Paper 3276-2019

Simple Techniques to Improve the Performance of your SAS® Programs When Processing Large Data Sets

Steve Cavill, Infoclarity

ABSTRACT

Programs that process large data sets can consume lots of a programmer's time, while waiting for jobs to complete. This paper presents simple techniques that all programmers can use to speed up their programs, leaving time to get more work done (or maybe to play more golf :). Techniques include subsetting, indexes, data set compression and in-memory data.

INTRODUCTION

One of the greatest costs in computer programming is programmer time. This paper presents simple techniques to save programmer time while developing code and running production reports.

The topics are all part of Base SAS®. The content assumes a basic knowledge of SAS data step processing. Topics covered are subsetting, indexes, data set compression, and in-memory data.

MEASURING PERFORMANCE

The standard SAS log option (stimer) does not show a lot of detail. Use fullstimer to get the most detail (output varies by operating system).

```
5     options nofullstimer;
6     data _null_;
7     do i = 1 to 10000000;
8     end;
9     run;
NOTE: DATA statement used (Total process time):
      real time           0.10 seconds
      cpu time            0.11 seconds
```

Windows output:

```
11    options fullstimer;
12    data _null_;
13    do i = 1 to 10000000;
14    end;
15    run;
NOTE: DATA statement used (Total process time):
      real time           0.10 seconds
      user cpu time       0.10 seconds
      system cpu time     0.00 seconds
      Memory                            135k
```

Unix output:

```
73         options fullstimer;
74         data _null_;
75         do i= 5e6 to 1 by -1;output;end;
76         run;

NOTE: DATA statement used (Total process time):
      real time           0.05 seconds
      user cpu time       0.05 seconds
      system cpu time     0.00 seconds
      memory              470.65k
      OS Memory           29088.00k
      Timestamp           03/18/2019 02:39:36 AM
      Step Count                        76  Switch Count  0
      Page Faults                       0
      Page Reclaims                     88
      Page Swaps                        0
      Voluntary Context Switches        0
      Involuntary Context Switches      6
      Block Input Operations            0
      Block Output Operations           0
```

SUBSETTING

During code development and testing you can save a lot of time by processing a subset of large data sets.

SUBSETTING OPTIONS

When reading a file or SAS data set, use options to reduce the amount of data to read. This is particularly useful when testing. There are two options :

- Firstobs= <specifies the row number of the first observation to be read, defaults to 1>
- Obs=   <specifies the last observation number to be read, defaults to last row>

These can be specified as global options, which will affect all following data sets (data step and proc)

e.g.

```
Options firstobs=100 obs=150;
   *Note this will read 51 rows (150-100+1), not 150!
```

Or as data set or infile options:

```
data _null_;
SET xxx (firstobs=1000  obs=1100) ; * read 101 rows
Infile yyy  firstobs=1000  obs=1100 ;
Run;

Proc xxx data=yyy(obs=1000);
```

```
data _null_;
set in (firstobs=1000000 obs=1001000);
run;
NOTE: 1001 observations were read from "WORK.in"
```

```
NOTE: The data step took :
      real time       : 0.218
      user cpu time   : 0.000
      system cpu time : 0.015

data _null_;
set in;
run;
NOTE: 100000000 observations were read from "WORK.in"
NOTE: The data step took :
      real time       : 1:19.687
      user cpu time   : 0:15.312
      system cpu time : 0:03.375
```

Output 1. Log output from obs= and firstobs=options

PROC SURVEYSELECT

When testing reports, using firstobs and obs can be problematic because the subset may not be representative of the whole data set. For example, if the data contains 5 years of data, but is sorted in date order, then a sequential chunk of data may only be from a few months, so a monthly report will not look very complete.

In this case you can write a data step to take a subset, or take advantage of a simple procedure called PROC SURVEYSELECT which takes random samples of data sets. PROC SURVEYSELECT has many, many options for controlling the subset which I don't show here. Note also if you run the proc again, you will get a different sample, so be careful when testing, as different samples will produce different output.

```
data test; /*create a sequential list of numbers */
 do i = 1 to 1000;
   output;
 end;

proc surveyselect data=test out=sample
   sampsize=10;  /* pull out a random sample of 10 */
   run;

Sample looks like:  *note still in sequential order;

 Obs     i

      1     13
      2    116
      3    214
      4    234
      5    305
      6    401
      7    748
      8    753
      9    818
     10    989
```

Output 2. Example PROC SURVEYSELECT output.

INDEXES

Indexes can greatly improve run times of programs that process large data sets.

An index is a supplementary file that provides direct access to required portions of a data set. It is analogous to the way an index works in a textbook. SAS data sets are stored on disk in pages. The index contains the page number(s) where you can find a particular observation, based on a key variable or variables.

