Paper IT-02

Reducing Big Data to Manageable Proportions

Sigurd W. Hermansen, Westat, Rockville, MD, USA

ABSTRACT

Billions and billions of observations now fall within the scope of SAS® data Libraries; that is, should you happen to have an HP Blade server or equivalent at your disposal. Those of us getting along with more ordinary workstations and servers have to resort to various strategies for reducing the scale of datasets to manageable proportions. In this presentation we explore methods for the scale of datasets without losing significant information: specifically, blocking, indexing, summarization, streaming, views, restructuring, and normalization. Base SAS happily supports all of these methods.

KEYWORDS

deflate, compress, pad, pipes, filters, connection strings,SAS/Access I/O Engine, summary, redundancy, signal-to-noise ratio, SAS FORMAT, joins, views, structure, restructure, normalization, sufficient statistics, sampling.

INTRODUCTION

I enjoy the complexities and nuances of old vintages of noble wines with haute cuisine, but I also appreciate the pleasures of a fresh and clean *vin de pays* with a simple meal. So too, some thorny programming problems require elaborate solutions that take a long time to develop in the first place, and at first to understand. Other programming problems arise on what seems to be a daily basis, and these often require only a few basic methods, applied consistently. In what follows, I assure you that I will stick to the ordinary and usual solutions to the problems that programmers encounter daily when dealing with big data. By “big data”, I mean whatever size of dataset seems big to you. For some it might be tens of thousands of observations (*obs, rows, records, tuples*, take your choice) that, when read over and over again to generate different summaries, run for an hour, instead of the fifteen minutes that would give you more time to check the results. For others, it might be billions of rows of weather, stock market transaction, or census data that may require a substantial investment in new servers. In what follows, we’ll see programming methods that we can use to reduce the volume of data while keeping essential information intact. Changing at least some habits of fledgling programmers can painlessly make programs run faster and reduce disk and memory usage. Changing some of the perspectives of experienced programmers will do the same. Though specific in some instances to programming in the SAS System, these habits and perspectives have value in all other programming environments as well.

DEFLATING AND COMPRESSING DATA

A few fundamental ideas help us understand conditions under which we can reduce data volume while at the same time preserving essential information. How can we reduce the size of a balloon prior to storing it? We let the air out; that is, we *deflate* it. A database management system can do the same to a column of names of variable length, for example, by trimming trailing blanks after shorter names and padding these names with blanks when displaying them. The DBMS has to keep track of the appropriate column widths so that the repadding of names will work as required.For example, the table below has more than twice the number of characters as the listing below it:

Jones	Sue	Ann
Li	Harrison	P
Benevides-Montenegra	Lu	
Livingston	J	Paul

Jones,Sue,Ann;Li,Harrison,P;Benevides-Montenegra,Lu,;Livingston,J,Paul

Simply letting commas represent tabs and semi-colons end-of-line markers preserves the information in the table in a much smaller space.We can also *compress* data further by recoding them in ways that use less storage space on average per character. For instance, the Huffman coding algorithm computes the frequencies of characters in a sample and assigns the character codes that occur more frequently to shorter bit stings. So a single bit might represent an "e" in word in the English language, and a longer bit string a "z". The savings of storage space used by the more frequent "e"s more than offset the greater space given to the much less frequent "z"s. Computing devices as basic as smartphones have even more sophisticated data reduction programs built into their little chips. The SAS System on a basic tablet or desktop supports dynamic compression as they are being read from a source.

SAS Solution: Data Deflation and Compression

Dataset option: (compress=yes)

syntax

Data DSN (compress=yes) ….
Proc SQL; Create table DSN (compress=yes) as ….

examples

```
data test1;set test0;
2705558 obs  WORK.TEST0...11 variables.
real time  39.48 secs  cpu time 4.10 secs
data test2(compress=yes);set test1;
2705558 obs WORK.TEST1...11 variables.
NOTE: Compressing data set WORK.TEST2 decreased size by
60.93 percent.
Compressed 22981 pp; un-compressed 58817 pp.
real time 12.84 secs  cpu time 7.17 secs
data test3(compress=yes);set test2;
real time 7.90 secs  cpu time 7.45 secs
```

caveats

Data step point=, PROC FREQ column frequencies, and other SAS programs that depend on direct access to binary SAS datasets may generate errors.

