Paper HW-02-2013

Fast Access Tricks for Large Sorted SAS Files

By Russ Lavery

ABSTRACT:

There are techniques for file access, *of sorted files*, that are described in the IT or Comp Science literature that most SAS programmers do not study or consider for their use. This is, in general, a time efficient decision on the part of the SAS programmer because we can trust SAS Institute to bring in useful techniques (e.g. indexing and hashing) and to make them convenient for us to use. However, there are situations where coding your own access method can be faster than a vanilla SAS access method. In addition, studying these techniques can be a reward in itself – sometimes giving a programmer a new view of how SAS works. This paper will show nine techniques that can be used on sorted files. Your performance will vary with the details of your SAS installation.

Code illustrating these techniques will be provided in a SAS abbrev file that can be installed by anyone (admin rights not required) on their PC. This means a reader can easily access, and play with, these techniques at their leisure.

INTRODUCTION:

Thanks to Paul Dorfman and Mark Keintz for sharing these techniques and putting them in the public domain.

SAS, while a well-designed fourth generation language, and complete in itself, does not exist independently of other programming languages/techniques. While the author is grateful that he has never had to program low-level tasks (e.g. programming his own sort routine), it is, occasionally, useful to read and study some lower-level techniques that exist in the main stream of what might be thought of as IT -- and to bring the techniques to SAS.

Many of the techniques in this paper involve programming a binary search in a SAS data step. In a binary search on a sorted table (sorted by some "key variable like subject_id) the algorithm picks the observation in the middle of the data set and checks to see if the subject_id found is the desired subject_id. If it is, the algorithm stops. If not, since the table is sorted, the algorithm can determine if the desired observation is "above" or "below" the one found and re-defines the search range to be half of the original file . The algorithm picks the observation in the middle of the new search range and the process repeats. If the desired subject_id is not in the file, the process will repeat until there is "no more file left to divide" and will stop.

Most of these techniques can only be applied to data sets with certain characteristics and will produce errors or erroneous results if applied to data sets without those characteristics. The important characteristics to consider are: 1) is SMALL and/or LARGE sorted and 2) do SMALL and/or LARGE contain observations with duplicate values of the key variables (can more than one row have the same value of the merge or key variable).

A binary search for a subject_id, that is not in the table, will take floor($\log_2$(N)+1) searches. $\log_2(N)-1$ is the expected number of searches if the subject_ID is the table and the maximum number of iterations in a successful search is just one more, or $\log_2$(N), Searching a table with a million rows will never take more than twenty iterations and so searching is relatively insensitive to N.

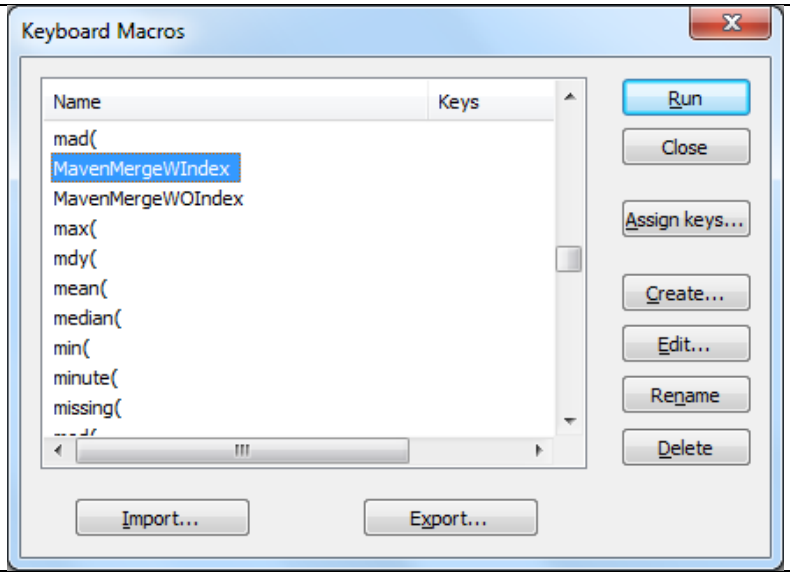
Since binary searches are fast, and insensitive to both the file size and gaps in the distribution of subject_ids in the table, one might ask "why does SAS not provide this feature?". The answer is that SAS *does* provide this feature in ways that are hidden. SAS uses binary searches inside PROC Format and inside a SAS index. With the fact that SAS uses binary searches internally, as a recommendation for the usefulness of the technique, this paper will show how to apply this technique through data step coding.

This paper will show nine different techniques and the code can be found in the abbrev file provided.

To access the sample code in the abbrev file you must install the abbrev file (see my web site russ-lavery.com). After installing, select: tools → Keyboard_Macros → Macros and you will see the box to the right.

Scroll up and down until you find the example code you wish to explore, select the code (background will turn blue) and click run.

The sample code will be pasted into your editor.



While one, static picture is not a good replacement for the dozens of animated PPT slides in the lecture, pictures of individual slides are provided in the paper because: 1) they show both data files (LARGE and SMALL) 2) they often show the PDV 3) they often have little circles overlaid on the pictures of LARGE that show the limits of the searchable rows 4) they often have annotations, pointing to specific lines of code. It is hoped that these single PPT slides will be useful to the reader.

Techniques for which code is provided are:

Example 1: MavenMergeWOIndex
Example 2: MavenMergeWIndex

Example 3: Binary_SmallNotSortedLargeSortedNoDupes
Example 4: Binary_SmallSortedLargeSortedNoDupes
Example 5: Binary_DoNot DeleteMismatches
Example 6: Binary_DoNot DeleteMismatches

Example 7A: Compressed_Index_Basic
Example 7B: CompressedIndex_Index
Example 7C: Compressed_Index__IORC_into_The_Compressed_Index

There are some general principles, or patterns, we should look for as we study these programs.

