Top 10 Ways to Optimize Your SAS Code

Presented by Jeff Simpson
SAS Customer Loyalty

Writing efficient SAS programs means balancing the constraints of

TIME

Writing efficient SAS programs means balancing Time and

SPACE

Writing efficient SAS programs means balancing Time and Space.

Luckily, the SAS programming language offers a wide assortment of efficiency techniques, intended to help you balance these constraints.

+

Agenda: Use Cases

#1—Minimizing Reads on INPUT

#2—KEEP and WHERE Data Set Options

#3—WHERE clause optimization

#4—Avoiding Multiple Passes of the Data

#5—Indexing Considerations

#6—Sorting Considerations

#7—Avoid Heterogeneous Joins

#8—Data Set Compression

#9 - SAS Dataset as a Table vs. a View

#10—Checkpoint/Restarting SAS Jobs

Processing Environment

- Software
  - SAS Release 9.3 & 9.4
  - Windows 7 Enterprise Edition – 64 bit
- Hardware
  - Intel Q720 @ 1.60GHz
  - 8 cores
  - 16 GB Physical Memory
  - 24 GB Virtual Memory (Page File Size)

SAS Default Settings

Log - (Untitled)

```
1     %macro stats;
2
3        %let megs = 1048576;
4        %let memsize = %eval(%sysfunc(getoption(memsize))/&megs);
5        %let sortsize = %eval(%sysfunc(getoption(sortsize))/&megs);
6        %let cpucount = %sysfunc(getoption(cpucount));
7        %put ************************************************;
8        %put * Key Performance Options are:                  *;
9        %put * SORTSIZE = &Sortsize%str(MB;) MEMSIZE = &memsize%str(MB;) CPUs = &cpucount%str(;
9   ! ) *;
10       %put ************************************************;
11    %mend;
12    %stats;
************************************************
* Key Performance Options are:                  *
* SORTSIZE = 256MB; MEMSIZE = 2048MB; CPUs = 8; *
************************************************
```

Processing Environment

- STIMER - Writes real-time and CPU time system performance statistics to the SAS log

Statistic	Description
Real-Time	the amount of time spent to process the SAS job. Real-time is also referred to as elapsed time.
CPU Time	the total time spent to execute your SAS code and spent to perform system overhead tasks on behalf of the SAS process. This value is the combination of the user CPU and system CPU statistics from FULLSTIMER

Note: Starting in SAS 9, some procedures use multiple threads. On computers with multiple CPUs, the operating system can run more than one thread simultaneously. Consequently, CPU time might exceed real-time in your STIMER output

Processing Environment

- FULLSTIMER - Specifies whether all the performance statistics of your computer system that are available to SAS are written to the SAS log

Statistic	Description
Real-Time	the amount of time spent to process the SAS job. Real-time is also referred to as elapsed time
User CPU Time	the CPU time spent to execute SAS code
System CPU Time	the CPU time spent to perform operating system tasks (system overhead tasks) that support the execution of SAS code
Memory	the amount of memory required to run a step.
OS Memory	the maximum amount of memory that a step requested from the System

Use Case #1: Minimizing Reads on INPUT

- Limit the columns read by using INPUT's @ column pointer and # for line pointer.
- Trailing '@' is useful in conditional reads by evaluating conditions since we don't have to read the entire file into a SAS data set:

```
data this_year;
  infile 'ALL_Sales';
  input @24 year 4. @;
  if year = 2005;
     input ...;
```