Example of a data set with an index based on age

Data Set				Index	
Page	Obs	Name	Age	Age	Page(obs)
1	1	Bill	49	9	3(7)
	2	Mary	22	12	2(6)
	3	Bob	49	17	2(4),3(9)
				22	1(2)
2	4	Ian	17	45	2(5)
	5	Jane	45	49	1(1,3)
	6	Mia	12	77	3(8)
3	7	Alex	9		
	8	Diana	77		
	9	John	17		

Table 1. Index example

HOW INDEXES MAKE DATA ACCESS FASTER

In the example above, accessing the data based on the index can make the access faster. Note in a real example you would have thousands of pages, not just three. Using an index with a data set with only three pages would not achieve much!

The following situations might use the index. I say might, because SAS uses an algorithm to decide if using the index will make things faster. More on this later.

- Accessing a subset of observations

  ```
  … where age=45…
  ```

- Access data in sorted order without pre-sorting (only for ascending order)

  ```
  … by Age;
  ```

- When doing direct access in a data step

  ```
  Set … key=age…;
  ```

- When doing an sql join on that column

  ```
  … a left join b on a.i=b.i
  ```

How the index is used:

SAS reads the index first. It uses a “binary search” to access the data quickly. This is similar to how you would use an index in a book. In simple terms this means start in the middle of the index file, then choose the appropriate half of the file. Then repeat that process by start in the middle of that half, then keep repeating till you find the appropriate index entry.

Illustration of binary search using the sample data above (Table 1)

If the where clause is “where age=45”, the index search will start in the middle (age =22), then check the middle of the bottom half (age=45). So, we only did 2 reads to find the right place. This is faster than reading the index sequentially. Again, this effect would only be noticeable in a much larger file.

Composite Index

An index can be based on more than one variable. SAS logically concatenates the variables and creates the index on the concatenated value. E.g. Surname,Firstname.

CREATING AND USING AN INDEX

Syntax:

Proc datasets:

- Modify <table_name>;
- Index create <one_var>;
- Index create <composite_name> (<var1> <var2> <…>…);
- Index delete <index_name>;

Proc SQL;

- Create index <one_var> on <table_name>;
- Create index <composite_name> on <table_name>(<var1> ,<var2>, <…>,…);
- Drop index <index_name> from <table_name>;

Example:

```
data test;
do i = 1 to 10000;
  x=int(ranuni(0)*10000);
  y=100000-x;
  output;
 end;
 run;

Proc datasets lib=work;
 modify test;
 index create x;
 index create compexample=(x y);
 index delete x;
 quit;

Proc sql;
```

```
    drop index compexample from test;
    create index y on work.test ;
    create index everything on test(y,x);
  quit;
```

SAS uses an algorithm to decide if using the index will make things faster. The algorithm chooses the "best" index or no index. You can use options msglevel=i; to see the index choice in the log. You can override the SAS choice if the algorithm seems to be choosing an inappropriate index. Use this option with caution!

```
305        options msglevel=i;
306        data _null_;
307        set test; where y>9876;
308        run;
INFO: Index y selected for WHERE clause optimization
NOTE: 132 observations were read from "WORK.test"
NOTE: The data step took :
      real time : 0.015
      cpu time  : 0.015

310        data _null_;
311        set test; where y>1;
       run;
<< no note >>
NOTE: 10000 observations were read from "WORK.test"
NOTE: The data step took :
      real time : 0.000
      cpu time  : 0.000
```

Output 3. Msglevel=I

You can override the SAS choice if the algorithm seems to be choosing an inappropriate index. Use this option with caution! SAS usually chooses the right index algorithm. The "Best" index can change over time, so if you override the index choice you need to monitor carefully.

Syntax:

- Data set options:
  - IDXwhere= [YES|NO]
  - IDXname= <index_name>

Examples:

```
390        data _null_;
391        set test(idxwhere=NO); where y>9876;
       run;
NOTE: The IDXNAME=NO option is in force so no index will be used
NOTE: 126 observations were read from "WORK.test"
393
394        data _null_;
395        set test(idxname=everything); where y>9876;
396        run;
NOTE: Index everything selected for WHERE clause optimization
NOTE: 126 observations were read from "WORK.test"
```

```
398       data _null_;
399       set test(idxwhere=yes); where y>1;
400       run;
NOTE: The IDXWHERE=YES option has caused an index to be selected
NOTE: Index y selected for WHERE clause optimization
NOTE: 10000 observations were read from "WORK.test‟
```