1) If you reset L (Low end of range) and H (High end of range) to 1 and N for every observation in SMALL
    a. For every obs. in SMALL, you will check ALL of LARGE.
    b. You do not need to sort SMALL.
    c. You can have duplicates in SMALL and get good results.
2) If the algorithm you pick has a “by merge” in it:
    a. You must sort, or index, SMALL and LARGE.
    b. You get a typical SAS merge. Remember that many to many merges are dangerous.
3) If you have duplicates in LARGE, you must add code to look “up” and “down” large, to check for duplicates, after you find the first “Matching observation”
4) Many to many merges are a problem in SAS and if you have duplicates in both SMALL and LARGE, you might have a problem in the program specifications that needs to be thought out.
5) Compressed indexes are cool and speedy and are designed to handle duplicates in LARGE.
6) You can combine these techniques to make things really hard on the next programmer to follow (e.g. putting an index on the compressed index and using a where clause is, I think, a bit tricky).
7) If small is sorted you can use that fact, and some coding so that you do not have to search ALL of large for every observation in SMALL and save time.

EXAMPLE 1: MavenMergeWOIndex

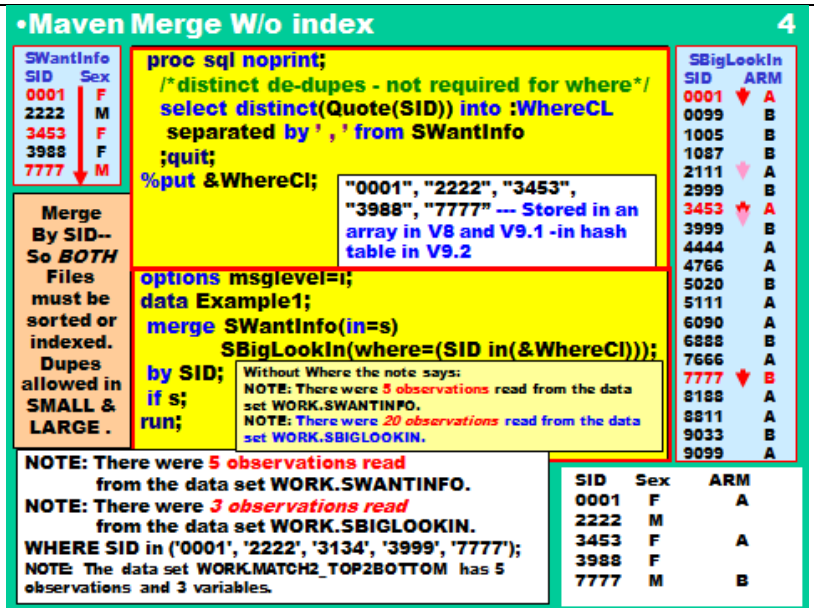
The image of the PPT slide, that is to the right, is typical of the images of slides to follow.

It shows both of the files being "merged", the code, the output (white box in lower right hand side) and the value stored in the macro variable WhereCL.

Hopefully these pictures will make the code easier to interpret.

SMALL (here called SWantinfo) will be the table on the left and LARGE (here called SBigLookIn) will be the table on the right.

This example takes advantage of the data engine (a subroutine that "sits" close to the hard drive) to remove observations that do not pass the where filter.



We use SQL (yellow) to create a macro variable of all the values of the "By Variable" in SMALL and then apply that list in a where clause when LARGE is read.

The data engine can do simple filtering as well as the managing of an index.

The observations that do not meet the where condition are filtered out early in the process of reading the data – the observations are filtered out, by the data engine, before the observations reach the PDV.

With the where clause filter in use (see code with green background) we see the top \note in the log saying three observations are read from LARGE..

If we commented out the green line and uncommented the gray line we would see the second note in the log: It says that twenty observations were read from the hard drive.

However, when the log says three observation were read it means that three observations were read by the data engine, and then flowed up through the layers of SAS and *finally reached the PDV*.

```
/*************************************************************
Section:__ Example 1 Maven merge w/o index
We are merging by subject_id so both files must be sorted
*************************************************************/
OPTIONS NOCENTER msglevel=i;
proc sql noprint;
/*the distinct de-dupes the where clause-
            duplicates not required for the where*/
   select distinct(quote(SID)) into :WhereCL separated by ', '
      from SWantInfo
   ;
quit;
%put &WhereCl;

Proc SQL;/*Be sure that there is no index on LARGE*/
  Drop index SID on BigLookIn;
quit;

options msglevel=i mprint symbolgen mlogic;
data Example1;
/*no index created on large yet, so top to bottom processing of
large, with where filtering in data engine*/
/*Use where to only read, from LARGE, the obs in SMALL*/
   merge  SWantInfo (in=s)
          SBigLookIn (where=(SID in(&WhereCl)));
/*      SBigLookIn ;  */
   by SID;
   if s;  run;
```

```
NOTE: There were 5 observations read
            from the data set WORK.SWANTINFO.
NOTE: There were 3 observations read from the data set
WORK.SBIGLOOKIN.
WHERE SID in ('0001', '2222', '3134', '3999', '7777');
NOTE: The data set WORK.MATCH2_TOP2BOTTOM has 5
observations and 3 variables.
```

```
NOTE: There were 5 observations read from the data set
WORK.SWANTINFO.
NOTE: There were 20 observations read from the data set
WORK.SBIGLOOKIN
```

With the where clause filter in the code. the hard drive *did* read SBIGLOOKIN, from top to bottom and saw every subject_id, so that the data engine could see the values of subject_ID and do filtering. The hard drive processed all the "hard drive pages" for 20 observations. It only passed three observations onto SAS and the PDV.