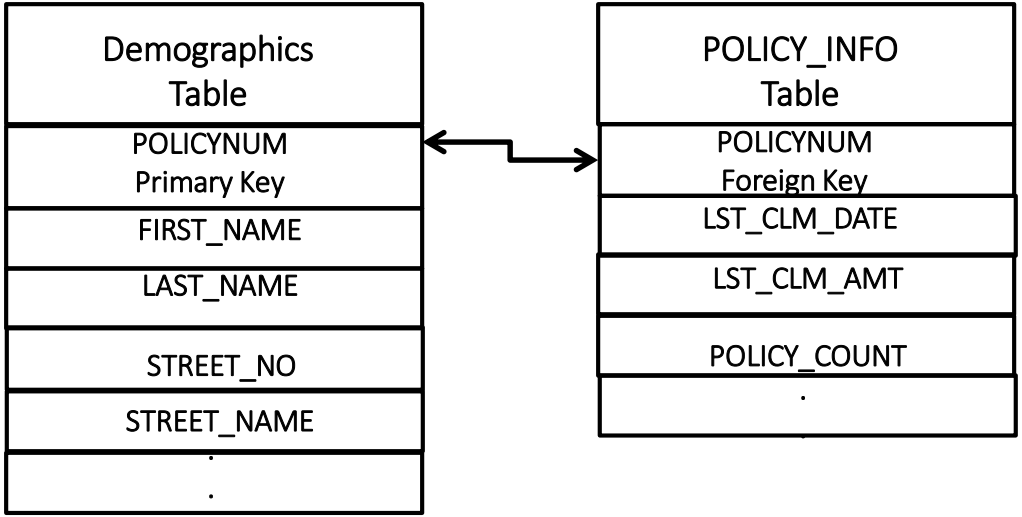
Use Case #1: Minimizing Reads on INPUT

- Limiting the columns
  - Delimited
    - » Use dummy variables of length 1 to reduce size

```
filename csvin 'F:\LargeData\SummaryInfo.csv';

data work.SummaryInfo(keep=date cust_id balance);
  infile csvin dsd;
  input date :YYMMDD10. dummy :$1. cust_id :$15.
    (dummy dummy dummy) (:$1.) balance :16.2;
run;
```

USE Case Data Model



Each Table contains 10,302,000 observations

Use Case #1: Minimizing Storage on INPUT

- **Consider SAS formats** since they behave as ‘look-up’ tables e.g.
  - Rather than storing state names, store their FIPS value and create SAS formats for state names
  - In our case, we would need ~196MB of storage without a SAS format and just ~29MB with a SAS format and 3 bytes for FIPS

Use Case #1: Minimizing Reads on INPUT

```
28   proc format;
29        value statefmt 17 = "Illinois"
30                       37 = "North Carolina";
NOTE: Format STATEFMT has been output.
31   run;

NOTE: PROCEDURE FORMAT used (Total process time):
      real time          0.07 seconds
      cpu time           0.01 seconds

32   data states;
33   attrib state_name length=3 label='Home State of Policy Holder' format=statefmt.;

 Alphabetic List of Variables and Attributes

        #   Variable       Type      Len   Format        Label

        1   state_name     Num         3   STATEFMT.     Home State of Policy Holder
```

Use Case #1: Minimizing Reads on INPUT

```
Data work.statefmt(keep = fmtname start label);
   attrib label length=$32 label="State Name";

/* Get the first value of each state */
   set maps.us;
      by state;
      if first.state;

/* Rename the fields for Proc Format processing in next step
*/
   Fmtname="statefmt";
   Start= State;
   Label=Statecode;

   Run;
```

Use Case #1: Minimizing Reads on INPUT

NOTE: There were 1551 observations read from the data set MAPS.US.
NOTE: The data set WORK.STATEFMT has 52 observations and 3 variables.
NOTE: DATA statement used (Total process time):
real time 0.02 seconds
cpu time 0.01 seconds

/* Create the Format Catalog from work.statefmt */

```
Proc format library = library
   cntlin  = work.statefmt;
```

NOTE: Format STATEFMT has been written to LIBRARY.FORMATS.

Use Case #1: Minimizing Reads on INPUT

```
60  data _null_;
61      fipcode=37;
62      put fipcode statefmt.;
63      run;

North Carolina

NOTE: DATA statement used (Total process time):
      real time          0.00 seconds
      cpu time           0.00 seconds
```

In our case, we would need ~196MB of storage without a SAS format and just ~29MB with a SAS format and 3 bytes for FIPS

Automatic Format Creation

```
proc sql;
    create view statefmt as
    select state as start,
      statecode as label,
      "$statefmt" as fmtname,
      "C" as type
    from maps.us;
quit;


proc format cntlin=statefmt;
run;
```

Use Case #2: KEEP and WHERE Data Set Options

- KEEP and DROP Data Set options are used to limit columns
- WHERE limits rows read from SAS Data Sets

- KEEP statements in the DATA Step apply only to output tables

- Note the arrangement of the WHERE and KEEP data set options

Use Case #2: KEEP and WHERE Data Set Options

- Use WHERE= data set options whenever possible

  (where=(state_loc not in ("CA","NC") or car_use = "Commercial")

  - With SAS/Access Engines, SAS makes an effort to send the clause to the RDBMS for evaluation rather than to SAS

- IF statements force SAS to read all rows and keep only those where the condition is TRUE
- KEEP options list must contain the WHERE filter column names
- Derived columns in the DATA Step are not listed in the KEEP option