extensions:

configV9.cfg file	-options compress=yes
autoexec.sas file	options compress = yes ;
start of a SAS program	options compress = yes ;
dataset level	Data BigFile (compress=yes)

The SAS System also provides LIBNAME and FILENAME "pipes" for decompressing and filtering data compressed by standard file compression packages. The excellent UCLA Web site,

http://www.ats.ucla.edu/stat/census/census2000/tips_reading_sas.htm,

documents step-by-step how to read compressed comma-separated variable (csv) files that anyone can download from the US Census 2000 Web site (see Tip#5 for Unix/Linux SAS or Tip#6 for Windows SAS). A really fast way to filter really big data takes advantage of a FILENAME pipe to decompress and stream data from compressed text files. An INFILE statement in a Data step view [e.g., `DATA vwDSN(VIEW=vwDSN); INFILE PIPE "UNZIP -P /HOME/MYFILE.ZIP" FIRSTOBS=2 OBS=11; …`] streams records into a SAS SQL query [… SELECT <variables> FROM vwDSN WHERE <conditions>]. The view vwDSN actually executes the Data step within the SQL query. The Data step executes the UNZIP command within the Data step. The variable list and the WHERE conditions subset columns and rows respectively as they are being read from the compressed text file. The referenced articles by Cohen and Llano describe in generous detail the fine points of reading uncompressed and compresssed text files.

SUMMARIZATION

Moving beyond the bits and bytes levels, we find analogs to tabs that represent patterns of blanks in a *summary* of repetitive information. In database tables parsed from an xml representation of an electronic health record (eHR), a visit to a doctor's office may generate multiple diagnosis, procedure, and other codes. The database table has not only visit date and coded outcome columns, but also columns with

thecoding system used for each code along with the version of the coding system and its name. Data for brand of medication also includes a brand name:

brand_code	nvarchar(255)
brand_code_display_name	nvarchar(255)
brand_code_system	nvarchar(255)
brand_code_system_name	nvarchar(255)
brand_code_system_version	nvarchar(255)
brand_freetext	nvarchar(4000)

While residing in a MS SQL Server database system, the Medication table in the example has variable length columns; the system automatically trims off trailing blanks. When read into a SAS dataset, with fixed column widths, the brand table devotes 5,275 characters in each row to the brand code, coding system, and name columns. Compressing the SAS dataset trims it down substantially, but why (as in some object-oriented databases) must each observation of a coded value have to have its properties stored with it? Since the same brand codes will appear repetitively in eHR's of a large group of patients, reducing the coding system and name data to one row per brand in a SAS format or an indexed look-up table preserves information while reducing the sizes of datasets: The SAS System supports both of these methods.

SAS Solution: Summarization

SAS SQL distinct query: SELECT distinct

syntax

```
procsql; createtable DSN as select distinct …
```

examples

```
procsql;
createtablebrandCodeSystemsas
selectdistinct
brand_code_system,
brand_code_system_name,
brand_code_system_version
fromMedications;
quit;
```

caveats

The SAS keyword DISTINCT tells the SAS SQL compiler to delete any rows that have the same attribute values as a row already in the dataset.

The *distinct* modifier instructs the SAS SQL compiler to delete all but one instance of any identical ***rows***. It modifies a SELECT clause that, in turn, operates on a variable list. The variable list defines each row in a table.

Although a summary table has value in its own right, it has special value as a look-up table. If we can link each row in the Medications table (above) to the brandCodeSystems look-up table, we can greatly reduce data redundancy in the Medications table.

The SAS SQL *join* operator links two tables *on* equal key values. Simply by putting equal values in key column(s) of two tables, we can link them. In effect, we are reducing redundancy in a database by splitting out columns with repeating values into a separate reference table of distinct rows. The SAS System gives us the tools we need to create look-up tables and link them.

Note that in the linked**view**, vwMedsw_bcs below, the SQL JOIN reconstructs the original **Medications** table (except for the truncated free text variable). We call this data reduction method "lossless" in that it does not sacrifice information for the sake of data reduction. "Lossy" methods trade off some loss of information for data reduction.