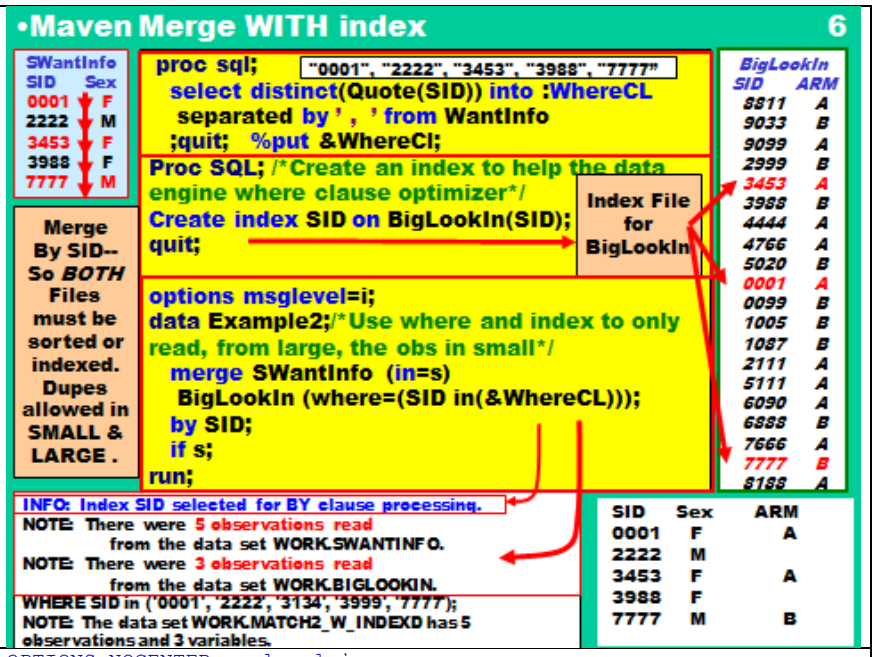
EXAMPLE 2: MavenMergeWIndex

In this example we add code to create a SAS index on LARGE and only SMALL must be sorted. It must be sorted because of the merge by subject_ID (SID).

The code, to right, is the same as the code in the previous example, but, since an index exists, the data engine will, if it determines that it makes sense to do so, make use of the existing index.

Documentation says that when a Where clause in a PROC or a SQL join "pulls" lt 15% of the LARGE file an index will be used.

There is a where clause optimizer that is invoked with this code, but I have not found documentation on the rules it applies.



In this example the hard drive does not read the big file from top to bottom, but , using the index, only reads the data pages that contain the sought for subject_ids. Please note the Blue comment in the PPT graphic ("INFO index SID selected for BY Clause processing").

This automatic use of an existing index (by any where clause in SQL, PROCs or a Data Step) is a behind the scenes feature of SAS that reduces programming effort and clock time.

In this example, for each observation, the data engine uses the index to find the page on the hard drive that contains the desired observations and only those pages are read.

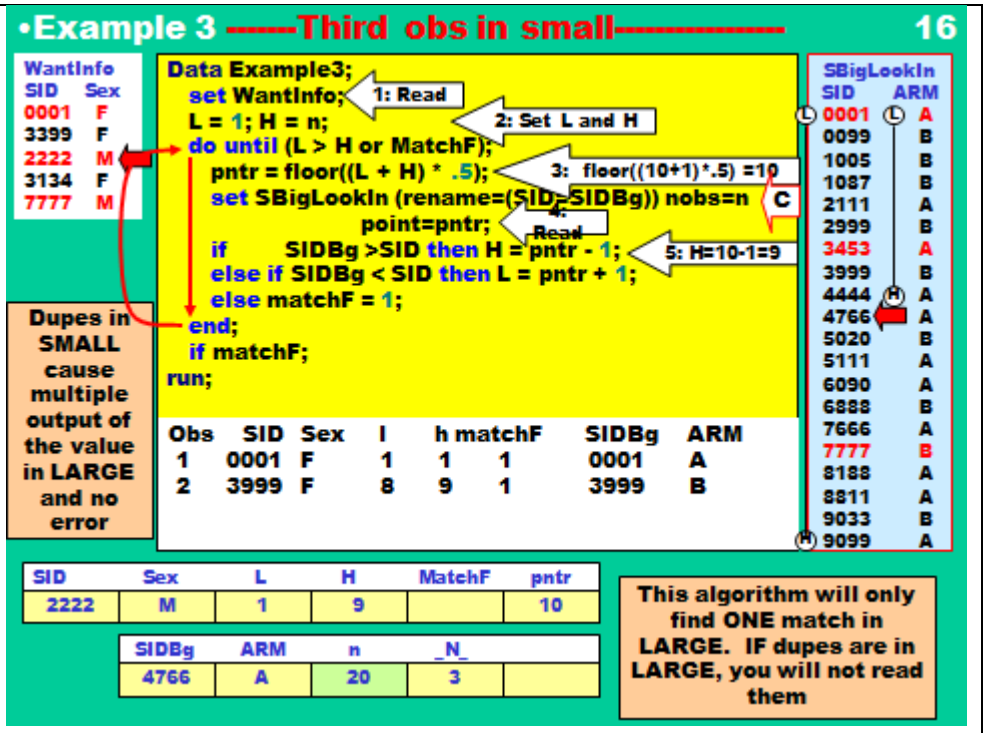
```
OPTIONS NOCENTER msglevel=i;
proc sql noprint;
/*distinct de-dupes the where-duplicates not needed in the where*/
   select distinct(quote(SID)) into :WhereCL separated by ', '
      from SWantInfo
   ;
quit;
%put &WhereCl;

/*****************************************************************
Section:__ Example 2 Maven merge WITH index on big file
*****************************************************************/
Proc SQL;
/*Create an index to help the data engine where clause optimizer*/
Create index SID on BigLookIn(SID);
quit;

options msglevel=i;
data Example2;
/*Use where and index to only read, from large, the obs in small*/
   merge SWantInfo  (in=s)
         BigLookIn (where=(SID in(&WhereCL)));
   by SID;
   if s;
run;

options nocenter;
Proc Print data=ExaMPLE2;
run;
```

EXAMPLE 3: Binary_SmallNotSortedLargeSortedNoDupes

This example introduces: binary searches, the concept of a "search range" and how a search range is "managed" through variables in the PDV.