Output 4. Overriding SAS index choice

WHEN INDEXES MIGHT MAKE THINGS WORSE

SAS keeps 20 "centiles" to estimate the data distribution in the data set. If the data is unevenly distributed, or the centiles become out of date due to data updates, this can cause the SAS index algorithm to choose the wrong index. You can see in the example below, that using an index was <u>slower</u> than not using an index.

```
374       data _null_;
375       set test; where x=101;
376       run;
NOTE: Index x selected for WHERE clause optimization
NOTE: 175012 observations were read from "WORK.test"
NOTE: The data step took :
      real time : 1:15.578
      cpu time  : 0:02.109
378       data _null_;
379       set test(idxwhere=no); where x=101;
380       run;
NOTE: The IDXNAME=NO option is in force so no index will be used
NOTE: 175012 observations were read from "WORK.test"
NOTE: The data step took :
      real time : 1:03.078
      cpu time  : 0:02.593
```

Output 5. SAS might choose an inappropriate index

Using an index to avoid a sort may make a job slower overall, particularly if the data set is accessed in the same sorted order multiple times. This is because the same page may be loaded repeatedly. Looking at Table 1 again, there are 3 pages in the data set. Sorting and reading sequentially would read 3 pages. Reading using the index could read the same page multiple times – in this order 3,2,1,2,1,3. So the same page is read more than once. Note that this is not a real example, and buffering of both data and index pages means that this is only a problem in very large randomly distributed data sets.

Example: using index

```
53        data _null_;
54         set test;
55         by x;
56        run;
NOTE: 10000000 observations were read from "WORK.test"
NOTE: The data step took :
      real time       : 10:42.890
```

Example: using sort

```
59          proc sort data=test force;
60          by x;
NOTE: Procedure sort step took :
      real time         : 1:55.062
61          data _null_;
62           set test;
63           by x;
64          run;
NOTE: 10000000 observations were read from "WORK.test"
NOTE: The data step took :
      real time         : 6.890
```

In the example above, sorting was much faster than using an index to retrieve the data in sorted order. Sorting a very large data set requires a lot of disk space, so it comes to testing in your environment if sorting or using an index is better.

COMPRESSION

Compression can make reading large data sets smaller by physically reducing the size of the data file, thus reducing the amount of I/O to read the whole file. You have probably used compressed files outside of SAS e.g. Zip compressed files.

Compressing and decompressing files takes extra CPU time but usually saves elapsed time because I/O is typically much slower than CPU resources. Compression works by removing repeated blanks and repeated numeric values (e.g. zeroes). Compression can be set either as a global option or on individual data sets.

Syntax:

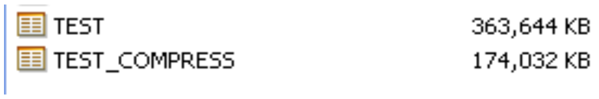
```
Options compress=yes|char|binary;
Data test(compress=yes|char|binary);
```

Char and binary are two different compression algorithms. Yes is the same as char. Char is best for data sets that are mostly character or mixed types. Binary works well for heavily numeric data sets. Options compress=yes is an acceptable default for most SAS jobs.

Compression example:

```
79          data test test_compress(compress=char) ;
80          retain num1-num20 999;
81          retain char1-char4 'blah blah' char5-char20 '         ';
82          do i = 1 to 1e6;
83            output;
84          end;
85          run;
NOTE: Data set "WORK.test" has 1000000 observation(s) and 41 variable(s)
NOTE: Data set "WORK.test_compress" has 1000000 observation(s) and 41
variable(s)
NOTE: Specifying compression for data set "WORK.test_compress" has
decreased its size from 90911
      to 43508 pages (a 53% reduction)
```

Comparing the physical file size on disk:



TEST	363,644 KB
TEST_COMPRESS	174,032 KB

You can see below that reading the compressed version of the file is significantly faster.

```
data _null_;
set test;
NOTE: 1000000 observations were read from "WORK.test"
NOTE: The data step took :
      real time : 3:42.437
      cpu time  : 0:02.078
data _null_;
set test_compress;
run;
NOTE: 1000000 observations were read from "WORK.test_compress"
NOTE: The data step took :
      real time : 1:25.562
      cpu time  : 0:02.093
```

Check the SAS log carefully when using compression as it's possible to make data sets bigger! This is because the compression adds an overhead to the beginning of each observation. If the data set is "skinny" i.e. has very few variables, this overhead can be more than the space saving of compression.

```
data test_compchar(compress=yes);
do i = 1 to 1e5;
  x=int(ranuni(0)*1e7);
  output;
end;
run;
NOTE: Data set "WORK.test_compchar" has 100000 observation(s) and 2
variable(s)
NOTE: Specifying compression for data set "WORK.test_compchar" has
increased its size from 395 to 476 pages (a 21% increase)
```

IN-MEMORY DATA TECHNIQUES

SAS MEMORY OPTIONS

Some SAS global options can affect the amount of memory available to SAS processes. Check the values of these options with

```
Proc options group=memory;
```

- Memsize=
  - Memsize=Max will use all of the available memory, which is OK in a single user environment but be careful in a shared server environment
  - Can only be set at SAS startup

- Sumsize=
  - Used by summary procedures like SUMMARY, MEANS, TABULATE
  - Defaults to memsize
- Sortsize=
  - Memory used by sort procedure, prior to disk utility files.
  - Set to a subset of real memory on your machine, again mindful in a shared server environment.