Summary tables with counts or sums in some instances preserve all information while dramatically reducing data volume. Cross-frequencies serve as good examples. We use them in reports and as descriptive statistics, so why not use them as data sources for report-writing and when computing inferential statistics such as relative risk, Chi-Sq tests, and linear regression? The same goes for preparing data for delivery to an analyst or client.

This election year inspires a case in point. A campaign analyst asks for data and sketches out this structure:

	Zip_9	media	adcost	votes
1	208160001	Television	28245.35	45283
2	208160002	FM Radio	2195.68	45283

Here we can make a good case for a lossy summary. Including details such as the last four digits of a zip-code, and costs down to cents seem unlikely to contribute to analytic results. Repeating the vote count by media actually leads to double-counting of votes. For a simple analysis of votes given costs by Zip_5, we can restructure the dataset, restrict the zip-code to five digits, and round costs to the nearest hundred: e.g.,

Zip_5	TVcost	Radiocost	Vote
20816	28200	2200	45283

Summing costs and votes to the Zip_5 level will reduce data complexity and volume. If a programmer maintains an archive of the original data source and the programs that created the summary dataset, it will be easy to generate additional analytic datasets. Remember the rule “Read only what you need” and you will save time as well as reduce the burden of data storage.

DATABASE NORMALIZATION

Splitting out repeating values of the brand_code_system columns into a separate table, as we have seen, reduces data redundancy. Database “normalization” builds on this idea. Database architects normalize databases to support quick and accurate databases searches and updates. Lossless data reduction comes about as a fringe benefit of good database design.

In the **Medications** table, the brand_code attribute has realistically a maximum number of characters not much greater than the number of distinct codes needed to represent all brands of medications in all potential coding systems; that is, twenty characters. The display name attribute may have the 255 characters allocated to it in the data dictionary. We assume here that the brand_freetext attribute contains descriptive data that goes beyond what a name for display would include, more what we would expect to see in a comment, but that the freetext as well as the display name remain the same for the same brand_code value (have a “functional dependence” on brand_code). If tests show that to be true, the SAS System supports normalizing of the medications table with respect to brand codes. We can shift the attributes brand_code_display_name and brand_freetext from the Medications table to a brandCodes table and use the attribute brand_code to link the two tables.Even though some database architects insist on creating artificial reference keys for each and every table link, the brand_code serves just as well as a key to the code display name and freetext. By the terms of this small instance of data architecture, the number of rows in the brand_code table will be limited to the number of distinct values of the brand_code attribute. Further assuming a functional

SAS Solution: Link Reference Table

SAS SQL JOIN: ON refKey

syntax

SAS Proc SQL and SAS Data step

examples

```
procsql;
createtablebrandCodeSysas
selectdistinct brand_code_system as
bcs,
brand_code_system_name as bcs_name,
brand_code_system_version as bcs_ver
from tmp1.Medications;
quit;
databrandCodeSystems;
set brandCodeSys;
      refKey=_N_;
run;
procsql;
createtable Meds as
select brand_code, <etc.>,
brand_code_display_name,
         trim(brand_freetext) as
brand_text length=100, r2.refKey
from tmp1.Medicationsas r1 innerjoin
brandCodeSystemsas r2
on  r1.brand_code_system=r2.bcs
and r1.brand_code_system_name =
r2.bcs_name
  and r1.brand_code_system_version
=r2.bcs_ver;
quit;
procsql; /* Reconstruct in view. */
createview vwMedsw_bcs as
selectr1.*,r2.bcs as
brand_code_system, <etc.>
from Meds as r1 innerjoin
ProviderTaxonomy as r2
on r1.refKey=r2.refKey;
quit;
```

dependency of brand_code_systems on brand code, we can use the basic normalization method to split out first all attributes related to brand_code, including those related to brand_code_system, and then split out the attributes related to brand_code_system. Once split into three tables, we can use first the brand_code_system key and then the brand_code key to reconstruct the original Medications table. Higher forms of normalization reduce still further remaining data redundanies.

