ABSTRACT

One of the most useful features of SAS® is its capabilities for handling large datasets. However, a sort of a very large dataset on many keys can present difficulties. Disk space or temporary space may be inadequate. The time needed may be enormous. The approach taken may overwhelm the software or the operating system. This talk covers multiple ways to sort on a large dataset with many sort-keys, pointing out some advantages and tradeoffs of techniques.

THE ORIGINAL PROBLEM

While SAS® is efficient and scales better than many alternative software solutions, it still has its weak points. One problem encountered in routine work was that of sorting large datasets on many keys.

In the Environmental Protection Agency’s (EPA) Office of Research and Development is a large-scale synoptic monitoring program known as EMAP: the Environmental Monitoring and Assessment Program. EMAP has used GIS-based triangular grids to create probability-based sampling designs for environmental monitoring. Triangular point grids based on equal-area projections yield a tesselation of equal-area hexagons over the landscape. This allows EMAP to build sampling designs which the spatial structure of the resource of interest, as well as providing sampling proportional to the extent of the resource.

The sampling is done as a systematic sample (with a random start) of weighted elements on a line, where the positions on the line are generated by a hierarchical randomization of the original two-dimensional space. This requires the building of a hierarchy of sort-keys, and then sorting the dataset on multiple keys. A large sample – every perennial stream segment within every 1.8 square kilometer hexagon across twelve western states of the conterminous United States – produced a dataset which strained the limits of a high-end Solaris box running SAS®. We ended up with a 1.2 Gigabyte dataset: 3.4 million records and 44 variables. Even though we got a bigger Solaris machine and solved our problems in other ways, this issue continues to be important.

So, how does one effectively sort a 1.2 Gigabyte dataset on 13 sort-keys? As an aside, we first found that the default installation of SAS® on Solaris – and many other unix systems – places the saswork directory within the /tmp filesystem. This can be a serious problem when working with large datasets, as /tmp is usually a very small partition, often with little more than a dozen Megs of free space. And filling this up can bring the unix OS to a screeching halt. So the first thing we learned was to allocate a lot more space to saswork.

SORTING OPTIONS

There are a number of options available for sorting a dataset. There are faster sort options. There is of course PROC SORT. There is PROC SORT with the TAGSORT option. PROC SQL can also sort a dataset. And PROC DATASETS can generate an index, which can give the same effect as a sort. In addition, there are two extra elements to consider: the effect of compressing the dataset, and the effect of packing the sort-keys.

On big iron, there may be host sorts which are faster than PROC SORT. And the user may have PROC SYNCSORT® licensed. SyncSort is uniformly faster than the native SAS® sort, and should always be used when available. A host sort can be accessed merely by inserting the following line of code into your program:

```bash
options sortpgm=host;
```

It should be noted that the use of the TAGSORT option in PROC SORT will negate the SORTPGM=HOST option. And there are special sorts which assume prior knowledge of the sort-keys, like radix sorts and the counting sort, both of which scale linearly.

The SAS® sort routine is of order O(NlogN), which is as fast as a comparison sort can be. PROC SORT can use the TAGSORT option like this:

```bash
proc sort tagsort data=emap.frame;
   by rp1 rp2 rp3 rp4 rp5 rp6 rp7 rp8 rp9
      rp10 rp11 rp12 rp13;
run;
```

The tagsort option was introduced in version 6.07 and can yield significant improvements in clock time, while typically increasing the CPU time. It causes the sort routine to store only the sort-keys and observation numbers in the SAS® temporary files. These keys and record numbers are the ‘tags’ of tagsort. Then, at the end of the sort, the tags are used to retrieve entire records from the original data set in sorted order. This has potential gains whenever the total length of the sort-keys is small compared to the record length. Temporary disk usage is significantly decreased.

One can also create a sorted data set with PROC SQL:

```bash
proc sql;
create table emap.t1 as
   select * from emap.frame
      order by rp1,rp2,rp3,rp4,rp5,rp6,rp7,rp8,rp9
      rp10,rp11,rp12,rp13;
quit;
```

And, rather than sorting the full data set, one can use PROC DATASETS to build an index which SAS® can
then use as if the data set were fully sorted.

```sas
proc datasets library=emap;
   modify frame;
   create index rplist = (rp1 rp2 rp3 rp4 rp5 rp6 rp7 rp8 rp9 rp10 rp11 rp12 rp13);
run;
```

**INDEXING INSTEAD**

Indexing will be faster than sorting, and the difference will be significant on a large data set. But you will end up with a large index file too. The above program working on our 1.2 G data set creates an index file that is roughly 340 Megabytes.

