FOURTH GENERATION LANGUAGES AND PROCESSING EFFICIENCY

John D. Alleyn-Day
A H Computers,
8210 Terrace Drive,
El Cerrito, CA 94530-3059 USA
415-525-5070

Fourth Generation Languages have great power and can be used to write processing programs easily and quickly. However, they also have a reputation for being extremely inefficient -- a reputation which is undeserved. Many programs written in fourth generation languages are inefficient because the programmer is tempted to use programming methods without really understanding what the language is doing.

I am going to discuss a particular situation that arose using RELATE/3000. The same situation could have arisen with several diffferent languages, and with many different data situations. Since the solution would have been similar in each case, I want to share it with you.

I have been working extensively with the Sierra Club, who use RELATE/3000 together with IMAGE for a significant part of their processing. The fourth generation language is often used to access data from both RELATE/3000 files and from IMAGE databases. The Sierra Club has run into devastating problems with efficiency. Some batch programs have taken all weekend to run, just to turn out a report. In some cases, the time needed was so extreme that the jobs were aborted so that other users could get their share of computer resources! In this environment, my task was to put together several batch programs that would be producing statistical summaries of data from files with 500,000 or more records each. I was facing a serious problem of efficiency and, in working out a solution, I learned a great deal about fourth generation languages.

The major conclusion to come out of my work was that the inefficiencies were not necessarily an integral part of the fourth generation language but rather of the way in which the language was used. You see, the simplicity of the programming methods encourages programmers, myself included, to construct very inefficient programs without realising the true import of their code. I will illustrate this for you as we go along.

In order to understand the example I am going to work with, let me give you some background about the data structures that are involved. We will be looking at three files. The first is "grd" which is an IMAGE detail set. It has a unique key of "resource". The second file is "grdx" which is a Relate/3000 file and is a type of history file, with old data from "grd". The third file is called "detail" and contains details of money amounts. The "detail" fields with which we are mainly concerned are

"resource", date, and amount.

We want to get the field "amount" from "detail" and collect it together on the basis of the "fiscal year" (which is a function of "date"), "class" which comes from "grd" and "prev_class" which comes from "grdx". We also need the "resource" for the time being as we want the count of the number of records in "detail" for each resource as one of our parameters. This was the first step in a series that ultimately produces a small statistical report. We shall not concern ourselves with the rest of the program as it is of no interest to our present problem. The following example of Relate/3000 code shows how we can do this in a few lines.

```
open file grdx.data;mode=read,share
open database gendbz.dba.para;type=image;mode=5;pass=READPW
open set grd;database=gendbz.dba.para
modify field grdresourc; name=resource
open file detail.donors;mode=read,share


select &
    detail.@, &
    fiscal= &
      $integer($substr($year($new_date(date,+92)),3,2),2), &
    memb_cnt=0, &
    rcount=0, &
    resource, &
    class=grd.grdclass, &
    prev_class=grdx.prev_class &
  by fiscal,class,prev_class,resource &
  where grdx.resource=grd.resource &
    and grdx.prev_class<>"" &
    and grdx.resource=detail.resource


consolidate &
    fiscal:f &
    class:f &
    prev_class:f &
    amount:t &
    memb_cnt:f &
    rcount:c &
  to testdata;records=150000 &
  by fiscal,class,prev_class,resource
```

The select statement is asking RELATE/3000 to read the "amount" data from the "detail" file, pick up the "class" from the corrresponding "resource" record in the "grd" file, and the "prev_class" from the "resource" record in the "grdx" file when "prev_class" is not blank. Each of these files is at least 500,000 records, but from our familiarity with the data we know that only about 15% or so of the resources will have a non-blank

"prev_class".

A program like this is very straightforward in concept and easy to put together. This is the program as I first wrote it. After it had run for the better part of a weekend and was still incomplete, we had to abort it on Monday morning to release computer resources for other users.

If I had written a COBOL program to solve this problem, it would have taken only a few hours. In frustration, I asked myself, "Is this inefficiency an inescapable problem associated with the relational database and the fourth generation language, or is there something that I can do about it?".

In working out a solution I learned that there is a cardinal rule which must be applied. KNOW WHAT YOUR FOURTH GENERATION LANGUAGE IS ACTUALLY DOING. Why does it take so long?

This is what I found out that RELATE/3000 was doing. In my program, the language will select one of the files to read sequentially (usually the shorter one) and, for each record, use the key "resource" to access the corresponding data in the other files. In my particular case, one of the RELATE/3000 files, "grdx", was the shortest, and was therefore the one that was read sequentially. For each of the 500,000 records there is likely to be two disc I/O's for "GRD": one to read the master and the second to read the detail in the IMAGE database. Additionally, for the "grdx" record, there will be about 2 or 3 disc I/O's on the "detail" file, 1 or 2 for the key and one for the data record. (This number depends on the size and the randomness of the files.) A quick calculation gives 2.5 million I/O's, and at 20 I/O's per second, these reads alone will take about 35 hours.

