KEYED ACCESS TO SAS DATA SETS
Alec Ramsay, Satellite Business Systems

ABSTRACT

This paper discusses the storage space – computing time tradeoff inherent in many SAS data systems and presents a method for achieving keyed access to SAS data sets that eliminates the tradeoff.

1. INTRODUCTION

A significant storage space – computing time tradeoff is generally accepted when considering alternative designs for many large SAS data systems. In a design, you can minimize the space required to store data by increasing the CPU-time used to retrieve it; or you can minimize the CPU-time to retrieve data by increasing the space used to store it.

This paper outlines a simple data system architecture and an associated access method that can, for many applications, combine most of the advantages of the storage space – computing time alternatives. The method stores all like data in one SAS data set but allows direct access to logical segments of the data.

The following sections review the problem which keyed access addresses, illustrate the problem with a detailed example, and discuss the mechanics of keyed access.

2. THE TRADEOFF

The question this paper addresses is, "what is the most efficient way to organize my data, provided they have some inherent grouping, say by day, factory, region, or age?" Without keyed access to SAS data sets, you really have only two alternatives: to put all your data into one data set, or to put each group, each location for example, in a different data set. There are advantages and disadvantages to each alternative, which revolve around the tradeoff between CPU-time and storage space. We can best understand the tradeoff by examining its two principal manifestations: segmentation and consolidation.

2.1 SEGMENTATION

If you have a set of data, how much space is required to store it in one SAS data set? The size of a SAS data set is an increasing step function of the number of observations plus a constant for data management overhead.

If we decide, however, to store each group, each location for example, as a data set, we'll have a data library. How much space is required to store the same set of observations as a data library? I assert that, given a constant number of observations, the size of a SAS data library is, in general, a monotonically increasing function of the number and size of the data sets that comprise it.

This means that it takes at least as much space to store data in a data library as it does the same data in a single data set. Usually it takes more for two reasons. First, the data management overhead is incurred for each data set in a data library. Second, SAS data sets use integral numbers of tracks: if you have only one-half a track of data, it takes one whole track to store it.

This is what I call "segmentation." To summarize, the most space efficient way to store data is in a single SAS data set. If you segment the data into a data library, you waste space.

2.2 CONSOLIDATION

There is a compelling reason, however, that you might want to segment your data. If you divide your data into individual SAS data sets based on the groups in your data, each location for example, you can spend less CPU-time retrieving your data.

Define relative access time (RAT) to be the number of observations read divided by the number of observations desired. Ideally we'd like the RAT for our data systems to approach one. If RAT were one, we'd only be reading the observations we wanted.

If you segment your data and incur the space cost, you can reduce your RAT over the case where all your data are in one data set. For example, if your data are grouped by location, and you make each city's data a data set, you can retrieve Chicago's data by reading only Chicago's data, instead of all your data.

So, the most CPU-time efficient way to store a set of data is as a data library, which is exactly counter to the space-conscious recommendation. Segmenting your data wastes space, while consolidating your data degrades the access time to it. You confront this tradeoff as a system architect.

3. ILLUSTRATION

Let's use a detailed example to illustrate the design dilemma. Say you're with an up-and-coming telecommunications company, and you're considering data system architectures for a data base of telephone call records to be created daily. Let's hope, for your sake, that the data base will be very large, six figures, and that
it will get larger -- ask a marketing rep!

Say, also, that you want to be able to report on the basis of calls made from each of 100 cities you serve and on the basis of the three networks you serve them with. You then have to ask: "Should I be space conscious and put all million plus call records a day in one BAS data set, and read the whole thing every time I want to report on an individual city's calls, or should I conserve CPU-time and segment the call records in individual SAS data sets for each city, but use a veritable disk farm to store them?"

Traditionally, you'd only have two choices: a complete tradeoff of space and time, one way or the other. At best, you might have variations by degrees in between. That is, you might be able to trade a little space for a little CPU-time. An example of this would be if you grouped your cities into separate network data sets. If each of the 100 cities is part of only one of three networks, you could divide the data into three data sets, one for each network. Three data sets would require marginally more space than one large data set, but the CPU access time to the segmented data would be less.

