Why Be Normal?
Constructing SAS-based Tools to Facilitate Database Design

Peter Kretzman, McCaw Cellular Communications, Inc.

ABSTRACT

Now that the SAS System (as of Release 6.06) features indexed data sets, it's more viable to do real database design using SAS software. This paper presents a number of automated techniques, implemented using SAS macros, to aid in the normalization, analysis, and documentation of SAS databases. These tools, combined with classical database design methods such as entity-relationship diagrams, help you take SAS-based applications out of the realm of ad hoc data hodgepodge and into the realm of production database design.

This paper is especially oriented toward SAS software users without much relational database design experience. During the presentation of these macro tools, the paper will also:

- cover what normalization is; why be normal?
- discuss briefly the performance tradeoffs, advantages, and disadvantages of database normalization
- present one approach to the construction of a SAS-based data dictionary model
- enumerate some strengths and weaknesses of the SAS System with respect to a data dictionary approach.

INTRODUCTION

The SAS System, as I have discussed in various previous SUGI papers (e.g., Kretzman, 1987), has its roots deeply and firmly in the ad hoc. The traditional strength of the system has been the ease by which the user could fly by the seat of his or her pants, creating new data sets, generating reports, transforming data into new and more useful shapes. Of much less (or no) concern, especially in those early days of the SAS System, were production database concerns such as integrity, ease of updates, or even performance. Analysis needs were not just paramount, but often took precedence over database design, efficiency, and maintainability when decisions were made on how to work with data in SAS. That practice made perfect sense at the time in most of the installations that exhibited it (I worked in one), and it probably continues to make sense for some individuals, for example. I should emphasize, incidentally, that it is not at all my intention to disparage the ad hoc way of doing things, but rather to point out potentially different approaches when confronted with different needs (i.e., production). In any case, the analyst may have referred to this ad hoc, dynamic collection of data sets as a database, but in a more rigorous sense of that term, such a collection is better characterized as a data store.

To define database more strictly for my purposes: a database is built carefully according to design, and is updated with clearly defined procedures. The entities themselves within that database (NOTE: entities, also referred to as tables in this and other SUGI papers on this subject, are simply another term for SAS data sets), as opposed to their contents, remain for the most part constant; with formal databases, more tables are not added regularly or even occasionally, unless the database is found to be deficient in some respect in a design sense.

As new data comes in, it is added, in the form of new or updated rows (or observations, in SAS parlance), to existing tables. Subsets of the data that are taken for analysis or other purposes are placed elsewhere, not in the permanent database library. But in many cases, subsetting or otherwise copying data that is contained in a production database is regarded as extremely dangerous and thus as something to be avoided. The whole point of setting up a production database, using the SAS System or any other package, is really to obtain control: control over when and how the data is updated, who has access to it, how it will be retrieved, and so on. The users of the database submit to such controls (when they have a choice, that is) in order to ensure, to the extent possible, accurate and consistent data in the database when they need it.

Finally, with the advent of Release 6.06 of the SAS System, the ante has been raised yet again. Questions of efficiency and the how to determine the correct approach to a data manipulation problem have been made more complex than ever, with the availability of indexed data sets, compression, SQL views, and so on. This paper is intended to be an initial examination of a few of the issues that arise as the SAS community begins to shift its focus, perhaps only in part, away from an overriding emphasis on analysis-oriented needs and towards an emphasis on production concerns.

What is a database?

The first issue of importance as we make this shift, an issue that many ad hoc users of SAS have never really felt the need to consider before now, is what exactly constitutes a database. For analysis purposes, most SAS data set libraries contained whatever data sets the analyst was working on: subsets of master files frozen at whatever point in time they were pulled, monthly or yearly files of time series data, and so on. Indeed, it was often the mark of the immensely skilled analyst that he or she was able to keep track of what data had or had not been edited, what data needed updating or refreshing, and so on. No matter what, however, discrepancies are bound to creep in: anomalies discovered and corrected in the main body of data often don't make it into the "hip pocket" subsets maintained by individuals, for example. I should emphasize, incidentally, that it is not at all my intention to disparage the ad hoc way of doing things, but rather to point out potentially different approaches when confronted with different needs (i.e., production). In any case, the analyst may have referred to this ad hoc, dynamic collection of data sets as a database, but in a more rigorous sense of that term, such a collection is better characterized as a data store.

