 episode and 7600 patient owners. By contrast traversal A visits each of 2100 patient records once only. This could be an advantage for the hierarchical traversal. However, in this database half the episode records own no Abnepi records, so the hierarchical traversal visits 2100 patients + 23000 episodes, which is more overall. If instead there were on average many Abnepis per episode traversal A would probably be best.

The reason why B has dramatically more physical reads than C and more than twice the I/O time is because of the fact that the Abnepi records are not clustered with respect to the Abnorn owners. In fact almost every record access can cause a page change. With the basic traversal C every page is visited once only. Thus we see the advantage of clustered access; provided the numbers of records to be visited are similar.

The sort-times are much higher than in Q1 and significantly different in the two cases, even though the same number of values (7636) are being dealt with. This is explained in 5.1. Nevertheless the sort-times are still only a fraction of the total time.

#### 4.1. Problems of I/O time prediction

**Cardinality**

If we know the number of records visited we can get a useful estimate of proc-time, e.g. from Table 1 it is roughly 0.0015 per record. However, I/O time estimates are more difficult.

Table 2, which is taken from Ref. 8, shows the average, minimum and maximum size of each type of set, together
Table 1. Traversal Cost Comparisons

<table>
<thead>
<tr>
<th>Traval. name</th>
<th>Records visited</th>
<th>Proc-time</th>
<th>Log. pages</th>
<th>Phys. pages</th>
<th>Sort pages</th>
<th>IO time</th>
<th>Proc + IO/3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1A</td>
<td>39075</td>
<td>54.6</td>
<td>4237</td>
<td>3785</td>
<td>8</td>
<td>20</td>
<td>61</td>
</tr>
<tr>
<td>A1</td>
<td>52140</td>
<td>80.0</td>
<td>27709</td>
<td>13480</td>
<td>8</td>
<td>94</td>
<td>112</td>
</tr>
<tr>
<td>B</td>
<td>3160</td>
<td>6.6</td>
<td>1837</td>
<td>1101</td>
<td>48</td>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>C</td>
<td>15171</td>
<td>17.2</td>
<td>2859</td>
<td>2397</td>
<td>48</td>
<td>11</td>
<td>21</td>
</tr>
<tr>
<td>Q2A</td>
<td>39075</td>
<td>55.0</td>
<td>4237</td>
<td>3785</td>
<td>8</td>
<td>20</td>
<td>62</td>
</tr>
<tr>
<td>B</td>
<td>23008</td>
<td>36.5</td>
<td>13090</td>
<td>7431</td>
<td>280</td>
<td>37</td>
<td>49</td>
</tr>
<tr>
<td>C</td>
<td>28337</td>
<td>36.2</td>
<td>6291</td>
<td>2542</td>
<td>118</td>
<td>14</td>
<td>41</td>
</tr>
</tbody>
</table>

Table 2. Set statistics for maternity database

<table>
<thead>
<tr>
<th>Set name</th>
<th>Occurs</th>
<th>MIN</th>
<th>MAX</th>
<th>AVG</th>
<th>MIN</th>
<th>MAX</th>
<th>AVG</th>
<th>Find owner</th>
</tr>
</thead>
<tbody>
<tr>
<td>PATIENTS</td>
<td>1</td>
<td>2173</td>
<td>2173</td>
<td>2173</td>
<td>2166</td>
<td>2166</td>
<td>2166</td>
<td>1.00</td>
</tr>
<tr>
<td>PATANP</td>
<td>2173</td>
<td>0</td>
<td>33</td>
<td>11</td>
<td>0</td>
<td>7</td>
<td>1.3</td>
<td>0.34</td>
</tr>
<tr>
<td>PATANPS</td>
<td>23938</td>
<td>3</td>
<td>0.5</td>
<td>0.01</td>
<td>0</td>
<td>4</td>
<td>0.01</td>
<td>0.04</td>
</tr>
<tr>
<td>ABNABN</td>
<td>100</td>
<td>0</td>
<td>1053</td>
<td>131</td>
<td>0</td>
<td>812</td>
<td>94.7</td>
<td>1.00</td>
</tr>
<tr>
<td>SYSABN</td>
<td>1</td>
<td>101</td>
<td>101</td>
<td>101</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>1.00</td>
</tr>
</tbody>
</table>