Indexing has other important advantages. Just as a dataset sorted on RP1 RP2 RP3 can be used as if it were sorted only on RP1, or only on RP1 RP2, a dataset indexed on k1 variables can be used as if it were sorted on just the first k2 of those variables. One important addendum: when building an index, one can build several orthogonal indices, so that later procedures and data steps can use the data as if they had been sorted into the requisite orders each time.

**An Index with a Packed Key**

The idea of creating a packed key out of multiple keys is straightforward. If one uses the unique part of each key in turn and welds those parts together, one can get a single key which contains all the needed information to create the multiple-key sort. Here is a naive approach to creating a packed key for sorting this data set. Each of the variables RP1 to RP13 is a random number between 0 and 1, stored in the standard default of eight bytes. Let’s create a single packed key of the same width:

```sas
data emap.frame2;
   set emap.frame;
   length rplist $ 104;
   array r{*} rp1-rp13;
   rplist = "";
   do i=1 to 13;
      rplist = trim(rplist) || substr(put(r{i},10.8)3,8);
   end;
   drop i rp1-rp13;
run;
```

This code takes a series of numbers, rounds them off at eight decimal places, and combines only the part after the decimal into the packed key. It looks like this:

```plaintext
rp1 = 0.8101551308
rp2 = 0.5367136349
rp3 = 0.7649560444
rp4 = 0.2447885565
rp5 = ....
```

so rplist =

```
81015513536713637649560424478856 ...
```

Note that the last digit for RP4 has been rounded off appropriately for the 10.8 format.

The above code has a few disadvantages. We can avoid building an array and unroll the loop if we use a macro solution instead.

```sas
%macro listmake(max=,len=);
   %local i l2;
   %let l2 = %eval(2 + &len);
   %do i=1 %to &max;
      ||substr(put(rp&i,&l2..&len),3,&len)
   %end;
%mend listmake;
data emap.frame2;
   length rplist $ 104;
   set emap.frame;
   rplist = %listmake(max=13,len=8);
   drop rp1-rp13;
run;
```

We’ll cover the improvement in execution time later, when we make a further improvement in the packing of the keys. For now, note that this key is a string of 104 characters, so that the new dataset is almost exactly the same size as the old one.

**COMPARISONS**

All the following time comparisons were done on a Sun SPARC Ultra 60 running Solaris 7 and SAS® 6.12 TS020. This machine has 1.2 G main memory, 2.5 G virtual memory, and a /tmp filesystem of 2.5 G. All times are averages of 5 runs while the machine was under light load. The first time is the wall-clock time – the actual real time required to complete the task. The second time is the CPU time – the time required for CPU usage only, ignoring all multitasking, I/O, etc.

<table>
<thead>
<tr>
<th></th>
<th>13 keys</th>
<th>1 packed key</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sort</td>
<td>11:06 / 3:37</td>
<td>10:37 / 3:33</td>
</tr>
<tr>
<td>Tagsort</td>
<td>11:07 / 4:52</td>
<td>10:41 / 4:49</td>
</tr>
<tr>
<td>Index</td>
<td>3:33 / 1:17</td>
<td>3:21 / 1:10</td>
</tr>
</tbody>
</table>

No PROC SQL results are reported here. PROC SQL needed even more memory and temporary space than was available on this workstation, and could not sort the given data set under these conditions. Simulations suggest that the actual time required to sort the data set using PROC SQL would have been roughly that of using the naive PROC SORT.

Notice that indexing instead of sorting had the greatest effect on the time required. Indexing gave about a 3-fold improvement. Using the packed key gave about a 5% improvement in each case.

Using the COMPRESS=YES data set option reduced the size of the data set by about 15%. It also increased the time needed by about 5%, except in one interesting case. It uniformly improved the time needed for the tagsort by about 10%. This may be due to the fact that the random numbers of the sort-keys should have strings of two or more identical numbers occurring in
about 10% of the decimal places throughout, so there
may be more compression of the sort-keys and less I/O
overall.  The utility of the COMPRESS =YES option
becomes a tradeoff in other cases.