Use Case #2: KEEP and WHERE DATA Set Options

```
3     data use_case2;
4        set states.demographics;
5        if upcase(state)="CA" then output;
6     run;

NOTE: There were 10302000 observations read from the data set
STATES.DEMOGRAPHICS.
NOTE: The data set WORK.USE_CASE2 has 313280 observations and 20
variables.

NOTE: DATA statement used (Total process time):
      real time           3.15 seconds
      user cpu time       1.99 seconds
      system cpu time     1.15 seconds
```

Use Case #2: KEEP and WHERE DATA Set Options

```
 9    data use_case2;
10      set states.demographics(where=(upcase(state)=("CA")));
11   run;

NOTE: There were 313280 observations read from the data set
STATES.DEMOGRAPHICS.
            WHERE UPCASE(state)='CA';
NOTE: The data set WORK.USE_CASE2 has 313280 observations and 20
variables.

NOTE: DATA statement used (Total process time):
      real time           3.89 seconds
      user cpu time       2.52 seconds
      system cpu time     1.12 seconds
```

Use Case #2: KEEP and WHERE DATA Set Options

```
8     data use_case2;
9        set states.demographics(where=(upcase(state)=("CA"))
10                       keep=state
11                            density);
12    run;

NOTE: There were 313280 observations read from the data set
STATES.DEMOGRAPHICS.
             WHERE UPCASE(state)='CA';

NOTE: The data set WORK.USE_CASE2 has 313280 observations and
2 variables.
NOTE: DATA statement used (Total process time):
      real time              3.50 seconds
      user cpu time          2.35 seconds
      system cpu time        1.15 seconds
```

Use Case #2: Behavior of WHERE Data Set Option and IF Statements

- WHERE and IF processing are not always ‘interchangeable’
- IF processing must be used with:
  - Accessing raw data using INPUT statements
  - With Automatic Variables, e.g. first.variable, last.variable, _N_, etc.
  - Using newly created variables in the same DATA Step
  - In combination with data set options such as OBS =, POINT = , FIRSTOBS =
  - To conditionally execute a statement

Use Case #2: Behavior of WHERE Data Set Option and IF Statements

- WHERE and IF processing are not always ‘interchangeable’
- WHERE processing must be used to:
  - Utilize special operators such as LIKE or CONTAINS
  - Filter rows for input to SAS Procedures
  - Trigger use of indexes*, if available
  - When sub-setting as data set option
  - When sub-setting using PROC SQL

*The presence of an index column on a SAS data set does not always guarantee its use in a query

Use Case #2: Behavior of WHERE Data Set Option and IF Statements

- WHERE and IF processing are applied differently in MERGE operations:
- With WHERE processing the sub-setting takes place before the MERGE operation.
- With IF processing the sub-setting takes place after the MERGE operation.

Be careful!

Use Case #3: WHERE Clause Optimization

- Avoid the NOT operator if you can use an equivalent form
  - Inefficient: where SALARY not>48000
  - Efficient: where SALARY<=48000
- Avoid LIKE predicates that begin with % or _ .
  - Inefficient: where COUNTRY like '%INA'
  - Efficient: where COUNTRY like 'A%INA'
- Avoid arithmetic expressions in a predicate.
  - Inefficient: where SALARY>12*4000.00
  - Efficient: where SALARY>48000.00

Use Case #4: Avoiding Multiple Passes Through the Data

- Plan ahead by making all calculations/derivations in a single step
- Some PROC options force a second pass through the data. E.g. UNIFORM option for PROC PRINT
- Consider PROC SQL for querying and modifying tables in a single step
- Consider creating a permanent SAS data set in those cases where the data is static and access is frequent

Use Case #4: Avoiding Multiple Passes Through the Data

Use Case #4: Avoiding Multiple Passes Through the Data

```
4    data usecase4;
5          set states.all_customers(where=(upcase(state)="CA" and
6                                                  clm_amt > 0 and
7                                                  clm_freq > 1)
8                                          keep  =  car_use
9                                                   clm_freq
10                                                  state
11                                                  clm_amt);
12     risk_factor = (log2(clm_amt*clm_freq)**clm_freq)/100;
13
NOTE: There were 33676 observations read from the data set STATES.ALL_CUSTOMERS.
            WHERE (UPCASE(state)='CA') and (clm_amt>0) and (clm_freq>1);
NOTE: The data set WORK.USECASE4 has 33676 observations and 5 variables.

14    proc sort data = usecase4;
15          by descending risk_factor; run;

NOTE: There were 33676 observations read from the data set WORK.USECASE4.
NOTE: The data set WORK.USECASE4 has 33676 observations and 5 variables.

/* Next example illustrates re-writing this code to make a single pass */
```