I must then add a few hours for sequentially reading the first RELATE file and for writing the new one. Furthermore the data must be sorted and this is another place for inefficiency. Relational database systems usually sort a file by finding an existing index that is suitable or by creating a new index. In my case the sort key is made up of data elements from different files, so Relate/3000 cannot use either of these options directly. Instead it starts by copying the data needed into a temporary file and then creating a new index for that data. Like IMAGE, Relate/3000 attempts to protect the data integrity by forcing the data to disc after each write. This will account for about 80,000 I/O's, taking about 2 to 3 hours. The keys also have to sorted and written out. The data is then read by key from this temporary file, totalled and written to the new file. It is not hard to see why this program could easily take several days to run.

This tendency to permit inefficient programming is not the characteristic exclusive to RELATE/3000. Let us look at the same basic program written in QUIZ. The following statements achieve approximately the same result (substituting IMAGE masters for the "detail" file and the "grdx" file).

```
access grdx &
  link to resource of detail &
  link to resource of grd

select if prev-class of grdx = " "

define n-date numeric*6
     = ascii(date((days(d-date of detail) + 92)))
define fiscal numeric*2 &
     = n-date[1,2]
define memb-cnt numeric*7 = 0
define rcount numeric*5 = 0

sort on fiscal, class of grd, prev-class of grdx,
  resource of grdx

report summary &
  fiscal &
  class of grd &
  prev-class of grdx &
  amount of detail subtotal &
  memb-cnt &
  rcount

set subfile at resource name testfile
```

This code will carry out a very similar process to the one that the RELATE/3000 code performs. The estimates of the number of disc I/O's obtained for RELATE/3000 apply equally well to QUIZ. The sort considerations are somewhat different. QUIZ uses a record complex made up of the join of all the data and sorts it as one huge record. Because of the large record being sorted, it is unlikely that the method used in QUIZ will be, on average, any more efficient than the method used by RELATE/3000. In any particular case, the relative efficiency will depend on the size of the data record, the size of the key, and other factors.

So the major part of the inefficiency of the processing is not dependent on any specific fourth generation language, but rather on the processing methods that are generally encouraged by fourth generation languages. Specific methods for improving this performance depends on the particular language used, but the general approach is the same. I will illustrate my methodology using the RELATE/3000 example, leaving you to make the necessary adjustments to achieve similar results in your own language.

Now that I know why my program takes so long to run, I can set about making it run faster -- much faster. Twenty or thirty percent improvement in efficiency will not be enough; I need it to run five to ten times faster. For this phase of my work, I adopted another rule, "Use batch techniques for batch programs". This shouldn't be anything new. The "Image Handbook" in the

chapter called "Throw off your Chains" contains lots of hints for handling database files in a batch environment. The fact that this is a fourth generation language rather than a third generation language shouldn't make much difference. In my original program I totally ignored the tenet "paths should be reserved for on-line users". The major reason for the poor performance is the keyed reads that are being carried out to obtain data from secondary files. How can I avoid this?

Let us go back a few years to the days of punched cards and sequential files on tape. Without keys and chains, there was no possibility of doing what I have done here. Instead, we used all kinds of processing tricks to get the answers in the most efficient way possible, usually making considerable use of the system sort, record sort breaks, and matching record keys. I can use that experience now to carry out a similar process.

I thought about how I would have written a COBOL program to do the same job. I would have read each file in turn, extracting the fields I wanted and then released the records to the sort, sorting on my key values. I would have arranged the sort so that the records from the different files were sorted together by "resource"; then I would have matched the records and created my output records, each of which would have been a composite of several of the input records. It might have been necessary to sort again before totaling, or, if the final results were sufficiently compact, I might have created a table in memory to accumulate the answers.

The answer to my problem is to do something similar here. Because of the intrinsic limitations of the various software tools, I will not be be able to achieve the same efficiency as a COBOL program, but I can approach it. I can do as much as possible with serial reads and extracted data. I must avoid using paths through datasets which, although they are excellent in on-line situations, are a disaster in batch processing. Also, to simplify things, I will deal with the files two at a time instead of all three as I might in a COBOL program. The first step is to do two extracts and a sort.

I start by reading serially through the "grdx" dataset and extracting an MPE file consisting of only those records and fields that I really need. Just as if I were writing COBOL, before starting I generate my "sort" record layout with the following piece of code.

```
create file sample;records=0;fields= &
    (fiscal,i,2), &
    (class,a,2), &
    (prev_class,a,2), &
    (resource,d,8), &
    (amount,d,10), &
    (memb_cnt,d,7), &
    (rcount,d,5)
```

Now I copy the data I want to my first work file with the following code.

```
open file grdx
copy rea.rcount=1 &
  to rea.data;type=mpe;structure=sample;records=300000 &
  for prev_class<>""
```

This process takes about one and a quarter hours.