Now before we move further, consider the packing of
the keys once again.  The random numbers are used to
order each successive level of the hierarchy of the grid
structure.  But at no level does the enhancement ever
create more than seven hexagons at the next level.  In
most of the levels, the enhancement is a factor of three
or four.  The sort-keys are numbers picked out of a
U[0,1] distribution, so they will be relatively sparse for
any hierarchical unit.  So we do not actually need to
keep all eight decimal places, as we did when we built
the original packed key.  Two or three decimal places
will be enough for this specific problem.  Let us
demonstrate this with four decimal places, which should
be enough to distinguish the values within any level for
any given unit.

```
data emap.frame2;
  length rplist $ 52;
  set emap.frame;
  rplist = %listmake(max=13,len=4);
  drop rp1-rp13;
run;
```

Note that using a more densely-packed key will also
make a smaller dataset.  For large data sets (where the
size of the header is relatively tiny) the size of the data
set decreases roughly proportionally to the decrease in
record length.  Here the data set shrinks by about 16%,
which is better than we did using COMPRESS=YES.
However, this size reduction will improve the execution
speed rather than impacting it.

First let’s see how using the macro method instead of
the original array method affects execution speed.  For
this case we get the following wall-clock and CPU times:

<table>
<thead>
<tr>
<th></th>
<th>array method</th>
<th>macro method</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>3:58 / 2:47</td>
<td>3:16 / 2:03</td>
</tr>
</tbody>
</table>

The array method takes about 21% longer by wall-clock,
and about 36% more CPU usage.  So the macro
method has a significant advantage.

Now let’s look at the improvements in speed going from
the original unpacked methods to the methods using a
single densely-packed key.

<table>
<thead>
<tr>
<th></th>
<th>13 keys</th>
<th>dense-packed key</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sort</td>
<td>11:06 / 3:37</td>
<td>7:12 / 2:26</td>
</tr>
<tr>
<td>Tagsort</td>
<td>11:07 / 4:52</td>
<td>7:43 / 4:17</td>
</tr>
<tr>
<td>Index</td>
<td>3:33 / 1:17</td>
<td>1:44 / 0:46</td>
</tr>
</tbody>
</table>

From the original naive sort to an index using a densely-
packed key, there is a six to seven-fold improvement in
speed.  Furthermore, there is slightly better than a two-
fold improvement in wall-clock time going from the
original indexing to the new index using the densely-
packed key.  There is also a further saving in disk
space: the new index takes about 179 Megs, which is
only 54% of the size of the original index file.  This is
hardly surprising when the densely-packed key requires
only half the storage space of the original thirteen keys
– or the original packed key.

As a brief aside, consider this same problem on a much
smaller machine.  Testing using the same dataset on a
Pentium II-MMX running Windows 98 found that the
same relative times still held.  All the procedures and
data steps required roughly a five-fold increase in wall-
clock time.  But the index still took significantly less time,
and the use of the densely-packed key led to
improvements in both time and disk space usage.

And finally, there is a further important point to bear in
mind.  While indexing speeds up the sort, it increases
the time needed to manipulate the data set in a
subsequent step.  The cost of using an indexed rather
than sorted data set varies depending on available
memory, disk space, and access bottlenecks (such as
accessing data over a network instead of from a local
hard drive).  But based on the tests I ran, I would
recommend that accessing a data set seven or ten
times using an index will cost roughly as much in
available resources as sorting the data set and
accessing the sorted data set the same number of
times.  This does not consider the case in which you
build multiple indices at the same time, so that you do
not need to do any subsequent sorts before data set
accesses.

CONCLUSIONS

In summary, there are six relevant points to be made:

1.  When working with large data sets, index them
    rather than sorting each time you need a
different order of variables.
2.  Index on a packed key instead of building a
    multiple-key index.
3.  Pack your keys as tightly as possible – but no
tighter!
4.  Clean up your workspace as much as possible
    at each step.  Remember that your temporary
files are also going to be huge.
5.  Don’t index if the data set will be sorted only
    once and read many times using that sort
    order.
6.  And, of course, use macros to unroll those
    data-step loops whenever you can.
Acknowledgements

SAS is a registered trademark of SAS Institute, Inc. in the USA and other countries. SyncSort is a registered trademark of Syncsort, Inc.

Contact Information

The author may be contacted by mail at

David L. Cassell
OAO Corp., c/o U.S. EPA
200 SW 35th St.
Corvallis, OR 97333

or by e-mail at

Cassell.David@epa.gov