Note that these options are often locked by your SAS admin, but if they are set to unreasonably small numbers, it's worth changing them.

LOADING SAS DATA SETS INTO MEMORY

If a data set is read repeatedly in the same program, one of the simplest ways to speed up the program is to load the data set into memory. That way it only gets read from disk once. This uses the SASFILE statement.

Syntax:

SASFILE <data set name> open|load|close;

- load – loads the data set immediately
- open – delay loading until the data set is first used
- close – unload and free up the memory

Example:

```
Timestamp           03/20/2019 05:50:18 AM


 75         sasfile sgf.sort load;
 NOTE: The file SGF.SORT.DATA has been loaded into memory by the SASFILE
statement.

Timestamp           03/20/2019 05:51:43 AM


 78         data _null_;
 79         set sgf.sort;
 80         run;

 NOTE: There were 5000000 observations read from the data set SGF.SORT.
 NOTE: DATA statement used (Total process time):
       real time           1.89 seconds
       cpu time            0.76 seconds


 81         sasfile sgf.sort close;
 NOTE: The file SGF.SORT.DATA has been closed by the SASFILE statement.
 82         data _null_;
 83         set sgf.sort;
 84         run;
```

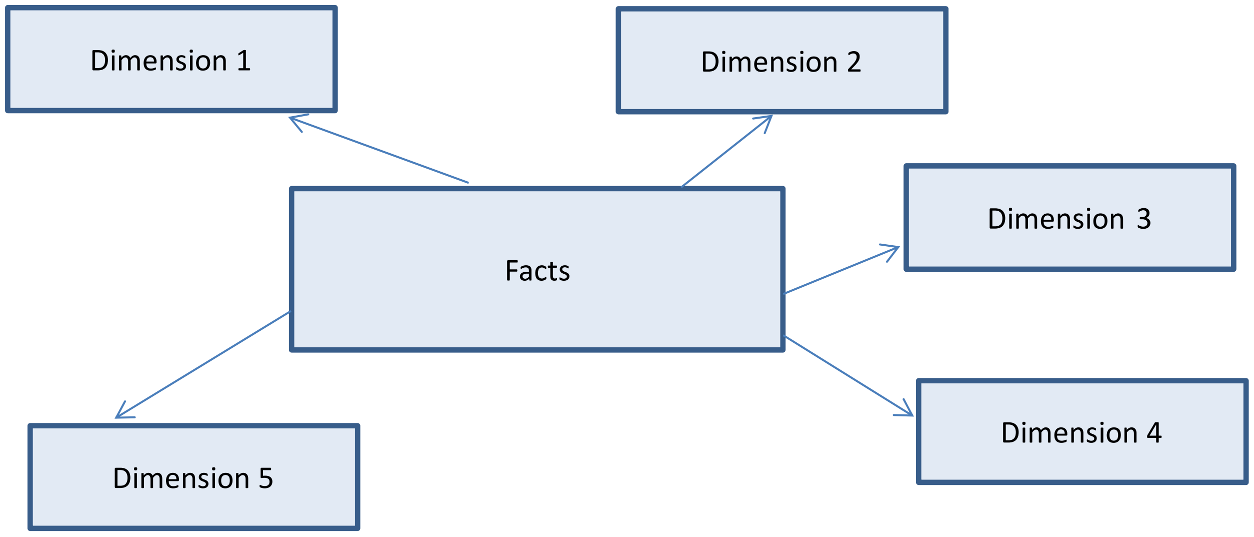
```
NOTE: There were 5000000 observations read from the data set SGF.SORT.
NOTE: DATA statement used (Total process time):
      real time           1:00.60
      cpu time            44.56 seconds


85
86
87
88         OPTIONS NONOTES NOSTIMER NOSOURCE NOSYNTAXCHECK;
100
```

You can see from the above log that reading the SAS data set from memory the first data step took under 2 seconds and reading from disk the second data step took a full minute. The data set in question is about 1GB in size. Note however at the top it took about a minute and a half to load the data set into memory using the SASFILE statement. SASFILE does not write timestamps to the log, I included a data _null_ step to write the timestamp, those statements are edited out of the log above.