To define database more strictly for my purposes: a database is built carefully according to design, and updated with clearly defined procedures. The entities themselves within that database (NOTE: entities, also referred to as tables in this and other SUGI papers on this subject, are simply another term for SAS data sets), as opposed to their contents, remain for the most part constant; with formal databases, more tables are not added regularly or even occasionally, unless the database is found to be deficient in some respect in a design sense.

As new data comes in, it is added, in the form of new or updated rows (or observations, in SAS parlance), to existing tables. Subsets of the data that are taken for analysis or other purposes are placed elsewhere, not in the permanent database library. But in many cases, subsetting or otherwise copying data that is contained in a production database is regarded as extremely dangerous and thus as something to be avoided. The whole point of setting up a production database, using the SAS System or any other package, is really to obtain control: control over when and how the data is updated, who has access to it, how it will be retrieved, and so on. The users of the database submit to such controls (when they have a choice, that is) in order to ensure, to the extent possible, accurate and consistent data in the database when they need it.
Finally, another reason for a more formally designed database is to provide maximum flexibility in retrieving the data. Carefully crafted data tables, with their relationships to other tables clearly known, can in many cases (although not necessarily all) allow for easier query on the database to serve a variety of needs, or exhibit better performance than retrieving data that has been informally specified or often spliced together in the heat of the moment in an ad hoc MERGE. For a more extended discussion of the other side of this not always simple coin, I direct the reader to Patricia Gerend's excellent paper in this same volume (Gerend, 1991).

Normalization

For reasons of both data integrity and performance, it is the (at least abstract) goal of a production database, typically, to represent each fact or data item controO to find and update the particular data item added storage. So a significant part of database design, in the only causes potential elTors happens to occur in the data base. This data redundancy not only causes potential errors in data maintenance, it also requires added storage. So a significant part of database design, in the more formal sense we have been discussing, is to eliminate such data redundancy, thereby saving disk space and alleviating data integrity problems.

A quick, and hopefully sufficient, glossary of terms: Normalization is the technical name for the process that reveals and then eliminates data redundancy. In essence, relationships between database tables in general, and the completeness or incompleteness of normalization in particular, is determined by how the various key fields (field is simply another name for variable) are set up in each table. Each table in a formal database has a primary key, which is a field or fields that uniquely identifies each record in the table. Any table may also have one or more fields that serve as foreign keys, which are fields with the same name as a primary key field in another table. With the foreign key concept, a relational database management system can quickly access information referred to by the current table (where the foreign key appears) by "looking up" the value of that field in the table where the same variable appears as a primary key. For example, if a table had a foreign key of State Code, the value in that field for a given record could be used against the State table to obtain the state's full name, population, tax rate, and so on.

Constructing a data model consists of determining, before any code is written, what basic tables are needed, what the primary keys (or distinguishing features, in other words) of those tables will be, and what the key relationships will be. Analyzing the occurrence of primary and foreign keys in the data model shows you what relationships exist in your data base, completely apart from the data itself. In order to normalize your data model, after it has been sketched initially, you examine it closely to determine if it matches first, second, and then third normal forms. You then make changes to the data model and reevaluate, continuing until those normal forms are compiled with. Later, for performance or retrieval reasons (see Gerend, 1981), it may be required that you denormalize some of your tables, but reasons to do so are beyond the scope of this paper.

Rather than go into great detail in limited space about how to determine whether a database is in the various normal forms, this paper will only touch upon the theory. The reader is directed to James R. Johnson and Roger D. Comejo's paper in the SUGI 15 Proceedings or, in more detail, to C. J. Date's classic textbook (Date, 1990) for a good discussion of the various forms of normalization. Other papers in this year's Proceedings also deal with this issue.

Statement of purpose

Instead of dealing more extensively with theory, the intended focus of this paper is to make an initial inquiry into ways that the SAS System itself can be utilized to help examine a database and its key relationships. The tools and methods described here, particularly if extended even further, can be of vital assistance in the normalization and database design process.

As stated before, *analyzing the occurrence of primary and foreign keys in the data model shows you what relationships exist in your data base, completely apart from the data itself.* This sentence is the primary key, so to speak, to this paper. Using the SAS System to write a series of macros and PROC SQL views, we will show how the actual structure of data within a given SAS data set library may be examined and analyzed in an automated fashion. It turns out that the SAS System has (or permits construction of all the necessary tools to compile and then examine data on its own data sets and relationships.