The search range is the part of the LARGE table that remains to be searched and is defined by the observation/row numbers being between the variables L and H on the PDV.

We use the Nobs=n data set option to load the number of obs in the table into a variable n (and then into H). As we repeatedly execute the loop, we change the values of L and H to narrow the search range.



Please see the conditions in the green section block comment to the right.

In this example, SMALL is read from top to bottom using the Data step loop inherent in the SAS set statement in a Data step.

For each observation in SMALL, a binary search is done on the complete large table.

For each new observation from the SMALL, the initial search range for the large table is re-set to be from the first row to the last row (see code with yellow background).

```
/*****************************************************************
Section:3 Example 3  Binary search of large file:
        Small file: Sorted or Not sorted and duplicates allowed
                large file: Sorted - not indexed - no duplicates
*****************************************************************/
*** Binary search ;
proc print data=wantinfo;
run;

data Example3 ;
   set WantInfo;
   l = 1; h = n;
   do until (l > h or matchF);
      pntr = floor((l + h) * .5);
      set SBigLookIn (rename=(SID=SIDBg)) nobs=n point=pntr;
      if      SID < SIDBg then h = pntr - 1;
      else if SID > SIDBg then l = pntr + 1;
      else matchF = 1;
   end;
   if matchF;
run;

Proc Print data=example3;
Run;
```

Because, for each observation in SMALL, the search range in LARGE is re-set to be from first-row to last row there is no need to sort SMALL and duplicates are allowed in SMALL. However, when one "match" is found the looping and searching stops, so duplicates are not allowed in LARGE.

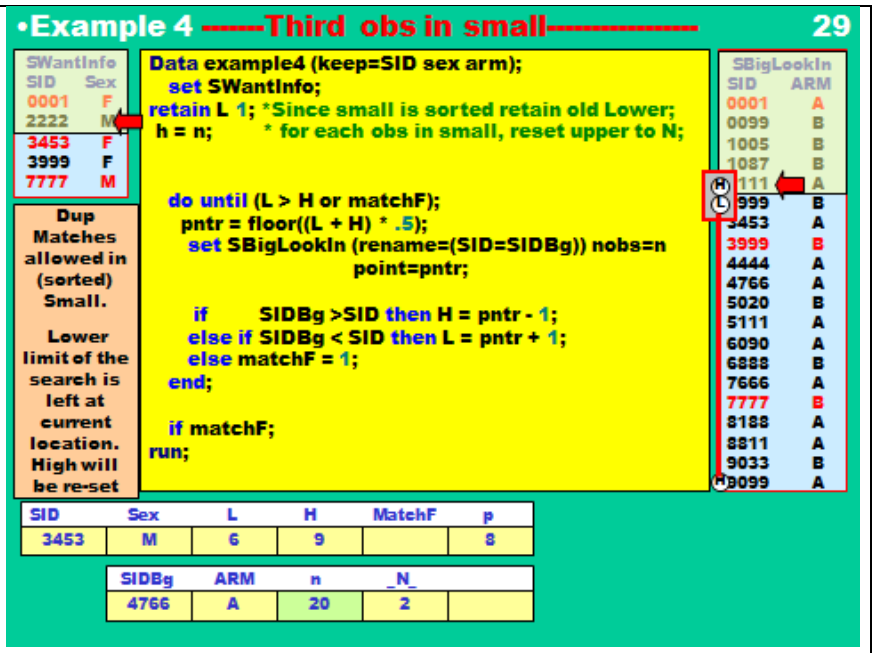
EXAMPLE 4: Binary_SmallSortedLargeSortedNoDupes

In this PPT slide, the search range (rows between L&H) is shown as it is being updated (as it is narrowed in scope) on SBigLookin (A.K.A. LARGE).

The complication in this example is SMALL is sorted and can contain duplicate values. Duplicates are NOT allowed in LARGE.

Once a match is found, the lower limit of the search range is left in its current location and, if the next obs. in the small file is a duplicate of the previous, the new obs. will find its match at the lower end of the search range.

Only the upper limit of the search range is re-set for a new observation from small (see code with yellow background).



Please see the conditions in the green section block comment to the right.

In the graphic above, a section of LARGE is masked.

Since SMALL is sorted and previous “search loops” have not found any matches in these rows, no future search loops need check this part of LARGE.

This is the reason why the code retains L (low end of search range) but resets H (high end of search range) for every new observation in SMALL. H is reset to be n, the highest observation number in LARGE

By not re-setting, for each obs. In SMALL, search time can be reduced.