Extracting IMAGE datasets is, of course, a job for SUPRTOOL. If you have IMAGE datasets this big and are serious about improving your efficiency, you must have SUPRTOOL, which can also be used for the sorting phase.

Now I switch to SUPRTOOL, extract the IMAGE dataset, appending it to the previous one, and then sort. (I make use of a field "rcount" that is not being using at the moment as a record-type indicator.) The resulting file will be sorted so that, for each "resource", there will be one record from "grd" and sometimes a preceding record from the "grdx" file. The following code achieves this.

```
base gendb
get grd
define f,1,2,integer
define p,1,2,byte
define a,1,4,double
define n,1,4,double
define g,1,4,double
extract f=0
extract grd-class
extract p="  "
extract grd-resource
extract a=0
extract n=0
extract g=2
output rea.data,append
xeq


input rea.data
key 7,4,double;19,4,double,desc
output reb.data
xeq
```

The extract took about 10 minutes and the sort about 20 minutes.

RELATE/3000, in common with most fourth generation languages, can operate on MPE files as well as on its own files. I use functions to "combine" the records and create a new file with the

combined records. My first step is to run through my new file getting the "prev_class" field from the records of type 1 into the "prev_class" class field of the records of type 2. How this is done will vary considerably with the particular fourth generation language that is being used and with the format of the data being processed. In my example, I used the following code.

```
open file sample
open file reb.data;type=mpe;structure=sample
let prev_class=$last(prev_class,resource)
```

The "let" statment holds the data in "prev_class" from one record to the next, resetting it to blanks when the "resource" key changes. The overall effect is to blank out "prev_class" in the records of type 1 and to move the value from that record to the records of type 2. This process takes about 30 minutes.

From here I could have proceeded in a variety of ways. One possibility is to copy the type 2 records which we want to another file and handle the "detail file" in a process similar to what I have discussed, namely extracting it to an MPE file, sorting and combining. However, I will actually get about 80,000 records from this file, and it takes only about twice as much time to link this file to the "detail" file using standard RELATE/3000 code as it does to extract the "detail" file, combine and resort. Because this was a once only report, I chose to go back to standard fourth generation language techniques as follows.

```
create file rec.data;structure=sample;records=250000
modify file rec.data;crashproof=no;compress=no;scan=0
close file rec.data

open file reb.data;type=mpe;structure=sample
open file detail.donors;mode=read,share
select &
    reb.resource, &
    reb.class, &
    reb.prev_class, &
    detail.@ &
  where reb.prev_class<>"" &
    and reb.resource=detail.resource
copy &
  rec.fiscal= &
    $integer($substr($year($new_date(date,+92)),3,2),2), &
  rec.memb_cnt=1 &
  to rec.data
```

This process took about five and a half hours.

From here on the files are getting progressively smaller, and the use of the standard fourth generation language procedures will

not be seriously time-consuming.

This raises a final point. A programmer must use judgement in applying the techniques I have illustrated. On small files the increased efficiency possible with my techniques will probably not repay the time you spend doing the additional analysis. However, if you run a program very frequently, analysis and reprogramming for greater efficiency may be very valuable, even if small files are involved. Some installations run small reports everyday at lunch-time in preparation for the afternoon's work. In such a case, the extra effort to increase speed can be justified.

Finally, I suggest that the Fourth Generation Language Developers consider this problem. Many customer representatives claim that their systems run batch programs. This is true -- in a way. Fourth Generation Language programs can be run in batch, but as I have demonstrated, they use on-line techniques most of the time. This should be changed. Language statements used by the fourth generation languages do not necessarily stipulate the processing actually carried out. The two examples that I used from RELATE/3000 and QUIZ now imply the use of keyed reads, leading to inefficient batch programs. Why could not a Fourth Generation Language interpret these examples as extracts, sorts, merges and record matching, similar to the processes that I actually used? A fourth generation language that could choose its processing method based on whether it was considered to be batch or on-line could achieve a substantial improvement in efficiency, and an increased market acceptance.

So far, the forth generation language vendors have not seen this as a problem that they need to address. However, there is one group that has stepped into the breach, namely Robelle. They are just bringing out an addition to SUPRTOOL called SUPRLINK, which carries out the matching of records in an efficient manner. As of this writing the program is in beta test. I have not had time to test it , but it appears to have all the cpability necessary to solve the problem that I have described here.

To sum up, if you are having problems with your fourth generation language efficiency, there are two steps to follow. First, understand exactly how your fourth generation language operates and carries out its processing. It may take quite a bit of work to get this out of your fourth generation language supplier or to do the detective work to find it out on your own. My advice is, "Be persistent". Secondly, make use of batch techniques for your batch programs and not the on-line techniques that you may be seduced into using by the your fourth generation language. And, of course, use your judgement as to when it is worth the trouble and when it is not.