Table 3. Test of predicted logical reads for Q2B

<table>
<thead>
<tr>
<th>Path name</th>
<th>Access segment</th>
<th>Records visited</th>
<th>Predicted</th>
<th>Actual</th>
<th>Physical pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q2B</td>
<td>S(An)</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td></td>
</tr>
<tr>
<td></td>
<td>D(An)</td>
<td>7636</td>
<td>6913</td>
<td>9041</td>
<td>4840</td>
</tr>
<tr>
<td></td>
<td>≈73*131</td>
<td>=73*94.7</td>
<td>=73*94.7</td>
<td>total</td>
<td></td>
</tr>
<tr>
<td></td>
<td>U(Ep)</td>
<td>7636</td>
<td>305</td>
<td>333</td>
<td></td>
</tr>
<tr>
<td></td>
<td>≈7636*0.04</td>
<td>=7636*0.04</td>
<td>=7636*0.04</td>
<td>total</td>
<td></td>
</tr>
<tr>
<td></td>
<td>U(Pat)</td>
<td>7636</td>
<td>2596</td>
<td>3616</td>
<td>2591</td>
</tr>
<tr>
<td></td>
<td></td>
<td>≈7636*0.34</td>
<td>=7636*0.34</td>
<td>total</td>
<td></td>
</tr>
</tbody>
</table>

with the page changes to traverse it. The set sizes are very variable, for example in ABNBYABN they range from 0 to 1053, with an average of 131. Thus a cardinality estimate made at program generation time may be out by a factor of 10. Figures for other sets in Ref. 8 show similar variation. This causes the variance in Table 3, where we estimate 9563 = (100 – 27)*131 Abnepi records in sets owned by Abnorm records with abcode > 27 but the actual figure is 7636.

Logical page changes

The page changes in Table 2 are ‘logical page changes’, that is, they require a change of page but there is only a physical page change if it is not found in the DBMS buffers. Using a known cardinality and average page changes we can estimate total logical pages. If we apply this method to Q2B how good is the prediction? Table 3 shows the comparison. The figures agree roughly; however, there is a big discrepancy in the Abnepi figures. This is because the tabulated figures are based on a traversal of the Abnepi records with no intervening access to other ones. However, in Q2B we do extra find-owner operations which not only cause 2591 physical page reads but also introduce an extra logical read to bring the current page back from the buffer before finding the next record in the set! This also explains the very large number of logical reads required for Q1A1 in Table 1.

Physical page changes

Even if we could estimate logical page changes accurately the number of physical page changes depends partly on the buffering strategy in use and the number of DBMS buffers (four in this case) and partly on how well clustered the records are. From Table 1 Q2B needs 7431 physical reads while Q2C needs only 2542. If one looks at the reads required for finding episode and patient owners in the two cases the dissimilarity increases. In both cases 7636 Abnepi records are selected, requiring a predicted 2901 logical reads (7636*(0.04 + 0.34)). However, Q2B needs 2591 physical reads while Q2C needs only 198! This is because of the way the database was loaded. Any Abnepi records or episode records which could not fit with their owners were stored on the next page up. Thus with a sequential page scan the owner is either on the current page or else on the previous one, which would still be in the DBMS buffer. Special factors such as this make prediction difficult.

504 THE COMPUTER JOURNAL, VOL. 28, NO. 5, 1985
5. AN IMPROVED SORTING AND GROUPING TECHNIQUE

Consider traversal B for Q3. The straightforward method for coding it is the first shown below. The algorithm is expressed in an abstract language which is more concise than the Ratfor (Rational Fortran)\textsuperscript{12} actually used.

First method: -- Sort then Group

initsort(sortfile, . . . );
for each ABNORM record owned by SYSABN do
if (abcnscd > 27) then
    for each ABNEPI record owned by this ABNORM do
        for the EPISODE owner of this ABNEPI do
            for the PATIENT owner of this EPISODE do
                (patgrp := patno/10;
                write(sortfile, patgrp, visitno, . . . )
            }
        }
    }
for each record in sortfile do
  freq := 0;
  if not(first record in file) and (change in patgrp key)
    then { print(patgrp,freq); freq := 0; }
  freq := freq + 1;
  if (last record in file)
    then print (patgrp,freq);

Second method: -- Combined Group Sort

initialise(A);
for each ABNORM record owned by SYSABN do
  if (abcnscd > 27) then
    for each ABNEPI record owned by this ABNORM do
      for the EPISODE owner of this ABNEPI do
        for the PATIENT owner of this EPISODE do
          (patgrp := patno/10;
          search(A, patgrp, freq, found, posn)
          if (found)
            then store(A, posn, patgrp, freq + 1)
            else insert(A, posn, patgrp, 1);
          }
    }
for each record in A do
  print(patgrp, freq);

The first method writes each tuple of the ABNFINDBINGS relation to a sort buffer; when the buffer is full it is sorted using quicksort and written to a sort work file. Subsequent buffers are treated the same way. In the case quoted the sort buffer has space for 64 tuples and in all there are 1920 pages read and written.