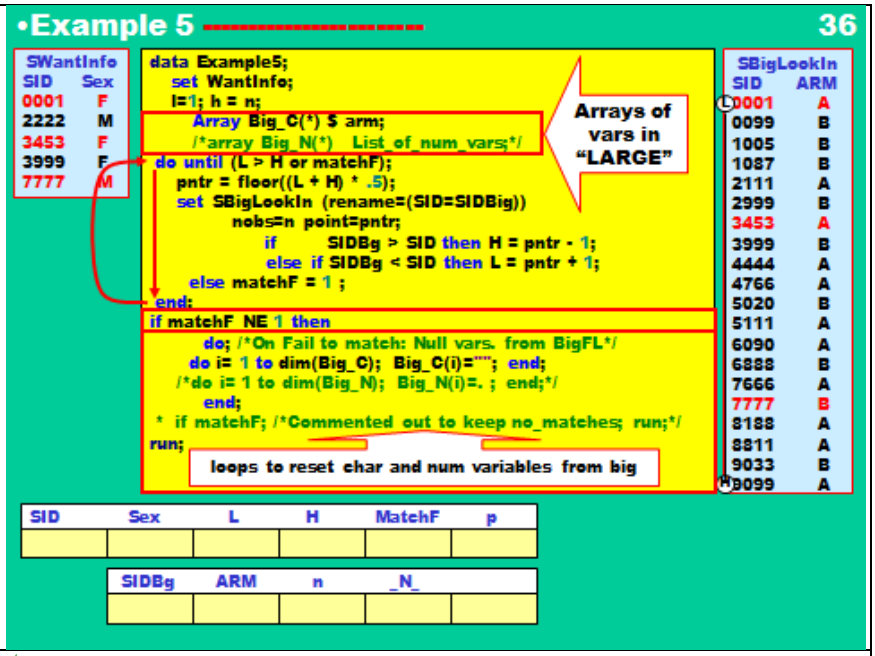
```
/*************************************************************
Section:4 Binary search of large file:
        Small file: Sorted - duplicates allowed
                 large file: Sorted - not indexed - no duplicates
*************************************************************/
Data example4 ;
   set SWantInfo;
  *Since small is sorted retain old Lower;
   retain L 1;     * for each obs in small, reset upper to N;
   h = n;

   do until (L > H or matchF);
     pntr = floor((L + H) * .5);
      set SBigLookIn (rename=(SID=SIDBg)) nobs=n point=pntr;

      if          SID < SIDBg then H =pntr - 1;
      else if SID > SIDBg then L =pntr + 1;
      else matchF = 1;
   end;

   if matchF;
run;

PROC PRINT DATA=EXAMPLE4;
RUN;
```

EXAMPLE 5: Binary_DoNot DeleteMismatches

In this example, imagine we changed the code in response to a client specification that all obs. in SWantinfo were to be included in the table returned by the programmer. This common client specification can be implemented via a left join in SQL.

It is often the case that getting all subject_ids back, with any matching data from LARGE, makes the client more comfortable than just getting back a table of matches.

We reset the search range to be L(low end of range) =1 and H (high end of range)=n for each obs in SMALL and we do not perform any by merging, so SMALL does not need to be sorted.



Please see the conditions in the green section block comment to the right.

We keep all the observations from SMALL by commenting out the statement: `if matchF;`

The complication here is that there is an automatic retaining of observations read using a set statement. If there is a "no-match" the values read from (the last successful reading of) LARGE must be set to missing before an observation can be outputted.

A simple way to do this is to create two arrays, one character and one numeric, where the arrays are made up of the variables in LARGE. Then use a loop to set the variables to missing after a "no match". See code with yellow background.

```
/*************************************************************
Section:_5_ Example5 - Third Binary search
   Simple code so that you do not have to delete "Small"
observations that do not match
*************************************************************/
data Example5;
    set WantInfo;
    l=1; h = n;
/*If we reset L to 1 each time, we do not need to sort Small*/
/*if No-Match: we will loop through arrays/pdv- to make variables
missing */
  Array Big_C(*) $ arm;  /*list of char vars in Large file*/
  /*array Big_N(*)*/     /*List_of_num_vars in Large file;*/
  do until (L > H or matchF);
     pntr = floor((L + H) * .5);
     /*Big must be sorted for binary search to work*/
     set SBigLookIn (rename=(SID=SIDBig)) nobs=n point=pntr;
       if      SID < SIDBig then H = pntr - 1;
       else if SID > SIDBig then L = pntr + 1;
       else matchF = 1 ;
 end;
if matchF NE 1 then
   do;
   /*On Fail to match: Clean PDV-Null NLL vars. from BigFL*/
       do i= 1 to dim(Big_C);  Big_C(i)="";  end;
   /*do i= 1 to dim(Big_N);*/  Big_N(i)=. ;  end;
   /*uncomment arrays as needed*/
  end;
 *if matchF; /*Commented out to keep no_matches; run;*/
run;

Proc Print data=example5;
Run;
```

In the example, the variables were typed into the array statement but this could be done automatically via a PROC contents and a short macro program to create lists of character and numeric variables in LARGE (excluding any By variables) that must be set to missing.

If there are both numeric and character variables in LARGE that need resetting, two arrays are needed.

EXAMPLE 6: Binary_BigFileSortedWithDuplicates

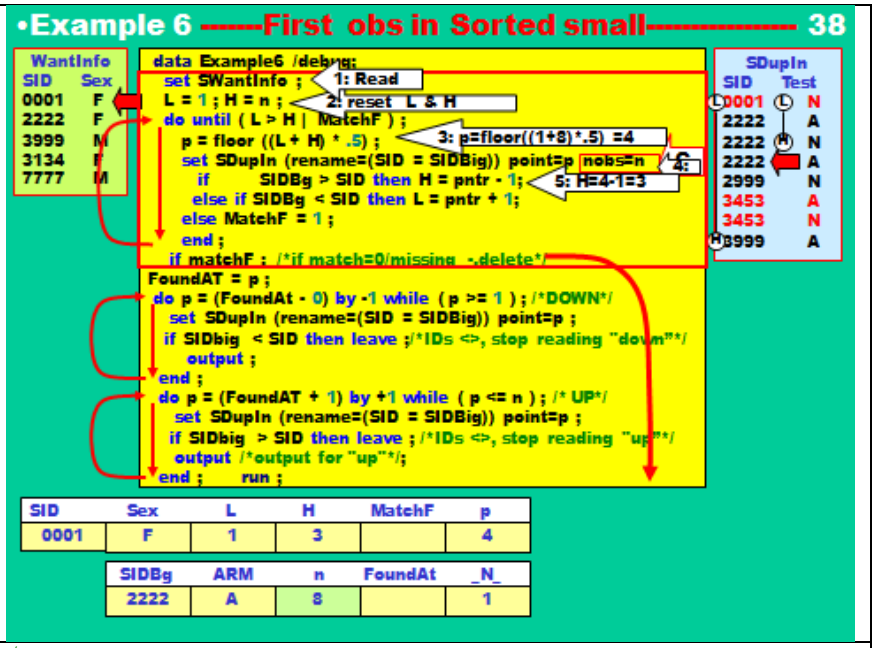
This code allows duplicates in LARGE.