A SAS-BASED DATA DICTIONARY

The SAS System is, however, a Johnny-come-lately as a true relational database management package. In the SAS System, indexes are new, SQL is new, compression is new; as a result, a good portion of the basic functionality offered by other RDBMS packages is now present in the SAS System. What still remains to be added (and it may not even be on the drawing boards as far as I know) is a truly integrated data dictionary that would be maintained automatically as tables are changed or updated. At present, the SAS System has the capability to find out nearly everything about the tables within a data set library — but it takes active steps (most notably, a PROC CONTENTS with an output data set, followed by some manipulation) by the programmer to compile that information.

That is the basic technique used by this paper (not exactly rocket science!) One other key assumption behind this kind of automated analysis needs to be documented: In order to determine for certain that a given table has a given primary key, we must, in the SAS System, rely on the fact that an index has been defined for it. This means that the database must have been constructed with this sort of analysis in mind; it is, of course, still quite possible to construct whole (and perfectly viable) systems using SAS that feature no use of indexes at all. The macros described in this paper obtain their information on primary keys solely from whatever indexes have been defined on the tables within the database. One report shown in the code in the Appendix, in fact, details those data sets for which no indexes have been defined, and which, for the purposes of this automated analysis, therefore don't have any primary keys.

The main setup macro, DBLOOK, as well as a few sample reports (contained in a macro named DBREPORT) is given in Appendix A. Both of these are examples of "Teflon and Tang" macros (see Kretzman, 1988), in terms of style and approach. DBLOOK, the core macro, does all the setup work, starting by calling PROC CONTENTS. The PROC CONTENTS accumulates virtually all the available information on each of the variables in each of the data sets in the referenced data set library, and places it into a temporary output data set. Following that step, a set of PROC SQL code creates predefined views that serve the user as temporary data set aliases. The following views are created:
View Name | View purpose
--- | ---
V_TABLES | Lists all tables in the database
V_VARS | Contains all variables for all DATA tables
V_PK | Obtains all primary keys
V_FK | Obtains all foreign keys

DBLOOK also prints two reports, simply listing the primary keys and the foreign keys within the referenced database. These reports for our sample database (as modeled in the diagram in Appendix B) look as follows:

<table>
<thead>
<tr>
<th>Primary keys in the data database</th>
<th>Use of in indexes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Library</td>
<td>Member Name</td>
</tr>
<tr>
<td>ACCOUNT</td>
<td>ACCOUNT</td>
</tr>
<tr>
<td>BANKS</td>
<td>BANKNUM</td>
</tr>
<tr>
<td>CALLHIST</td>
<td>DATE</td>
</tr>
<tr>
<td></td>
<td>PHONE</td>
</tr>
<tr>
<td></td>
<td>TIME</td>
</tr>
<tr>
<td>PAYMENTS</td>
<td>ACCOUNT</td>
</tr>
<tr>
<td></td>
<td>PAY_DATE</td>
</tr>
<tr>
<td>PHONE</td>
<td>PHONE</td>
</tr>
<tr>
<td>STATE</td>
<td>STATE</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Foreign keys in the data database</th>
</tr>
</thead>
<tbody>
<tr>
<td>Library</td>
</tr>
<tr>
<td>ACCOUNT</td>
</tr>
<tr>
<td>CALLHIST</td>
</tr>
<tr>
<td>PAYMENTS</td>
</tr>
<tr>
<td>PHONE</td>
</tr>
</tbody>
</table>

**Reporting Tools**

Following a call to DBLOOK, an analyst can then run DBREPORT, which uses the information that DBLOOK has placed in appropriate work data sets and in the SAS data set views. Currently implemented (and again, this is intended as an initial direction or investigation, rather than as a completed project) are the following reports:

1. **Non-key variables occurring in more than one table.** If a given non-key variable occurs in two places in the database, this indicates that the goal of eliminating data redundancy and representing each fact or attribute in a single place has not been met.

2. **Primary keys that don't occur as foreign keys.** If a primary key doesn't occur anywhere in the database as a foreign key, there is no reason for the table containing it as a primary key to exist.

3. **Data sets for which no indexes have been defined, and which, for the purposes of this automated analysis, therefore don't have any primary keys.**

4. **List of one-to-many relationships in the database.** This list allows the analyst to see at a glance just what the key relationships are in the database. This list can usually be directly compared to the data model in order to check for discrepancies.

**EXAMPLE APPLICATION**

For the purposes of demonstrating some of the sample reports delivered by DBREPORT, a sample database was established. Using the example of cellular telephones, data tables were designed that would represent the following (simplified) real-world situation: An account may have several telephones, each of which makes multiple telephone calls on any given day. Invoices and payments are tracked by account, not by telephone. A state table contains information, by two-letter state code, on the state name and tax information.