When 'sort' is called the last buffer is sorted and written out. The values are then merged by a binary merge (unless there is only one). The second part of the program then reads back the values from the sort file and counts up the number in each group.

Now consider an alternative method. Suppose we have notionally an array A indexed by patgrp which contains counts initially all zero. For each patient number we have just to access the array element and increment the count at the point in the other program where we wrote the tuple to the sort file. Being a direct-access technique this is much faster. Finally we just scan through the records in A and write them out.

However, we have not actually room for such an array. So we treat it as a sparse array\textsuperscript{4} and use the representation where this is kept as an ordered sequence of (patgrp,count) pairs. We now need a function search (A, patgrp, freq, found, posn) which returns the position posn in the sequence where the value of patgrp is stored (or should be stored) and the boolean value 'found' shows if it is there. We also need a function insert (A, posn, patgrp, n) which inserts the tuple (patgrp,n) at the given position posn. The search routine can use a binary search as the sequence is ordered; the insertion needs to move items in the sequence down to make space. Given these routines we can model the sparse array very effectively, as shown in the second method.

This works well if the sequence is small and can fit in a buffer. However, if the sequence is long we have to deal with it by writing out the existing array and starting again with an empty one. Then we need to read and merge the different arrays as for a sort.

We have written a general package in Ratfor,\textsuperscript{13} which can be called from any FORTAN program at run-time. It will handle sorts (used for some merge joins), sorts with removal of duplicates (used for projection) and also the calculation of grouping functions by this sparse array technique. The method has been generalised to deal with cases where one wishes to group by a multi-level key, e.g. by visitno within patno. It also deals with the case where
several different grouping functions (by the same key) are to be calculated in the same pass through the set of records. The possible functions are:

- count — no. of values
- sum — add values real or integer
- max — find largest real or integer
- min — find smallest real or integer
- all — logical ‘and’ of boolean values
- any — logical ‘or’ of boolean values

All these functions have the very important property that if S is any sequence [x1, x2, ...] made by concatenation of sequences S1 and S2 and f is one of the six functions listed, then f(S) = f(f(S1), f(S2)), where the square brackets denote a sequence of two values f(S1) and f(S2). This allows us to keep starting a new sparse array when the buffer is full. The property is not true of the ‘average’ function, but this can easily be obtained by calculating sum(S) and count(S) in the same pass and dividing one by the other.

We also allow other sort-group processes to start before the first has run to completion. Thus each process uses its own workspace (which is reclaimable on completion of the sort) and the generated code calls the sort routines with a process number. This is necessary for evaluation of queries like Q1 & Q2 with access paths like B or C which invoke nested sorts.

The routines are written to receive information on the sizes and positions of keys at run time. Thus ‘initialise(A)’ in the code is replaced by several calls to pass this information as well as the key field types (real, integer or logical). This facility is often unavailable with standard sorts.

5.1. Performance analysis of group-sort

Code has been generated and run for traversal B for Query Q3 both with grouping (Q3BG) and without (Q3BS). The results are given in Table 4. For comparison results are given also for Q2B, which needs two sort processes.

The following points emerge.

(a) Combined group-sorting reduces sort I/O pages from 1920 to 68 and I/O times from 64 to 38, which is very significant. If we consider the last column we see from Table 1 that Q3BS is even more expensive than Q2A, which searches the whole database. Group-sorting makes Q3BG the cheapest.

(b) The savings in time depend on two factors.

(1) The reduction in tuple width, as we only store the result tuples for grouping (usually smaller than the input).

(2) How often we start a new sparse array. If the tuples are fairly well clustered as in Q2C it takes only 118 pages as against 280 for Q2B. This in turn depends on the sort-page size for each sort process, which is set at 320 words (1 disk block). If this were increased or if there were only 160 tuples in the result then there would be no I/O used. This also explains the 8 pages used for traversal A in Table 1, where apparently no sort is needed. The program does not realise that patgrp is in the same order as patno and so uses a group-sort for the outer group... by which writes two pages which have to be merged.

(c) Q3BS uses very much more proc-time as well as I/O time than Q3BG. It is becoming comparable with that due to database access and marks a limit to the usefulness of the method. We could speed up the sort routines, probably two or three times, but as the sort time is not often a critical factor we have not yet done this.

6. CONCLUSIONS

This is only a study on one database but we believe it shows the strengths and limitations of the sorting technique in conjunction with database access. It also identifies the principal factors regarding the costing of different algorithms.