The code with the yellow background (below) loops through a binary search until it finds a "match".

The "found match" may be the row in LARGE with the lowest/ highest/ or some middle observation number.

Since, for each obs. in SMALL, the low and high are reset to be the lowest and highest obs. number in LARGE, the small table does not need to be sorted.

If no match is found the observation is deleted by the statement : *if matchF ;*



Please see the conditions in the section block comment to the right.

If a match is found, the position of the match is stored in FoundAt. Please note that no output has performed at the end of the yellow section.

The gray section starts and re-reads the "match" observation found in the first loop. At this time, the obs. is sent to the output file. The gray loop moves the pointer "upwards" in LARGE, until it finds a new Subject_ID, and then exits.

Then control shifts to the blue-backed loop, which starts at FoundAT +1.

This code loops "down" towards larger observation numbers until it finds a different Subject_ID.

At this event, control escapes from the loop to the run statement.

The logic for duplicates in LARGE, is to dine a match and rem ember that observation number. Then loop upwards and downwards in LARGE to find all other matches.

```
/*****************************************************************
Section:_6_ example 6
below is binary search of a sorted BIG FILE that has duplicates
   - for ID in samll file, can we find match in Large
Small fie not sorted and No Duplicates
 Large file is sorted and has duplicates
*****************************************************************/
data Example6 ;
   set SWantInfo ;
   L = 1 ; H = n ; /*low starts at first obs, high at last obs*/
   do until ( L > H |  MatchF ) ;
      pntr = floor ((L + H) * .5) ;
        /*pointer for the Binary search */
      set SDupIn (rename=(SID = SIDBig)) point=pntr nobs=n ;
      if      Sid < SIDBig then H = pntr - 1 ;
      else if Sid > SIDBig then L = pntr + 1 ;
      else MatchF = 1 ;
      end ;
    if matchF ; /*if match=0/missing -.delete*/
/*IF there are dupes in big- check UP/DOWN from 1st "match"*/
/*Since we will need to search up AND down from where we matched*/
/*Save the current pointer where we found a match in FoundAt*/
  FoundAT = Pntr ;
  do Pntr = (FoundAt - 0) by -1 while ( pntr >= 1 ) ;
    /*Read DOWN big file*/
    set SDupIn (rename=(SID = SIDBig)) point=pntr ;
   /*read "prev"*/
    if SIDbig  < SID then leave ;
   /*IDs <>, stop reading "downâ€*/
       output ;
  end ;
  do Pntr = (FoundAT + 1) by +1 while ( pntr <= n ) ;
     /*read UP big file*/
     set SDupIn (rename=(SID = SIDBig)) point=pntr ;
    /*read "next"*/
     if SIDbig  > SID then leave ;
    /*IDs <>, stop reading "upâ€*/
     output /*output for "up"*/;
  end ;
   run ;
```

EXAMPLE 7A: Compressed_Index_Basic

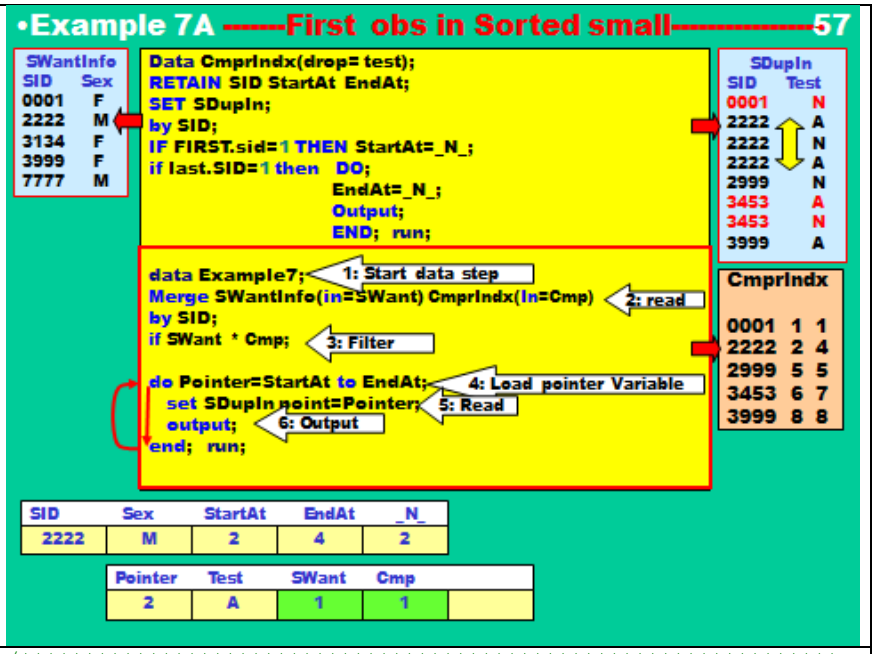
This is the first view of a compressed index.