Use Case #4: Re-written multi-pass DATA Step using PROC SQL

```
3    proc sql;
4       create table usecase4 as
5         select car_use, clm_freq,
6                state, clm_amt,
7               (log2(clm_amt*clm_freq)**clm_freq)/100 as risk_factor
8
9        from states.all_customers
10       where upcase(state) = "CA" and
11                     clm_amt > 0 and
12                     clm_freq > 1
13      order by risk_factor desc;
NOTE: Table WORK.USECASE4 created, with 33676 rows and 5 columns.

14   quit;
NOTE: PROCEDURE SQL used (Total process time):
      real time             4.64 seconds
      user cpu time         2.82 seconds
      system cpu time       1.82 seconds
```

Use Case #5: Indexing Considerations

- SAS can use either simple or compound indexes
- Generally, SAS will use an index if the WHERE clause returns ~30% or less of the rows in the data set
- SAS will always use an index if the WHERE clause returns 3% or less of the rows without doing a cost estimation
- Factors influencing index utilization:
  - Size of sub-setted rows relative to the size of the data set
  - data file value order (that is, sorted in ascending index value order or not)
  - data file page size
  - number of allocated buffers
  - cost of uncompressing data file for a sequential read

Use Case #5: Indexing Considerations

- General rules for candidate keys:
- If your data file is small, sequential processing is usually just as fast or faster.
- If your page count (available from the CONTENTS procedure) is less than three pages, do not use an index.
- Frequently changing data is not a candidate for indexing. An index is automatically updated when the data file is updated, requiring additional resources.
- If the subset of data for the index is not small, it may require more resources to access the data than sequential access.
- Options MSGLEVEL=i is also useful option to receive feedback on whether or not an index is used.

Use Case #5: Indexing Considerations

- Consider your data access needs. An index must be used often in order to make up for the resources consumed when creating and maintaining it.
- Do not use more indexes than you actually need. Find the most discriminating variables in commonly used queries and use them as your key variables.

Use Case #5: Indexing Considerations

```
1 /* policynum is not sorted and not indexed */
2 /* Note that SAS needed to perform a sort for the equi-join */
3
4  proc sql;
5   create table usecase5 as
6   select d.income,
7          d.density,
8          p.car_use,
10         p.bluebook
11   from states.demographics as d,
12        states.policy_info as p
13   where p.policynum = d.policynum;

NOTE: SAS threaded sort was used.
NOTE: Table WORK.USECASE5 created, with 10302000 rows and 5 columns.

11   quit;
NOTE: PROCEDURE SQL used (Total process time):
      real time           45.64 seconds
      user cpu time       27.69 seconds
      system cpu time     10.59 seconds
```

Use Case #5: Indexing Considerations

```
4   /* policynum is sorted and indexed */

5  proc sql;
6     create table usecase5 as
7     select d.income,
8            d.gender,
9            d.density,
10           p.car_use,
11           p.bluebook
12     from states.demographics as d,
13          states.policy_info as p
14      where p.policynum = d.policynum;
NOTE: Table WORK.USECASE5 created, with 10302000 rows and 5 columns.

15   quit;
NOTE: PROCEDURE SQL used (Total process time):
      real time           16.70 seconds
      user cpu time       12.29 seconds
      system cpu time      4.18 seconds
```

Use Case #5: Indexing Considerations

- For better index performance, sort the data into ascending order on the key variable before indexing the data file
- The more sorted your key variable data is, the better the index performance
- If appending data to an indexed file, sort the data you are appending before executing the APPEND procedure.
- Sorting an indexed data set with a different key than the index results in an ERROR. The FORCE option is used to override the ERROR, but drops the index.
- A sorted data set may be a better performer than an indexed one where many users are reading the data set

Use Case #6: Sorting Considerations

- If the input data to be read by SAS is sorted already, then use the SORTEDBY data set option to assert a sort key.
  - Follow this by a PROC SORT using the PRESORTED option for validation
  - This sets the sort-verified flag on the SAS data set
  - Some parts of SAS may choose to implement a sequence check regardless of the strength of the assertion