Five tables were established to mirror this situation: a PHONE table, an ACCOUNT table, a CALLHIST table to contain the individual call records, a PAYMENTS table to record payments for the accounts, and a STATE table to contain state information.

(See Appendix B for the data model diagram. This data model diagram shows the variables within each table, as well as diagramming the one-to-many relationships and the primary and foreign keys.)

The PHONE table is indexed by phone number (in other words, has that as its primary key), but also contains the account number variable as a foreign key, so that information can be obtained on the account’s name, address, and so on. The ACCOUNT table is indexed on ACCOUNT number. The CALLHIST table is indexed on phone number, date and time, in order to keep each record distinct. Note that the telephone number in the CALLHIST table serves as a foreign key into the PHONE table, so that information can be obtained for the individual phone (such as phone type). The PAYMENTS table is indexed on account number and date of invoice. Finally, the STATE table is indexed on two-letter state code abbreviation; the ACCOUNT table uses that state code as a foreign key to obtain other information about the state.

In general, this is not a very complex database, and the test code has to introduce intentional anomalies in order to test the reports appropriately. It does so by erroneously adding AMT_OWED to the ACCOUNT table, even though AMT_OWED is already in the PAYMENTS table. Our first report thus kicks it out in the following report:

| Non-key attributes occurring in more than one table |
| --- | --- | --- |
| Library | Member Name | Variable Name |
| PAYMENTS | AMT_OWED | Amount owed |
| ACCOUNT | AMT_OWED | Amount owed |
Then, to check the second report (Primary keys that don't occur as foreign keys), it is necessary to introduce a bogus table, in this case BANKS. BANKS thus appears on the list since its primary key occurs nowhere as a foreign key.

Finally, and perhaps most usefully, a report is printed that lists the One-to-Many relationships in the database. This report can be compared to the data model and can unearth discrepancies in how indexes were set up and so on. It appears like this:

<table>
<thead>
<tr>
<th>One to many relationships in database</th>
<th>Linking variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>ONE ACCOUNT PHONE ACCOUNT</td>
<td>ACCOUNT</td>
</tr>
<tr>
<td>ACCOUNT PAYMENTS ACCOUNT</td>
<td>ACCOUNT</td>
</tr>
<tr>
<td>PHONE CALLHIST PHONE</td>
<td>PHONE</td>
</tr>
<tr>
<td>STATE ACCOUNT STATE</td>
<td>STATE</td>
</tr>
</tbody>
</table>

This report allows easy, intuitive examination of the data model's assumptions: one account has many phones, one account receives many payments, one phone has many CALLHIST records, and one state has many accounts.

**CONCLUSION**

We have begun our inquiry into ways to utilize the SAS System itself to assist in the process of a robust database design. This paper is, however, only an initial investigation into these issues. The tools described herein are but a beginning; many more are possible. Tools could be constructed that will compensate for SAS weaknesses: for example, to examine whether or not key fields used as foreign keys are ever NULL, or to determine the appropriate length for given variables based on current data. The possibilities have few limits.

Comments, suggestions, and questions are welcome. The author may be reached at the following address:

Peter Kretzman  
McCaw Cellular Communications, Inc.  
P.O. Box 86865  
Kirkland, WA 98033-9765  
Voice: (206)828-1344  
FAX: (206)828-1300

In addition, the author may be contacted via Internet, using the address kretzman@u.washington.edu.

**ACKNOWLEDGEMENTS**

The author wishes to thank the following people for their contributions (wittingly or not) to the thought that went into this paper: Patricia Gerend, Gene Hart, Paul Kent, and Al Williams.

**BIBLIOGRAPHY**


SAS is a registered trademark of SAS Institute, Inc., Cary, NC.

Appendix A

**Source Code Listing**

```sas
%macro dblook
  (lib=DATA, /* LIBREF containing the data base */
   out=_vars /* work data set to hold contents */
  );
  proc contents data=&lib,,_ALL_ noprint
    memtype=data
    out=&out;
  run;
  proc sql feedback;
    create view v_tables as
      select distinct memname, memlabel, engine, crdate, modate, delobs, idxcount
      from &out;
    create view v_vars as
      select distinct memname, name, type, length, varnum, label, format, formatl, formatd, idxusage
      from &out;
  run;
  proc sql feedback;
  %end;
```

