Paper 160-27

Unit-of-Analysis Programming

Vanessa Hayden, Fidelity Investments, Boston, MA

ABSTRACT

There are many possible ways to organize your data, but some ways are more amenable to end-reporting and analysis than others. A common tendency is to simply denormalize clinical data to the lowest level and build reports from there. Not only is this inefficient, but the logic is harder to trace and can lead to programming bugs. Building data at the unit of analysis level will help you avoid these common pitfalls and inefficiencies. This paper will overview basic database structures, simple denormalization, and alternate ways of aggregating and storing your data, based on the unit of analysis. It will demonstrate how such a design facilitates reporting and analysis, and stands up to scrutiny. This paper is oriented towards anyone who works with data, regardless of SAS® software experience. Examples will be presented using SAS software code, but the concepts apply regardless of the specific data management tool used.

INTRODUCTION

Clinical research data are often supplied as flat files or SAS System tables[1]. Typical documentation comprises the output of the CONTENTS procedure output for each table, a printout of the first 10 records, an annotated case report form, and a codebook. You as the programmer must determine – mostly from the annotated case report form – the exact nature of the relationships among the tables and how they can be combined to create meaningful, valid answers to clinical questions[2].

Without a thorough understanding of the data you may overlook important methodological points affecting data analyses. For example, suppose a table called *scores* contains weekly mobility scores for 150 randomized patients with arthritis. The statistical analysis plan calls for the percentage of patients who achieved a mobility score of 50 or higher on a 100 point scale, at any point during the study. One approach would work as follows (see Figure 1 for the corresponding SAS System log messages):

1. Find the number of unique patient IDs in the scores table where mobil50=1.
2. Manually divide the result from the log (63 unique patients with mobil50=1) by the number of patients enrolled in the study (which is already known).

The statistic can now be computed as 42% (63/150 patients); however, this manual computation masks the nature of the denominator. One hundred and fifty patients may have been randomized, but how many completed the study? How many showed up at the clinic for at least one weekly assessment? How many even received treatment? Suppose 12 randomized patients walked out of the study immediately after randomization and baseline assessment, without receiving any treatment and without any follow-up assessments. Removing those 12 patients would change the denominator to 138, yielding a percentage of 45.7%.

[1] The following terms will be used in this paper:

table	a data set or entity
column	a field, variable, or attribute
row	a record or observation
database	a set of tables containing related information

[2] Database programmers typically communicate the database schema with a physical or logical data model, a visual diagram that spells out the relationships among the tables; however, these entity-relationship diagrams do not seem common in clinical research.

Assume: Clinical trial with 150 randomized asthma patients.

Question: What percentage of patients achieved a mobility score of 50 or better?

```
proc sort nodupkey
          data=work.scores
          out=work.himob(keep=patid)
  by patid;
  where mobility >= 50;
run;
NOTE: 342 observations with duplicate key
values were deleted.
NOTE: The data set work.himob has 63
observations and 1 variables.
```

Manually Compute: 63/150 = 42%

Figure 1. On strategy to answer the mobility question for asthma patients. Is the 42% figure correct?

The purpose of this paper is not to recommend the appropriate denominator. That is a statistical and regulatory (in the case of clinical trials) question, and plenty of methodological literature addresses the question (e.g., Klein, 1998). The purpose is to demonstrate that your ability to anticipate and handle important data issues will depend on the approach you take to organizing the data.

The programming approach described in this paper organizes analytical data in a way that helps you

- explicitly pick a relevant denominator.
- avoid manual computations
- write self-documenting code
- produce multiple report variations with minimal effort

The techniques used in this paper use macro variables and simple stored macros. Throughout the paper, user-defined macro names (names of macro variables and macro routines) will be printed in bold-italic font. For a brief overview of macro programming and a discussion of self-documenting code, see Appendix A of the author's other SUGI 2002 paper in the Coders' Corner section: Paper 98-27, Bulletproofing your SAS® Results. All clinical and database examples in this paper use fictional data. The opinions expressed herein are solely the author's and do not necessarily reflect the opinions of her employer.

UNIT-OF-ANALYSIS PROGRAMMING

Programming toward the unit of analysis (UA) enables you to anticipate and address many common data issues as they arise. For the purposes of this paper, the UA is defined as the primary unit of interest for research[3]. In pharmaceutical research the UA

[3] The phrase "Unit of Analysis" is borrowed from the statistical literature, where there is much discussion about the appropriate level of analysis with hierarchical data. See, for example
http://trochim.human.cornell.edu/kb/unitanal.htm
http://w3.nai.net/~dakenny/u_o_a.htm
http://www.tufts.edu/~gdallal/units.htm

would typically be the patient, but if you are working with claims data your UA might be the inpatient hospitalization stay. The UA is the level at which units are sampled and statistics are performed (in survey research the UA could either be individual or the household). UA-oriented programming involves five basic steps:

1. Identify the primary UA for reporting and analysis.
2. Relate the database structure to the primary UA.
3. Determine which questions are answerable.
4. Audit primary keys.
5. Build tables at the primary UA level.

For the example presented in the introduction, it would be helpful to have several patient-level flags that indicate sub-populations of interest for reporting. In addition, the final reporting data set should contain patient-level outcomes built off the detail data (boolean for achieving a mobility score of 50, best mobility score achieved, average mobility score, etc.).