But it would be more than it needs to be!

Next I outline a method that allows you to put all your data in one data set and read only the portions of it that you want.

4. ALTERNATIVE APPROACH

An alternative approach to the storage space - CPU time tradeoff uses the Macro Language of SAS 8.2.2 and the direct access SET statement. The general technique is to set up indexes into a SAS data set using Macro variables. I call the indexes keys, and I show you how to use them, along with the POINT option of the SET statement, to read selected segments of a SAS data set.

Keyed access to SAS data sets is a three step process.
1. Find logical segments in your data;
2. Create the keys; and
3. Read the logical segments of your data.

Each step is discussed below.

4.1 FIND THE LOGICAL SEGMENTS

This is a utility process that finds the logical segments in your data and saves data that are used to create the keys. This process must be executed only once after your data set is created. Using our previous example, if you are going to create a call data base nightly, you will need to run this part nightly as well.

The general structure of this process is shown in Figure 1. The process takes as input a data set sorted by a list of BY-variables. The BY-variables are the variables for which you want to create keys. The process finds the FIRST.byvariable and LAST.byvariable observations and creates a data set ('key_data_set') that holds for these occurrences a value, _KEY_, and its observation number.

_KEY_'s values consist of a BY-variable's VALUE plus an "F", for FIRST.byvariables, or an "L", for LAST.byvariables.

4.2 CREATE THE KEYS

This process creates the keys using the data from the first step and the SYMPUT function. It must be executed each time SAS is invoked. Once the keys are defined in a SAS session, however, they can be used any number of times.

The structure of this step is shown in Figure 2. The two variables in 'key data set' are used in the SYMPUT function. For each observation, it creates a macro variable with the name of _KEY_.'s VALUE and a value of _INDEX_.'s value.

4.3 READ LOGICAL SEGMENTS

Figure 3 shows the general form of keyed access to a SAS data set. This step assumes that the previous two steps have been performed as indicated. The five statements used to achieve keyed access to SAS data sets are DATA, DO, SET with POINT option, OUTPUT, and STOP.

Though Figure 3 is an example of using keyed access to create only one output data set from one logical segment of a larger input data set, the five statements can be combined, along with others, in many variations to meet different needs. For example, two or more logical segments could be read to create any number of output data sets, all in the same DATA Step.

Easy as 1, 2, 3, right? Try a simple example to reinforce the ideas.

5. APPLICABILITY

The technique outlined above can be extremely useful, if it fits your application. In this regard, there are both general and specific technical requirements.

5.1 GENERAL REQUIREMENTS

There are two general characteristics of applications that are ideal for keyed access to SAS data sets:

1. The number of unique values of a BY-variable is not too large.
2. The data set is large compared to the space required by the keys.

These characteristics ensure that the process of creating the keys is not too time-consuming and that the keys are small enough to be stored efficiently.

By meeting these requirements, you can leverage the advantages of keying access to SAS data sets, making your data analysis more efficient and effective.
**EXHIBIT 1: KEYED ACCESS TO SAS DATA SETS, STEP 1**

This code must be executed once after each time the input data set is created/modified.

Macro to prepare key data. Macro DefKey is used in the following data step.

```sas
%MACRO DefKey(data_type, suffix);
   IF &TYPE .. &DATA THEN DO;
      _KEY_ SCMH&DATA,l);
      INDEX N;
      OUTPUT;
   END;
%MEND DefKey.
```

Find first and last occurrences of key variables in the input data set. Save the key-variables's values and the observation numbers. 'INPUT_DATA_SET' must be in BY-VARIABLE ORDER.

```sas
DATA 'KEY_DATA_SET'(keep=_KEY_ _INDEX_);
   SET 'INPUT_DATA_SET';
   LENGTH _KEY_ 8 _INDEX_ 8;
   BY 'LIST_OF_KEY_VARIABLES';
   %DEFKEY('VARIABLE1',FIRST,'F')
   %DEFKEY('VARIABLE1',LAST,'L')
   ... 
   PROC PRINT DATA='KEY_DATA_SET';
```