%dblook

create view v_PK as 
select memname, name, idxusage 
from &out 
where idxusage ne 'NONE' 
order by memname, name 
;
create view v_FK as 
select distinct t.memname, t.name 
from v_vars t, 
v_PK i 
where i.name=t.name and 
t.memname ne t.memname and 
t.idxusage ne 'SIMPLE' 
order by memname, name 
;
quit;
run;
proc print data=v_PK label;
by memname;
id memname;
title "Primary keys in the &lib database";
run;
proc print data=v_FK label;
by memname;
id memname;
title "Foreign keys in the &lib database";
run;
%mend dblook;

%macro dbreport 
();
proc sql;
title 'Non-key variables in more than one table'; 
select memname label='Table name', 
    name label='Variable name', 
    label label='Label' 
from v_vars 
where name not in (select name from v_PK) 
group by name 
    having count(*) gt 1 
;
title 'Primary keys not present as foreign keys'; 
select memname label='Table name', 
    name label='Primary key' 
from v_PK 
where name not in 
    (select name from v_PK) and 
    idxusage eq 'SIMPLE' 
;
title 'Tables where no index has been defined'; 
select memname label='Table name' 
from v_vars 
where memname not in 
    (select memname from v_PK) 
;
title 'One to many relationships in data base'; 
select pk.memname label='ONE ', 
fk.memname label='MANY ';
quit;
run;
%mend dbreport;

* make data for the hypothetical data base;
libname data 'misc/sugHalk/data';
data data.phone(keep=PHONE ACCOUNT PHONETYPE FEATURE1 FEATURE2 FEATURE3);
    attrib PHONE 
        length=$10 
        label='Cellular Telephone Number';
    attrib ACCOUNT 
        length=$7 
        label='Account Number';
    attrib PHONETYPE 
        length=$1 
        label='Type of phone';
    attrib FEATURE1 
        length=$1 
        label='Call Waiting';
    attrib FEATURE2 
        length=$1 
        label='Call Forwarding';
    attrib FEATURE3 
        length=$1 
        label='Three-Way Calling';
run;
data data.account(keep=ACCOUNT NAME ADDR CITY STATE);
    attrib ACCOUNT 
        length=$7 
        label='Account Number';
    attrib NAME 
        length=$20 
        label='Name';
    attrib ADDR 
        length=$25 
        label='Address';
    attrib CITY 
        length=$15 
        label='City';
    attrib STATE 
        length=$2 
        label='State';
run;
data data.callhist(keep=PHONE DATE TIME NUMCALL DURATION);
    attrib PHONE 
        length=$10 
        label='Cellular Telephone Number';
attrib DATE
length=4
format=mmddyy8.
label='Date of call';
attrib TIME
length=4
format=tod.
label='Time of call';
attrib NUMCALL
length=4
label='Phone Number Called';
attrib DURATION
length=4
label='Duration of Call';
run;
data data.payments(keep=ACCOUNT INV_DATE AMT_OWED AMT_PAID DATEPAID);
attrib ACCOUNT
length=7
label='Account Number';
attrib INV_DATE
length=4
format=mmddyy8.
label='Date of invoice';
attrib AMT_OWED
length=4
label='Amount owed';
attrib AMT_PAID
length=4
label='Amount paid';
attrib DATEPAID
length=4
format=mmddyy8.
label='Date paid';
run;
data data.state(keep=STATE STATENAM);
attrib STATENAM
length=15
label='State name';
attrib STATE
length=2
label='State';
run;
data data.banks(keep=BANKNUM BALANCE);
attrib BANKNUM
length=4
label='Bank number';
attrib BALANCE
length=4
label='Balance';
run;
proc sql;
create index PHONE on
data.phone(phone);
create index ACCOUNT on
data.account(account);
create index TIMESTMP on
data.callhist(PHONE, DATE, TIME);
create index ACCIDATE on
data.payments(ACCOUNT, INV_DATE);
create index STATE on
data.state(STATE);
create index BANKNUM on
data.banks(BANKNUM);
quit;
run;
* Now set up an extra (erroneous) condition by
adding a non-key to a second data set;
data data.account;
set data.account;
attrib AMT_OWED
length=4
label='Amount owed';
run;
* recreate index on the above data set;
proc sql;
create index ACCOUNT on
data.account(account);
quit;
run;
* Now do the work;
%dblookup(lib=data);
%dbreport;
Sample Data Model

STATE
* State
State Name

ACCOUNT
* Account Name
Address
City
State

PAYMENTS
* Account
* Invoice Date
Amt Owed
Amt Paid
Date Paid

PHONE
* Phone
Account
Phone type
Call Fwd
Call Wait

CALLHIST
* Phone
* Date
* Time
Num Called
Duration