The combination of grouping with sorting has been shown to be an effective improvement over performing them separately. It is particularly useful where it allows the use of a selective access path such as B to cut the number of records accessed, leaving under a thousand to be sorted. When several thousand records are to be sorted it is better to look for an ordered access path.

A much wider range of queries was asked than has been discussed. In almost all of these the proc-time exceeded the I/O time and seems to be the dominant component, for these kinds of queries with substantial serial access. The remarkable thing is that even with a query like Q2 with a hit rate of just over 50% on the relation it still pays to use a non-clustered index followed by a group-sort. This contrasts sharply with the study by Astrahan et al.,1 where their costs are taken to be of the form

\[ \text{page changes} + (1/3) \times \text{no. of RSS calls} \]

Their RSS calls are like high-level DML calls, which each retrieve one result tuple. Thus in comparing different access methods to a single relation the number of RSS calls is constant and the method with lowest page changes wins. Hence, for any hit rate over a few per cent on their test relation, access by clustered index (corresponding to paths A or C) is best. Our cost estimation is different. We are looking at total DML calls which are much more closely related to real proc-time, and because we have to look at up to four records to reconstruct one tuple (in their terms ABNFINDINGS is like a join of four relations) the number of DML calls can vary dramatically and can be the deciding factor. This is reinforced by the current charging algorithm.

The study1 got good correlation between estimated and actual "costs". However, this is largely because they used a test database with uniformly distributed data values. In a real database, like the one examined, set cardinalities can vary dramatically. Thus we cannot hope for such accuracy. A simple heuristic though is suggested.

(i) Choose an access path that most reduces the estimated number of records visited (on the basis of uniform value distribution).

(ii) If several paths visit similar numbers of records then choose a clustered path in preference to an unclustered one.

(iii) If the method requires a sort, check that the estimated cardinality is below a few thousand; otherwise look for a method that does not require such a large sort.

It may be objected that we could do the sort quicker using a B-tree technique, as is used in some relational databases, rather than by classic quicksort/merge techniques. We have not tried this because we expected to do sorts mainly in store and we could still achieve this by using larger buffers. However, research is in progress.
to use an ASTRID-generated program to output a relation such as ABNFINDINGS, and read it into a RAPPORT relational database and try these queries, using a high-level query language under development at Aberdeen University Computing Centre. This will establish the relative value of the two techniques.

REFERENCES


7. ACKNOWLEDGEMENTS

We wish to thank the Science and Engineering Research Council for a grant supporting this project. We are very grateful to Aileen Stone for assistance in using her database and for the monitoring programs, and also to Robert Bell for help and inspiration in implementation.


The Wilkes Award 1984

The Society is pleased to announce that the Wilkes Award for 1984 has been won by four authors of a paper ‘The Rainbow Workstation’ published in the May 1984 issue of The Computer Journal. The paper was jointly written by six people, of whom four were eligible under the terms of the award, namely that they should be under 30 years of age at the time of submission of the paper.

The paper describes an experimental workstation that supports high-quality colour raster graphics in a distributed environment. Hardware assistance for screen management is provided, using a method that allows rapid reconfiguration of screen images. Much of the workstation function is defined by high-level software and loadable microcode, thus permitting the exploration of different ways of using this kind of equipment.

At the time of the paper being written the four authors, John Wilkes, Dave Singer, Jon Gibbons and Tony King, were research students at Cambridge University. The work on which the paper was based arose from discussion with the other two authors, Dr Neil Wiseman, supervisor of the team, and Dr P. Robinson, Director of Research in the Cambridge University Computer Laboratory.

The number of papers eligible for the award this time was consistent with last year, although this had increased from previous years. All the papers were of a high standard, with the final decision being taken between two on completely different subjects. The judges agreed that in future preference should be given to papers of which all authors were under 30. The judges considered that the Wilkes Award was achieving its primary objective: to encourage the development and publication of research work in computer science and applications among young people.

The award was presented to Mr King at a ceremony hosted by Computer Weekly. Mr King is the only one of the four authors still resident in this country. He is halfway through his PhD course at Cambridge University, where his thesis will describe the successor to the Rainbow workstation reported in the Journal. Dr Wilkes is now with Hewlett-Packard Laboratories, Dr Singer is with Fairchild Camera and Instrument Corporation and Dr Gibbons works for Acorn, all in Palo Alto, California. Certificates will be given to the four winners.

The Wilkes Award, which consists of a silver gilt medallion, was instituted by The British Computer Society to mark the retirement of Professor M. V. Wilkes as Professor of Computer Technology at the University of Cambridge, in recognition of his pioneering work in both computer hardware and software and his unceasing efforts on behalf of the Society.

THE COMPUTER JOURNAL, VOL. 28, NO. 5, 1985 507