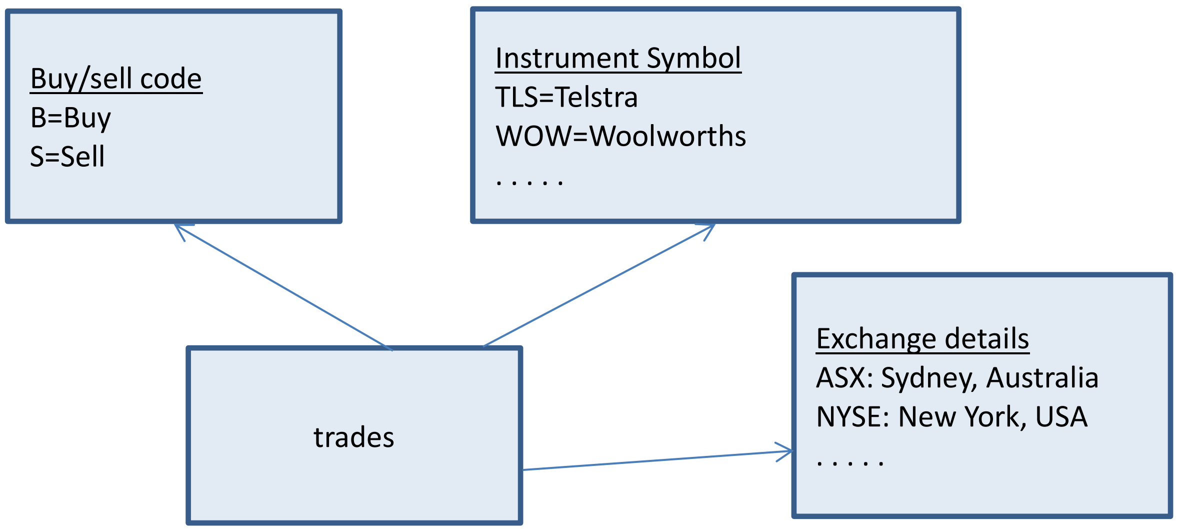
Obviously, this only works if the data set fits into memory. And it saves time only if the large data set is read more than once. Note that most operating systems have a file cache in memory, which serves a similar purpose, but it's not controllable by users and is shared among all open files.

TABLE LOOKUP TECHNIQUES

What is a table lookup? Table lookup typically refers to a data structure where one or more tables containing detail data (the "fact" table) need to lookup "dimension" tables that define some of the facts in more detail. Here's a generic schematic diagram:



EXAMPLE:



In the example above the trades tables is a list of stock trades, across multiple exchanges and stocks. The trades table contains details of the trades, include the codes for the stocks and exchanges. To produce a report, we want to "lookup" the code and return the description. E.g. for the exchange code ASX we want the description Australian Stock Exchange. A typical method to achieve this is a data step merge or SQL join. If there are many lookup tables, data step is complicated requiring multiple sort/merge steps, and SQL requires joining all the tables, which again can get complicated and slow.

I will present two alternatives to data step merge and sql join:

- Formats as lookup tables
- Hash tables

FORMATS AS LOOKUP TABLES

Traditionally, formats are used to enhance output. However, in essence they are ways to recode codes into formatted values, which is what we are trying to achieve with our lookup tables.

There are two steps in the process:

1. Create a format from a data set
2. Use the lookup table in a data step or where clause

This allows us to avoid data step sort/merge and sql join, which can greatly reduce execution time. As formats can be used in analysis procs, formats also allow automatic aggregation of data without any recoding.

The format is loaded into memory when it is required, which is why it is very fast. However, that means the format(s) must fit in available virtual memory. This is not usually a problem for typical lookup tables.

You can create the formats with source code:

```
Proc format;
Value $Buysell
          'B'    ='Buy'
          'S'    ='Sell'
     other       ='Unknown Code'
;
```

This method works for very small tables, but is unwieldy for large tables and also requires source code changes if the lookup tables are updated. A better solution is to load the format directly from the lookup table, using the cntlin= option of the FORMAT procedure.

Syntax:

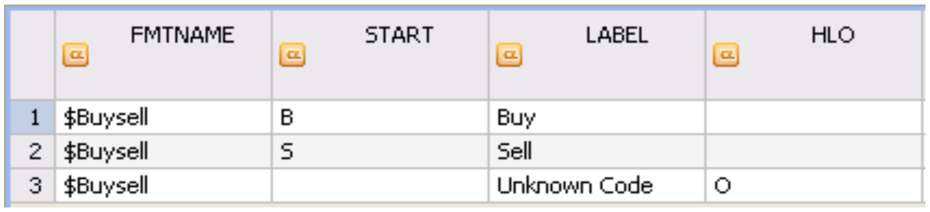
```
Proc format cntlin=xxx;
```

The cntlin data set contains (at least) these three columns

- Start (and end if it is a range)
- Label
- Fmtname
    - Fmtname is the name of the format
    - Standard SAS name rules, but must not end in a digit
    - $ symbol denotes a character format
    - $fmtname -> character,   fmtname -> numeric

There are many more possible columns, mainly for numeric formats. Check the documentation for all the possibilities. In our example we need a column HLO with the value 'O' meaning other.