A compressed index holds three variables:
1) the key variable (here SID or Subject_ID)

2) the row number of the first observation in LARGE with that subject_ID

3) the row number of the last observation in LARGE with that subject_ID

The italic section (see below) builds the compressed index – a list of subject_ids and the first and last row in LARGE where that id appears.



Please see the conditions in the section block comment to the right.

A data step implements the compressed file algorithm. Note we do not index the compressed file.

Small is merged with the table holding the compressed index so that the algorithm has access to the row numbers, in LARGE, of the first and last rows for subject_ids in SMALL..

The gray-backed statement stops processing an obs. if there is no match on subject_id.

The blue section loops through LARGE and, used the point= option to find the desired rows in LARGE, and output observations.

```
/*************************************************************
Section:_7A_ First use of compressed index Mark Keintz
Small and large must be sorted.  We need to so a merge between
small and the compressed index
*************************************************************/
Data CmprIndx; /*Create the comprtessed index on the large file*/
 RETAIN SID StartAt EndAt;
SET SDupIn;
by SID;
IF FIRST.sid=1 THEN StartAt=_N_;
if last.SID=1 then   DO;
   EndAt=_N_;  Output;  END;  run;

proc SQL;
drop index SID from CmprIndx;
run;

data Example7A;
Merge SWantInfo(in=SWant) CmprIndx(In=Cmp);
by SID;
if SWant * Cmp;
/*SET CmprsdIndx key=SID / unique; */
do Pointer=StartAt to EndAt;
   set SDupIn point=Pointer;
   output;
end;
run;

Proc Print data=Example7A;
RUN;
```

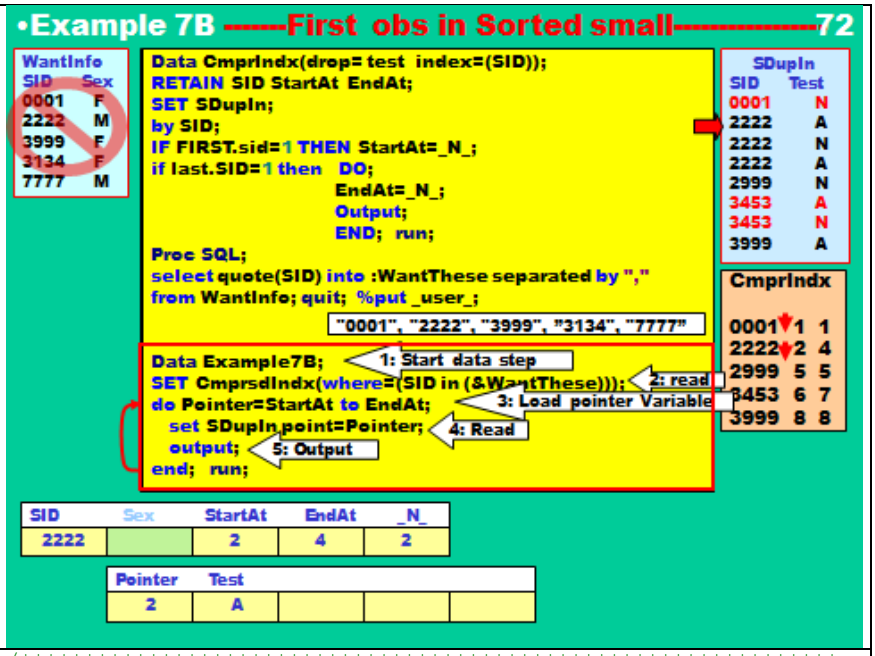
EXAMPLE 7B: CompressedIndex_Index

This example:
1) builds a compressed index table and
2) builds an index on the compressed index table.
3) loads the values from WantInfo (SMALL) into a macro variable

After loading the subject_ids from SMALL into a macro variable the table SMALL is not used again.

The macro variable is used in a where clause on the compressed index (which, for this technique to be useful, is assumed to be large and to have been created by some middle-of-the-night batch job.).

The where clause reduces the number of rows read from the (large) compressed index.



Please see the conditions in the section block comment to the right.

A merge of the table containing the desired subject_Ids with the compressed index can result in a large table – if the compressed index itself is large.

This technique uses a macro driven where clause on the compressed index file (which itself has an index to speed up processing) to eliminate the need for a join with the large compressed index, and hopefully to reduce clock time on the data step.