Use Case #6: Sorting Considerations

```
4  data read_sorted;
5     infile "C:\Users\sastrb\Desktop\Customers\States\Data\input.dat";
6
7     input @1   policynum     $12.
8           @35  gender        $1.
9           @37  income ;
10    run;

NOTE: The infile "C:\Users\sastrb\Desktop\Customers\States\Data\input.dat" is:
Filename=C:\Users\sastrb\Desktop\Customers\States\Data\input.dat <┬snip┬>
NOTE: The data set WORK.READ_SORTED has 100000 observations and 3 variables.

11  proc sort data=read_sorted presorted ;
12       by policynum;
13       run;

NOTE: Sort order of input data set has been verified.
NOTE: There were 100000 observations read from the data set WORK.READ_SORTED.
NOTE: Input data set is already sorted, no sorting done.
```

Use Case #6: Sorting Considerations

Although still supported, the **NODUPLICATES** option for Proc Sort is no longer documented.

The same results can be generated using

Proc SQL;
  select DISTINCT <list_of_variables>…

And is more efficient.

Use Case #7: Multi-Threading

- Threaded enabled processes include
  - Base SAS engine indexing
  - Base SAS procedures: SORT, SUMMARY, MEANS, REPORT, TABULATE, and SQL
  - SAS/STAT procedures: GLM, LOESS, REG, ROBUSTREG
  - EM procedures: DMREG, DMINE
  - Eligible RDBMS Access Reads

Use Case #7: Multi-Threading

- Use the CLASS statements in procedures that support them to avoid the need for sorting:
  - PROC MEANS
  - PROC SUMMARY
  - PROC UNIVARIATE
  - PROC TABULATE

Use Case #7: Multi-Threading

- Beginning with SAS 9.4:
- DS2 and FedSQL now enable the data step to be multi-threaded

Distinguishing Implicit and Explicit / Pass-Through SQL

It depends

Implicit SQL = DBMS options on libname statement

- SAS creates a connection to the DBMS
- SAS translates your code into implicit SQL

+ you don't have to know DB-specific SQL

- can be inefficient; all or portions may not translate

Explicit SQL = DBMS options on CONNECT statement + DB-specific SQL

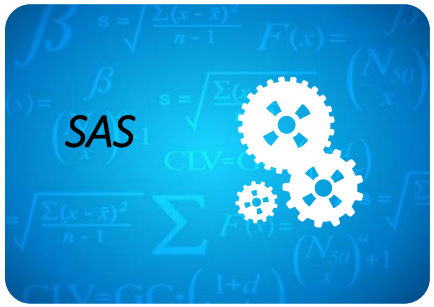
- SAS creates a connection to the DBMS
- You submit DBMS-specific explicit SQL to the DBMS

- you have to know DB-specific SQL

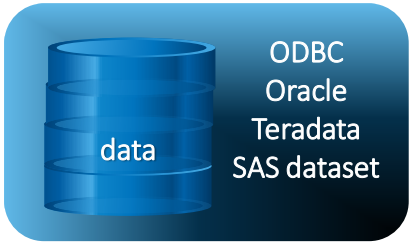
+ guarantees in-DB efficiency / no translation

Use Case #7: Avoid heterogeneous joins

SAS/ACCESS Interface



program interface



Join takes place on SAS server

ALL data moves to SAS first

SAS extracts, queries, summarizes…

Your results may cause more data movement…

- Use Case #7: Avoid heterogeneous joins
- **Minimize Data Returned to SAS for Processing**

Homogeneous

```
LIBNAME MTG '/sas/data/mortgage/';

LIBNAME HPI
  '/sas/data/housing_data/';

PROC SQL;

CREATE TABLE MTG.MYDATA AS

SELECT M.LTV, H.CURR_PROP_AMT

FROM MTG.MORTGAGE_DATA AS M

JOIN HPI.HOUSING_INDEX AS H

ON M.ACCT_NUM = H.ACCT_NUM;

QUIT;
```

Heterogeneous

```
LIBNAME MTG '/sas/data/mortgage/';

LIBNAME DRI_DBO Teradata
  Datasrc=DRI_CITY SCHEMA=dbo
  USER=&userid PASSWORD=&pwd;


PROC SQL;

CREATE TABLE MTG.MYDATA AS

SELECT M.LTV, D.REO_DATE

FROM MTG.MORTGAGE_DATA AS M

JOIN DRI_DBO.FLAT_REO AS D

ON M.ACCT_NUM = D.ACCT_NUM; QUIT;
```

• Use Case #7: Avoid heterogeneous joins

To merge SAS (or other) data with DBMS

- use pass-through SQL queries to process only the data you need on DBMS
- save the results to a SAS dataset
- merge all other SAS datasets to the newly copied dataset