For our $buysell format, this is the data set we need:



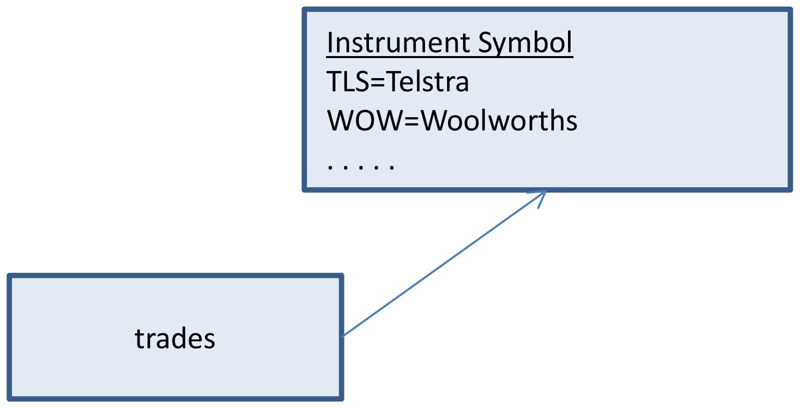
	FMTNAME	START	LABEL	HLO
1	$Buysell	B	Buy	
2	$Buysell	S	Sell	
3	$Buysell		Unknown Code	O

You can use a data step or sql view to create the cntlin data set.

To lookup the format in a DATA STEP or SQL, use the PUT function, e.g.:

```
Put ('B',$buysell.)   ➔ 'Buy'
Code='X'; Put (code,$buysell.)  ➔ 'Unknown Code'
If Put (code,$buysell.)  = 'Sell' then …;
```

Avoiding sort/merge with a format:



Sort/merge method:

```
81         /* merge requires an explicit sort */
82
proc sort data=trades;by symbol;run;
ERROR: An error occurred during sort : Error writing to data set
"WORK.trades" :
       No space left on device
NOTE: Procedure sort step took :
      real time : 39:22.890
      cpu time  : 1:13.546
84         proc sort data=crsdata.prices;by symbol;
85         run;
86
87         data test;
88         merge trades crsdata.prices(keep=Symbol InstrumentName);
89         by symbol;
90         run;
NOTE: 7122000 observations were read from "WORK.trades"
NOTE: 1908 observations were read from "CRSDATA.prices"
NOTE: Data set "WORK.test" has 7122000 observation(s) and 22 variable(s)
NOTE: The data step took :
      real time : 13:05.781
      cpu time  : 0:48.937
```

You can see the sort is very slow – I left the out of space error message to show that sorting requires a lot of space – this is a common error. I then reran the sort (but left those lines out of the log)

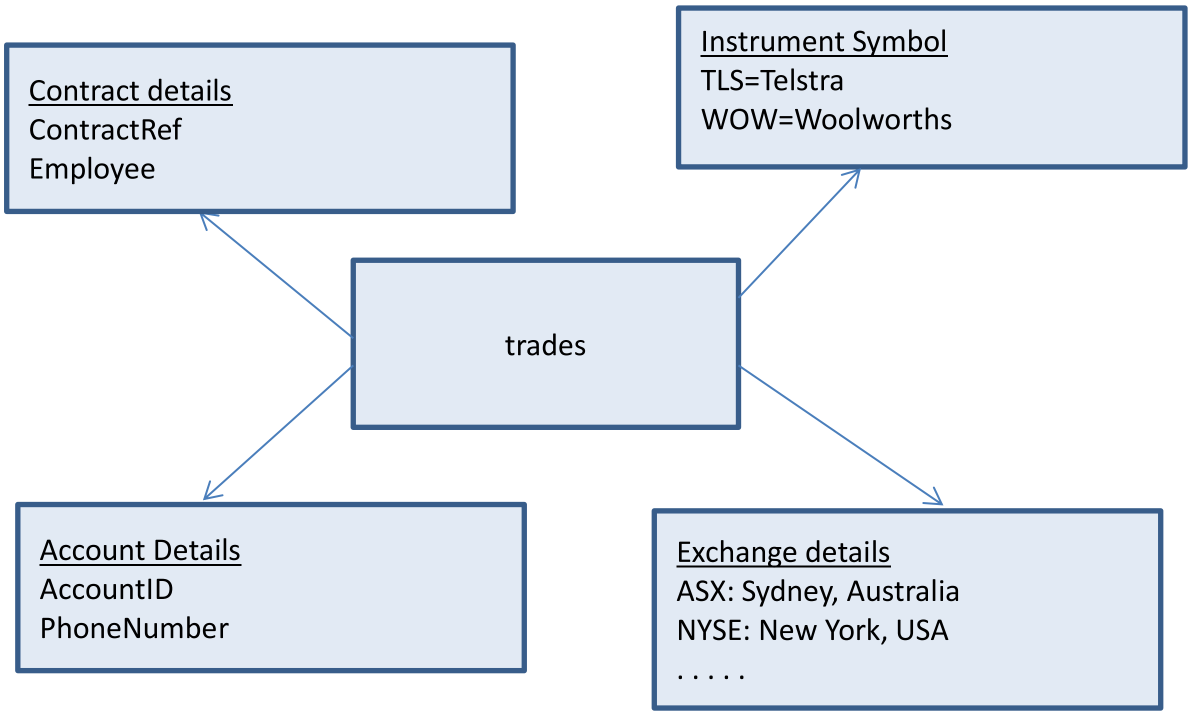
Format method:

```
/* format lookup requires no sorting and a single pass of each table. */
93  data cntlin;   *could use a view here to make more dynamic ;
       retain fmtname '$symbolname';
96         set crsdata.prices(keep=Symbol InstrumentName);
97         rename Symbol=start;
98         rename InstrumentName=Label;
```

```
101 proc format cntlin=cntlin;
       run;
NOTE: 1908 observations were read from "CRSDATA.prices"
NOTE: Data set "WORK.cntlin" has 1908 observation(s) and 3 variable(s)
NOTE: The data step took :
      real time : 0.546
      cpu time  : 0.015
NOTE: Format $symbolname is already in the catalog and will be overwritten
NOTE: 1908 observations were read from "WORK.cntlin"
NOTE: Procedure format step took :
      real time : 4.000
      cpu time  : 0.046

104       data test;
105       set trades;
106       InstrumentName=put(symbol,$symbolname.);
107       run;
NOTE: 7122000 observations were read from "WORK.trades"
NOTE: Data set "WORK.test" has 7122000 observation(s) and 22 variable(s)
NOTE: The data step took :
      real time : 11:02.515
      cpu time  : 0:36.218
```