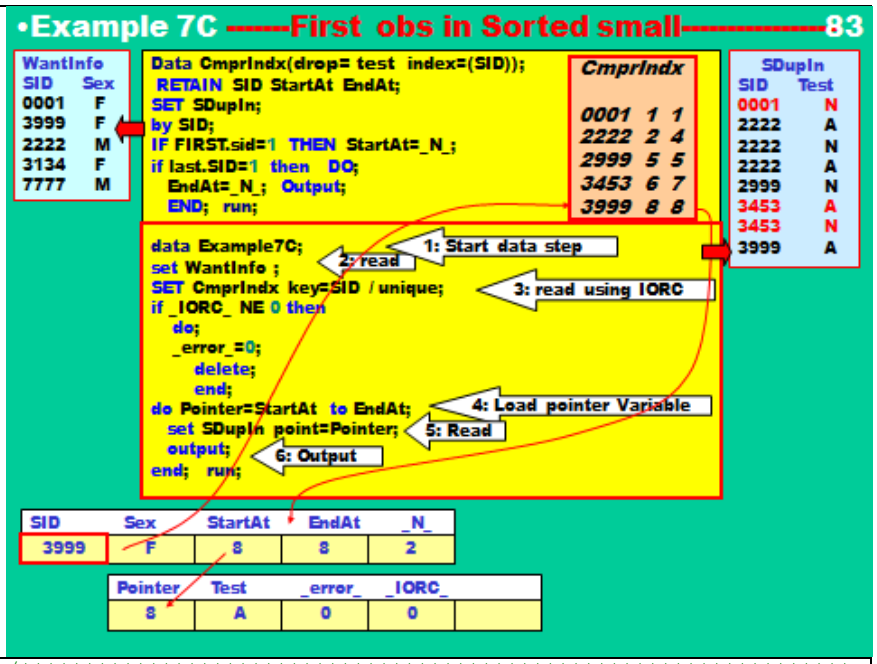
This code with the where in the set statement will “call” the where clause optimizer and let SAS decide if using the index will likely reduce clock time.

```
/*****************************************************************
Section:_7B_ Condensed index V1 - Mark Kentz - WORDS UPenn s
LARGE has an index and is sorted
*****************************************************************/
Options nocenter;
   /*DO NOT Build index on all of big file*/
Data CmprsdIndx(index=(SID));
   /*create an indexed "compressed index" file*/
RETAIN SID StartAt EndAt;
SET SDupIn;
by SID;
IF FIRST.sid=1 THEN StartAt=_N_;
if last.SID=1 then
               DO;
               EndAt=_N_;
               Output;
               END;
run;

/*We do not want to read ALL of the compressed index, only rows in
WantINfo*/
Proc SQL;
select quote(SID) into :WantThese separated by "," from WantInfo;
/*Options are good*/
quit;
%put _user_;

options mlogic mprint symbolgen fullstimer msglevel=i;
data Example7B /DEBUG;
SET CmprsdIndx(where=(SID in (&WantThese)));
/*let SAS decide if it should use the index*/
/*read the file that tells us what obs we want - likely to be
small*/
do Pointer=StartAt to EndAt;
   set SDupIn point=Pointer;
   output;
end;
run;

Proc Print data=Example7B;
run;
```

EXAMPLE 7C: Compressed_Index__IORC_into_The_Compressed_Index

This final example builds a compressed index with a SAS index on it.

It then uses an IORC lookup to get the proper lowest row and highest row from the compressed index - which is assumed to be large in order to have this technique have a possible advantage over other techniques.



Please see the conditions in the section block comment to the right.

This technique could be used where the compressed index is large. The technique builds an compressed index (the section with yellow background) and a SAS index on the compressed index (red italic letters in the yellow section).

The gray section reads SMALL and uses an IORC lookup to get the starting and ending rows from the compressed index.

This is invoking an IORC merge and forces the use of the index. For a paper explaning an IORC merge see references.

```
/*************************************************************
Section __: 7C
Compressed index and IORC lookup into the compressed index
Dupes in LARGE file: LARGE file is sorted ** Index on Compressed
index file
*************************************************************
/
Data CmprIndx(drop= test  index=(SID));
 RETAIN SID StartAt EndAt;
SET SDupIn;
by SID;
IF FIRST.sid=1 THEN StartAt=_N_;
if last.SID=1 then   DO;
   EndAt=_N_;  Output;
   END;
run;

Data Example7C;
set WantInfo ;
SET CmprIndx key=SID / unique;
if _IORC_ NE 0 then
    do;
      _error_=0;
       delete;
    end;
do Pointer=StartAt to EndAt;
   set SDupIn point=Pointer;
   output;
end;    run;
```

The do loop (yellow) uses the information from the starting and ending rows and the point=option to read individual rows in the large file.

CONCLUSION

Thanks to Paul Dormfam and Mark Keintz for putting these tools in the public domain.

Binary searches are, in many situations, the fastest way to find a value in a sorted table and using a binary search can provide a performance improvement. The compressed index has been found, in speed tests, to outperform a SAS index. Your results will depend on the hardware at your site and file sizes.

There are some general principles, or patterns, we should look for as we study these programs.

1) If you reset L (Low end of range) and H (High end of range) to 1 and N for every observation in SMALL
    a. For every obs. in SMALL, you will check ALL of LARGE.
    b. You do not need to sort SMALL.
    c. You can have duplicates in SMALL and get good results.
2) If the algorithm you pick has a "by merge" in it:
    a. You must sort, or index, SMALL and LARGE.
    b. You get a typical SAS merge. Remember that many to many merges are dangerous.
3) If you have duplicates in LARGE, you must add code to look "up" and "down" large, to check for duplicates, after you find the first "Matching observation"
4) Many to many merges are a problem in SAS and if you have duplicates in both SMALL and LARGE, you might have a problem in the program specifications that needs to be thought out.
5) Compressed indexes are cool and speedy and are designed to handle duplicates in LARGE.
6) You can combine these techniques to make things really hard on the next programmer to follow (e.g. putting an index on the compressed index and using a where clause is, I think, a bit tricky).
7) If small is sorted you can use that fact, and some coding so that you do not have to search ALL of large for every observation in SMALL and save time.

The Abbrev file containing the examples can be downloaded from russ-lavery.com. The web site has instructions on how to install it.

For those interested in learning "big file techniques, the abbrev file also has seven other examples of big file techniques which can be studied. The abbrevs have data and code, so these examples will run.

HashLookup Hash_3Way_merge Hashing_LeftJoinLookup HashingInnerJoinLookup

IORC_Lookup CharFormatLookup NumFormatLookup

REFERENCES

Dorfman, Paul, collected SAS-L comments

Keintz, Mark, "A Faster Index for Sorted SAS® Datasets" proceedings of SAS Global Forum 2009
http://support.sas.com/resources/papers/proceedings09/024-2009.pdf

Lavery, Russell. "An Animated Guide: Speed Merges with Key Merging and the _IORC_ Variable"
proceedings of WUSS 2003 conference
http://www.lexjansen.com/wuss/2003/SASSolutions/c-an_animated_guide_speed_merges__iorc_.pdf

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Russ Lavery Contractor Analyst/Programmer
Bryn Mawr, PA 19010:
E-mail: Russ.lavery@verizon.net
Web: Russ-lavery.com