SAS Solution: Normalization

SAS Database Restructuring: SQL Queries

syntax

SAS Proc SQL

examples

```
procsql;
createtable brandCodes as
selectdistinct
brand_code,
brand_code_display_name,
brand_freetext
from tmp1.Medications;

createtableMedicationsas
selectdistinct
<all attributes except
brand_display_name,brand_freetext>
from tmp1.Medications;
quit;
```

SAMPLING

Often we need really big data to lend enough statistical power to a study of a rare event. Because high frequency events tend to repeat the same patterns, lossy summarization, if well planned, increases the information/data ratio. In the data mining world, and especially in SAS's vision of data mining, sampling has a major role. Under certain conditions, alarge enough sample of big data will serve the same purpose as those big data and greatly reduce data volume.

Random sampling to reduce big data fits easily into SAS SQL and Data step programs (see inset below). The first example illustrates how to select at random from amillion row dataset, test, about 1 row per 10,000 rows into the dataset,**sampleMean**. Because we selected the sample randomly using a distribution function, we find that the sample mean comes close to the population mean that we compute using all rows of data: population mean=0.500021 vs. sample mean = 0.49439, or less than 1% difference. The accuracy of the estimate depends largely on the size of the sample and the proportion of the population sampled.

We can just as easily illustrate a downside of simple random sampling. A sample of big data containing a few needles in a enormous haystack may have all hay and no needles. The second example shows that after replacing the last hundred of a million values from one distribution with values from another distribution (class=1), the sample mean (0.495662) and population mean (0.500062) look no different than what we saw in the earlier example. If, based on what we know about these big data, we could distinguish observations from the two distributions, the difference between the sample mean and the mean narrows (class=0 and sample mean=0.500057) while the "needles" stand out clearly (class=1 and mean=0.988058). These basic examples underscore the importance of understanding a data source and recognizing its characteristics prior to using samples to compute estimates.

The SAS System provides many tools that support stratified sampling (e.g., sampling class=0 at a rate of 1 per 10,000 and class=2 at a rate of 1 per 1 as shown in the last example) as well as summarization or skewness/kurtosis statistics using PROC SUMMARY/MEANS or PROC SQL, and least/ greatest observations and the median using PROC UNIVARIATE. With the help of these tools, we can reduce data through sampling while retaining essential information.

SAS Solution: Sampling to Reduce Big Data

SAS SQL: Random Number Functions

syntax

SELECT …. WHERE *random function*< *x*

examples

```
data test; do i=1to (1E7);
    class=0; x = ranuni(1223471);
      output; end;
procsql;
createtable mean as
select mean(x) as estx from test;

createtablesampleMeanas
select mean(x) as  estx
from (select x from test
where ranuni(123453)<=0.0001);
-----------------------------------
data testS;
do i=1to (1E7 - 100);
    class=0; x = ranuni(1223459);
      output;  end;
do i=1to100; class=1;
x = min(0.9999,0.1*ranuni(1223467)+ 0.95);
output;
end;
procsql;
createtable meanM as
select mean(x) as estx from testS;

createtablesampleMeanMas
select mean(x) as  estx
from (select x from testS
where ranuni(123461)<=0.0001);

createtablesampleClassMeanas
select class, mean(x) as estx
from testS groupby class;
```

CONCLUSIONS

A few basic methods applied consistently will usually reduce big data to manageable proportions. File deflation and compression built into SAS and other database programming environments support lossless data reduction. Lossless and lossy summarization not only reduce data volume, but often make patterns in data easier to see. Normalization builds data reduction into the architecture of a database. Sampling from repetitive data, properly implemented, may scale down data collection, processing, and storage costs to minimal levels.

REFERENCES

J. Llano ***Reading Compressed Text Files Using SAS® Software*** *SUGI 31, San Francisco 2006, Paper 155-31.*

*M. Cohen****Reading Difficult Raw Data****NESUG 2008, Pittsburg 2008.*

ACKNOWLEDGMENTS

Mark Friedman contributed a valuable list of options for invoking SAS compression of datasets. Stan Legum andMichael Raithelsuggested improvements of content and presentation; despite their best efforts, the author retains sole responsibility for any errors or oversights.

DISCLAIMERS

The contents of this paper are the work of the author and do not necessarily represent the opinions, recommendations, or practices of Westat.

CONTACT INFORMATION

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

Sigurd W. Hermansen
Westat
1650 Research Blvd.
Rockville, MD 20850
Work Phone: 301.251.4268
E-mail: hermans1@westat.com