The format method is slightly faster than the merge but avoids the slow sort completely.

Avoiding slow sql join with a format



SQL join method:

```
proc sql _method;
create table testsql as
Select  distinct
         facts.*
        ,symbols.InstrumentName
        ,con.EmployeeName
        ,ex.Country
        ,ac.TelephoneNumber
  from  crsdata.trades       as facts
       ,crsdata.customers as ac
       ,crsdata.exchanges as ex
       ,crsdata.prices       as symbols
       ,crsdata.Contracts as Con
  where
        facts.accountid=ac.accountid
    and facts.ContractRef=Con.ContractRef
    and facts.symbol=symbols.symbol
    and facts.exchange=ex.exchange
  ;
Sqxsort
    sqxuniq
      sqxhsh
         sqxhsh
           sqxhsh
             sqxhsh
              sqxsrc (CRSDATA.trades)
              sqxsrc (CRSDATA.exchanges)
              sqxsrc (CRSDATA.customers)
              sqxsrc (CRSDATA.Contracts)
            sqxsrc (CRSDATA.prices)

NOTE: Data set "WORK.testsql" has 111863 observation(s) and 24 variable(s)
NOTE: Procedure sql step took :
      real time       : 1:25.015
      user cpu time   : 0:05.156
      system cpu time : 0:11.359
```

The _method option of the SQL procedure shows details in the log of the execution method of sql.

Format method:

```
data testformats;
  set crsdata.trades;
  InstrumentName
    =put(symbol,$symbolname.);
  EmployeeName
    =put(ContractRef,$ConEmployeeName.);
  Country
    =put(exchange,$exchangecountry.);
  TelephoneNumber
    =put(AccountID,$ACTelephoneNumber.);
run;
```

```
NOTE: Data set "WORK.testformats" has 109519 observation(s) and 24
variable(s)
NOTE: The data step took :
      real time       : 11.406
      user cpu time   : 1.796
      system cpu time : 1.156
```

The format method uses significantly less execution time and CPU time. As a personal opinion, I think it's easier to read as well. ☺

Formats in analysis procedures

Although not technically a "lookup", the ability to use lookup tables as formats in procedures can save time by avoiding the recoding altogether

```
/*Use format to create analysis variable*/
data cntlin;
set crsdata.customers
    (keep=AccountID PostCode);
start=AccountID;
label=Postcode;
fmtname='$ACPostCode';
proc format cntlin=cntlin;run;
```

Use DATA step to recode the lookup variable:

```
data trades_postcode;
  set crsdata.trades;
  Postcode=put(AccountID,$ACPostcode.);
run;
proc freq data=trades_postcode;
tables postcode;
run;
```

Compared with simply using the format in the FREQ procedure. This avoids a pass of the data to do the recoding:

```
proc freq data=crsdata.trades
 order=formatted;
tables AccountID;
Format AccountID $ACPostCode.;
run;
```

HASH TABLES

Hash tables are quite similar in concept to using formats as lookup tables. The syntax is somewhat more complicated, but they overcome a significant shortcoming of formats. A format can only have one lookup code and one return value. A hash table can have multiple codes in the lookup (referred to as composite keys) and return multiple values.