To filter large amounts of DBMS data based on a smaller SAS (or other) dataset

- load the smaller SAS (or other) dataset into DBMS
- use pass-through SQL queries to process in-database (filter before join)

Use Case #8: Data Set Compression

- Compression trades reductions in I/O for increased memory utilization
- COMPRESS = YES and COMPRESS = CHAR are equivalent
  - Uses Run Length Encoding to reduce record lengths
- COMPRESS = BINARY is effective on records with numeric record lengths of 100 or more
- Compression is a permanent attribute of the data set specified on DATA statements and OUT= statements
  - Uncompressing a data set requires creating a new copy

Use Case #8: Data Set Compression COMPRESS=CHAR

```
2  proc sort data=polstate.demographics details
3                out=polstate.demographics_compressed (index=(policynum)
4                                                     compress=char);
5  by policynum ;run;
```

```
NOTE: Input data set is already sorted; it has been copied to the output data set.
NOTE: There were 10302000 observations read from the data set POLSTATE.DEMOGRAPHICS.
NOTE: The data set POLSTATE.DEMOGRAPHICS_COMPRESSED has 10302000 observations and
      20 variables.
NOTE: Simple index policynum has been defined.

NOTE: Compressing data set POLSTATE.DEMOGRAPHICS_COMPRESSED decreased size by
      44.39 percent.
      Compressed is 112326 pages; un-compressed would require 202003 pages.
NOTE: PROCEDURE SORT used (Total process time):
    real time               40.31 seconds
    user cpu time           33.71 seconds
    system cpu time          6.56 seconds
```

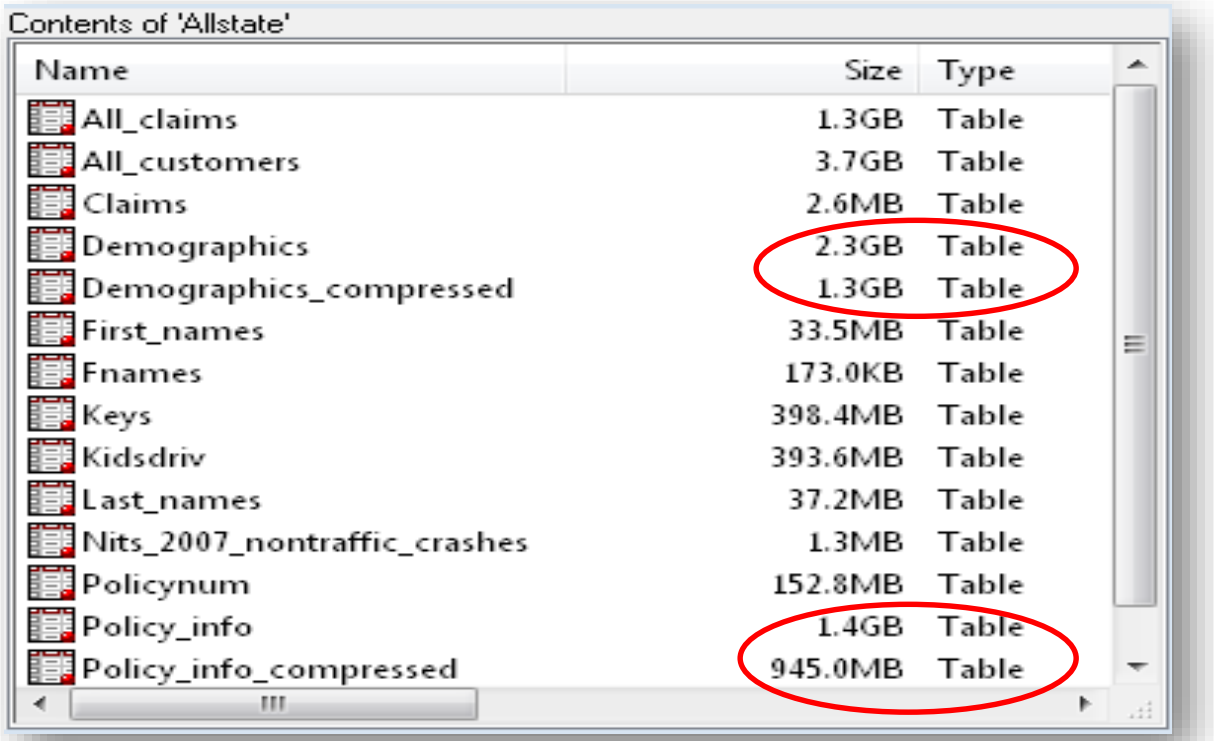
Use Case #8: Data Set Compression COMPRESS=BINARY

```
4  proc sort data=polstate.policy_info details
5                  out=polstate.policy_info_compressed (index=(policynum)
6                                                      compress=binary);
7    by policynum ;run;
```

```
NOTE: Input data set is already sorted; it has been copied to the output data set.
NOTE: There were 10302000 observations read from the data set POLSTATE.POLICY_INFO.
NOTE: The data set POLSTATE.POLICY_INFO_COMPRESSED has 10302000 observations and 15
      variables.
NOTE: Simple index policynum has been defined.

NOTE: Compressing data set POLSTATE.POLICY_INFO_COMPRESSED decreased size by 34.25
      percent.
      Compressed is 120956 pages; un-compressed would require 183967 pages.
NOTE: PROCEDURE SORT used (Total process time):
    real time               45.74 seconds
    user cpu time           29.01 seconds
    system cpu time          4.46 seconds
```