**EXHIBIT 2: KEYED ACCESS TO SAS DATA SETS, STEP 2**

Define key macrovariables. This must be executed each time SAS is invoked.

```sas
DATA _NULL_;
   SET 'KEY_DATA_SET';
   FILE LOG HEADER=H;
   CALL SYRIPUT('KEY',_INDEX_);
   PUT 210 _KEY_ #25 _INDEX_;
   RETURN;
   H:
   PUT 210 'THESE ARE THE KEYS YOU JUST SET UP: ';
   RETURN;
   RUN;
```

**EXHIBIT 3: KEYED ACCESS TO SAS DATA SETS, STEP 3**

Sample keyed access to SAS data set. This data step selects a segment of 'INPUT_DATA_SET' and creates 'DATA_SET'.

```sas
DATA 'DATA_SET';
   DO I = &FROM_KEY TO &TO_KEY;
      SET 'INPUT_DATA_SET' POINT=I NOBS=NOBS;
      OUTPUT;
   END;
STOP;
```
1. There must be some logical grouping in your data. Some examples I've offered above are by day, factory, city, or country; and

2. It really only pays off to use keyed access to SAS data sets if you have a severe problem with segmentation or consolidation.

There are three main manifestations of segmentation:

1. Overhead space is a large fraction of the total space used in your data library. An example of this would be if your data library used 100 tracks of which 50 were overhead.

2. Overhead space is simply large. An example of this would be if the overhead space in your data library were 500 tracks.

3. The number of observations per data set is small compared to the number of observations SAS stores per track. An example of this would be if you had 100 observations in each data set, but SAS could store 1000 in each track.

There are two principal symptoms of consolidation:

1. The number of observations you have to read is large compared to the number of observations you want. An example of this would be if you had 10,000 observations but only wanted to find 10 of them. This tends to characterize ad hoc queries.

2. The number of observations you have to read is large in absolute terms. An example of this would be if you had 1,000,000 observations.

3.2 TECHNICAL REQUIREMENTS

There are two technical requirements for implementing keyed access, as I presented it above:

1. All keys must be unique.

2. Key values must conform to macro variable naming conventions.

Each key variable you select must itself uniquely segment the input data. In our call data base example, locations were unique across networks. If, however, a location could have been serviced by more than one network, then we could not apply the technique directly because locations would not have uniquely segmented the data. In such a case, a slight reworking of the data might still allow keyed access to be used. Here we might be able to concatenate network and location names to create a new variable that would uniquely segment the data.

Macro variable naming conventions impose some limitations on the data you can use for keys. Keyed access will not work directly with data that contains special characters or that does not start with a character. There is also a seven character limit on the variable values that you can use as keys. The eighth character of keys is used to distinguish between the first and last occurrences of the key variables.

These requirements are not terribly limiting, and there are probably slick ways to work your application around them should they arise.

6. SUMMARY

I presented an alternative to the traditional storage space - CPU time SAS data system design tradeoff. If you have a large SAS data reporting system, especially one that requires ad hoc queries, you should investigate the keyed access technique further. I hope these ideas can help you make your SAS jobs even more efficient!

7. FOOTNOTES

1. Author’s address: Alec Ramsay, Satellite Business Systems, 4283 Greensboro Dr., Mc Lean, Va. 22102, Mail Zone M5E

Satellite Business Systems hereby disclaims any liability of any nature arising out of any use of or reliance upon this Article by any person or entity. Satellite Business Systems is a general partnership organized under the laws of the State of Connecticut, and all persons or entities exposed to this Article agree that recourse for satisfaction of claims of any nature against Satellite Business Systems or any of the partners in Satellite Business Systems arising in connection with the release contemplated hereby shall be had solely to the property of Satellite Business Systems and not to the non-partnership property of any of the partners.