The advantages are similar – they are stored in memory so very fast. Unlike a format though they can only be used in a DATA step, not a procedure or SQL.

Hash tables have a wide range of uses out of the scope of this paper. I will not explain the syntax in detail. I encourage you to read some of the many excellent papers on hash tables in the SAS Global Forum online proceedings. This paper looks at hash tables as lookup tables in comparison to formats.

As mentioned, the major advantage of hash tables over formats is that hash tables support multiple keys and multiple return values. For example, our stock trades example has a table of high and low prices for stocks by month. This is not suitable for a format, but is suitable for a hash table

The lookup table looks like this:

Keys(s)		**Data**	
Symbol	**QuoteMonth**	**Opening Price**	**Closing Price**
TLS	Sep 2010	2.50	2.60
TLS	Oct 2010	2.55	2.45
WOW	Sep 2010	26.55	27.00
WOW	Oct 2010	27.50	28.55

Hash table syntax to create the hash table

- Uses *object* and *dot* notation
    - **Declare hash** <objectname> ();
    - <objectname>**.definekey();**
    - <objectname>**.definedata();**
    - <objectname>**.definedone();**

Example

- Using our prices tables, Find the value of a portfolio for a series of dates.

Create the hash table (in a data step)

```
 length Exchange $3
        Symbol $6
        QuoteMonth open close 8
        ;
   declare hash stockprices (dataset:'crsdata.prices');
   stockprices.definekey('Exchange','Symbol','QuoteMonth');
   stockprices.definedata(‘open',‘close');
   stockprices.definedone();
```

Hash table syntax to create the hash table

- <objname>**.find()**
    - use defined key variables

- <objname>.find(key:xxx,key:yyy)
  - Can use different key variables
  - Cannot use different data variables
- Finds the row in the hash that matches the key(s)
- Returns the data values to the PDV in the data vars

```
data portfolios errors;
keep investor symbol holding quotemonth value open close;
length Exchange $3 Symbol $6  <--------------------------------(3)
       QuoteMonth open close 8
       ;
if _n_=1 then do; <--------------------------------------------(1)
  declare hash stockprices (dataset:'crsdata.prices');
  stockprices.definekey('Exchange','Symbol','QuoteMonth');
  stockprices.definedata('open','close');
  stockprices.definedone();
end;
Exchange='ASX'; Year=2007;  /* hardcoded in this example */
set crsdata.investors;
do month=7 to 12;
 QuoteMonth=mdy(month,1,year);
 rc=stockprices.find();<---------------------------------------(2)
if rc=0 then output portfolios;
         else output errors;
end;
Run;
```

Some notes on the code above:

(1) The hash table is created at run time (not execution time), so it is imperative to add this "if _n_ =1" code, so the hash table is only created once. It's not an error to not do so, but would be very inefficient.

(2) We take the symbols in our investors file and lookup the stock prices for that symbol for July to December 2007. It's important to check the return code from the find. A non-zero return code means the lookup code was not found (in our example meaning no prices for the specified exchange, symbol and month). If you DON'T check the return code you can produce incorrect results.

(3) We need to create the hash table variables in the PDV before we try to create the hash table or we will get an error. Using a length statement works, but gets pretty tedious if there's lots of variables, as we need to know the length of all the variables. We can use this code instead of the length statement to create the variables by reading the header of the lookup table, but not actually reading any data:

```
 if _n_=0 then set crsdata.prices;  /* define the vars in the PDV, */
                                    /*  but never actually read the dataset */
```

_n_ is never zero, so that statement compiles (and creates the variables in the PDV) but never executes

Comparison of hash and formats as lookup tables:

	Format	**Hash**
Available in Data Step	Yes	Yes
Available in Proc Step	Yes	No
Size limit	Available memory	Available memory
Key columns/Data Columns	Single/single	Many/many
Can load from SAS data set	Yes	Yes
Duplicate Keys	No	Yes

CONCLUSION

Improving run times for SAS programs that process large data sets can significantly improve programmer productivity. With large data sets, the major contributor to slow run times is disk I/O. Reducing I/O usually will improve run time.

Two methods of reducing I/O presented in this paper are:

- Reducing the actual amount of data that is read, by using data sub setting techniques and indexes.
- Making use of memory to improve run time. Current day computers typically have large memory capacity, so this paper demonstrated techniques to load data into memory because access to data in memory is orders of magnitude faster than accessing data on disk

None of the techniques are a "magic bullet". Users need to individually test techniques to identify which techniques provide the best solution to each individual performance problem.

CONTACT INFORMATION

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

Steve Cavill
Infoclarity
Ph: +61 425 333 233
steve.cavill@infoclarity.com.au