Use Case #8: Data Set Compression



Use Case #9: SAS Dataset as a Table vs. a View

- SAS Table contain descriptor portion and data portion
- SAS View contain descriptor portion and compiled instructions portion (rules describing how to retrieve the data from external sources)

Use Case #9: SAS Dataset as a Table vs. a View

- The descriptor portion for both of them contains information on what is the dataset name, what columns it has and etc.
- The data portion for the table contains the actual data.

Use Case #9: SAS Dataset as a Table vs. a View

- In summary, data tables requires more disk space, than view
- Operations with data tables are usually faster than with view
- If you want to manipulate data relevant to certain point in time you will probably want to use tables
- If you want to always use latest data then you will probably consider using a view

Use Case #9: SAS Dataset as a Table vs. a View

- Data tables contain actual data (static) and view contains no data (dynamic)!
- If source tables from which the data table is created are modified the data table data will not be updated
- For the view, if the source data is changed, you will see the latest data (because the view contains instructions how to retrieve the data, not the data).

Use Case #10: Checkpoint/Restarting SAS Jobs

- Batch SAS jobs can now set checkpoints and recover without starting the job from the beginning
- Provides a more robust error/handling and recovery
- SAS batch jobs tend to have dependencies that may not be available causing the SAS job to fail even when the program is syntactically correct. e.g.
  - Offline database table
  - Failed network access
  - Disk full condition
  - and so on…

Use Case #10: Checkpoint/Restarting SAS jobs

- For checkpoint-restart data that is saved in the WORK library, start a batch SAS session that specifies these system options:

  - SYSIN, if required in your operating environment, names the batch program
  - STEPCHKPT enables checkpoint mode
  - NOWORKTERM saves the WORK library when SAS ends
  - NOWORKINIT does not initialize (clean up) the WORK library when SAS starts
  - ERRORCHECK STRICT puts SAS in syntax-check mode when an error occurs in the LIBNAME, FILENAME, %INCLUDE, and LOCK statements
  - ERRORABEND specifies whether SAS terminates for most errors

Use Case #10: Checkpoint/Restarting SAS jobs

Starting in Checkpoint mode

The following SAS command starts a batch program in **checkpoint** mode using a user-specified checkpoint-restart library:

sas -sysin 'c:\mysas\myprogram.sas‘ –work mywork–noworkterm –noworkinit -stepchkpt -errorcheck strict –errorabend –log test.log

Use Case #10: Checkpoint/Restarting SAS jobs

```
NOTE: Begin CHECKPOINT execution mode.
NOTE: Checkpoint library: C:\Program
Files\SASHome\SASFoundation\9.4\mywork.
NOTE: WORK library: C:\Program Files\SASHome\SASFoundation\9.4\mywork.

NOTE: CHECKPOINT 1.

1          data a;
2            do n=1 to 10;
3              x=n;
4              output;
5            end;
6          run;
2                                The SAS System                10:58 Thursday, July 24,
2014


NOTE: The data set WORK.A has 10 observations and 2 variables.
NOTE: DATA statement used (Total process time):
      real time          0.00 seconds
      cpu time           0.00 seconds
```

Use Case #10: Checkpoint/Restarting SAS jobs

```
NOTE: CHECKPOINT 2.

8         data b;
9          set a;
10          if x=5 then abort abend;
11          else y=3*x;
12         run;

ERROR: Execution terminated by an ABORT statement at line 10 column 14, it specified the ABEND option.
n=5 x=5 y=. _ERROR_=1 _N_=5
NOTE: The SAS System stopped processing this step because of errors.
NOTE: There were 5 observations read from the data set WORK.A.
WARNING: The data set WORK.B may be incomplete.  When this step was stopped there were 4 observations
and 3
        variables.
WARNING: Data set WORK.B was not replaced because this step was stopped.
NOTE: DATA statement used (Total process time):
      real time          0.00 seconds
      cpu time           0.00 seconds


ERROR: SAS ended due to errors.
       You specified: OPTIONS ERRORABEND;.
ERROR: Errors printed on page 2.
```

Use Case #10: Checkpoint/Restarting SAS jobs

Restarting the job

The following SAS command **restarts** starts a batch program using a user-specified checkpoint-restart library:

sas -sysin 'c:\mysas\myprogram.sas‘ –work mywork–noworkterm –noworkinit -steprestart -errorcheck strict –errorabend

Use Case #10: Checkpoint/Restarting SAS jobs

```
NOTE: Begin CHECKPOINT-RESTART(2) execution mode.
NOTE: Checkpoint library: C:\Program
Files\SASHome\SASFoundation\9.4\mywork.

1          data a;
2            do n=1 to 10;
3              x=n;
4              output;
5            end;
6          run;
7

NOTE: End CHECKPOINT-RESTART(2) draining and resume normal
execution.
```

Use Case #10: Checkpoint/Restarting SAS jobs

```
8       data b;
9          set a;
10          /* X is never > 10 */
11          if x >10 then abort abend;
12          else y=3*x;
13         run;

NOTE: There were 10 observations read from the data set WORK.A.
NOTE: The data set WORK.B has 10 observations and 3 variables.
NOTE: DATA statement used (Total process time):
      real time          0.00 seconds
      cpu time           0.00 seconds

14
15

NOTE: SAS Institute Inc., SAS Campus Drive, Cary, NC USA 27513-2414
NOTE: The SAS System used:
      real time          0.18 seconds
      cpu time           0.18 seconds
```

Other new features

PRESERVE THE SAS ENVIRONMENT

- The Work library data sets and catalogs, and the values of global statements, macro variables, and system options can be **preserved between SAS sessions**.

- ***The PRESENV Procedure***
- The PRESENV procedure preserves all global statements and macro variables in your SAS code from one SAS session to another.

Other new features

PRESERVE THE SAS ENVIRONMENT

Set SAS System Option

```
options presenv;
```

Creates data to be used in subsequent session- submitted before exiting SAS

```
proc presenv save permdir=permdir
sascode=sascode;
run;
```

Restore

```
%include 'restore-file';
run;
```

Agenda: Use Cases

#1—Minimizing Reads on INPUT

#2—KEEP and WHERE Data Set Options

#3—WHERE clause optimization

#4—Avoiding Multiple Passes of the Data

#5—Indexing Considerations

#6—Sorting Considerations

#7—Avoid Heterogeneous Joins

#8—Data Set Compression

#9 - SAS Dataset as a Table vs. a View

#10—Checkpoint/Restarting SAS Jobs

References

Dear Miss SAS Answers: A Guide to Sorting Your Data, Stroupe and Jolley, SAS Institute Inc. at:

http://support.sas.com/resources/papers/proceedings10/140-2010.pdf

SAS Performance Tuning Strategies and Techniques, Kirk Paul Lafler, Software Intelligence Corporation, Spring Valley, CA

http://www.google.com/url?sa=t&source=web&cd=2&ved=0CCcQFjAB&url=http%3A%2F%2Fwww.scsug.org%2FSCSUGProceedings%2F2003%2FLafler%2520-%2520SAS%2520Performance%2520Tuning%2520Techniques.pdf&ei=JM7GTbPuGMPL0QH509yLCA&usg=AFQjCNF3bT6XxEvRIIqzpatMhknr23gacQ

Power Indexing: A Guide to Using Indexes Effectively in Nashville Releases at:

http://www2.sas.com/proceedings/sugi25/25/dw/25p124.pdf

SAS 9.2 Language Reference: Dictionary, Fourth Edition at:

http://support.sas.com/documentation/cdl/en/allprodslang/63337/HTML/default/viewer.htm#titlepage.htm

Ten Ways to Optimize Your SAS Code, Ron Coleman,

Internal Presentation

Top 10 Ways to Optimize Your SAS Code

Thank you!