Table 1. Patient-level table including subpopulation flags and analysis variables.

Column Name	Column Description
patid	Patient ID
mob50flg	Did patient ever achieve a mobility score of 50 or better?
complflg	Did patient complete the study?
post2flg	Did patient have at least two post-basement assessments?
maleflg	Is the patient male?
bestmob	Best mobility score achieved post-baseline
avgmob	Mean modility score post-baseline

With this data table, you could write a simple macro that would run statistics for different subsets of the original intent-to-treat sample. A self-documenting title applying the same *&wherstmt* displays the criteria used for each report.

```
%macro runpct(wherstmt);
title1 "PATIENT SUBSET:  &wherstmt";
proc freq data=perm.ptlevel;
  tables mob50flg;
  &wherstmt;
  title2 "Achieved Mobility of 50+ (%)";
run;
proc univariate data=PERM.ptlevel;
  var bestmob avgmob;
  &wherstmt;
  title2 "Best and Average Mobility Scores";
run;
%mend runpct;
%runpct(wherstmt=where completer=1);
%runpct(wherstmt=where postbase=1);
%runpct(wherstmt=where female=1);
```

IDENTIFY THE PRIMARY UA FOR REPORTING & ANALYSIS

The first step, albeit obvious, is to identify your primary UA. In clinical research, this is generally straightforward; you are studying the patient. In other cases there are decisions to make. For example, when evaluating a marketing campaign, do you want to see the account activity of a particular individual, or of the entire household? When examining loan center productivity, are you interested in the calls answered by loan representatives, or only in completed loan applications? The answer depends on the specific research question being asked.

Once you have identified your primary UA, consider secondary or interim UAs that you may require in building analysis variables. For example, to report the average number of hospitalizations per patient, you must first identify the unique hospitalizations in the table. (In typical claims databases, multiple claims make up a single inpatient hospital stay.) The easiest way to count hospitalizations per patient is to first create a table with one row for each unique hospitalization, then sum those by patient in a separate step. You could skip this intermediate step using the RETAIN statement in the DATA step, but that logic is harder to follow and is much more likely to result in bugs.

RELATE THE DATABASE SCHEMA TO THE PRIMARY UA

Before you start programming with new data, you need to understand the database schema (how the data are organized into tables). It is helpful to understand some basic principles of database design, and the reasons for them.

The simplest form of a database has one table, with one row for each unit of interest and columns containing all the relevant data about that unit. For example, cross-sectional survey data (each subject surveyed only once) can be stored in a single table, with one row per survey. Analysis is straightforward, since the data are already stored at the level of the survey respondent (the primary UA). Most research and business data, however, requires a much more complicated data form. Further, the optimal form for data storage and processing is rarely the form required for statistical analysis and reporting. See Appendix A for a brief discussion of OLTP and OLAP databases.

Relate each table to the primary UA

Database design models describe the relationships in terms of their direction and type (parent-child, one-to-many, etc.), but for data analysis I find it more useful to think in terms of how each table relates to the UA. Tables can either be at, above (lookup tables), or below (detail tables) the primary UA. For example, consider clinical trial data with the patient as the UA. Tables at the primary UA contain one and only one row per patient, e.g., demographic data. Data tables below the primary UA may contain multiple per patient (e.g., adverse effects table, dosage table, hiatus table). Data tables above the primary UA contains rows that apply to multiple patients, e.g., a table listing hospital characteristics. See figures 2 through 4 for examples at each level.

Unit of Analyis: Patient

```
   -------- Demographic Table --------
PatID  TrtFlg  Age  Female  HospID  Region
-----  ------  ---  ------  ------  ------
  001       1   32       1     H01      NE
  002       0   44       0     H01      NW
  003       1   27       0     H02      NE
  004       0   19       1     H02      SW
  005       1   26       1     H03      SW
```

Figure 2. Demographic table *at* the patient level. Note how this table would be joined to the tables in Figure 3 by the patient ID (patid) and to tables in Figure 4 by other relational keys.

Once you understand how each table relates to the patient, you are prepared to think about how you can create analysis variables at the patient level. Most of these analysis variables come from some level of summarization of detail data. The process of moving from the detail level to the patient level forces you to think about meaningful ways of reducing the data.

The paper discusses techniques for building unit-of-analysis driven data sets, so I will assume that the appropriate unit of analysis is known.

For example, in claims data a patient will have multiple diagnoses, representing multiple diagnoses per claim, over many claims. The only way to store the richness of all that detail at the patient level would be to create an array of diagnosis fields dx_1, dx_2, …, dx_n storing all the actual ICD-9-CM codes the patient was assigned over the entire time period. Not only would this take a huge amount of space, but it would not be meaningful. What you need to do is define questions that are relevant at the patient level, use the detailed diagnosis table to answer them, and store the answers at the patient level. The questions you ask will be study-specific, but will include questions like the following:

1. Did the patient ever have a diagnosis of asthma (study selection criterion)?
2. How many study patients had comorbid diagnoses of sinusitis (potential study confounds)?
3. How many asthma-related visits did the patient have during the course of a year (treatment vs. control patients)?
4. What is the average annual cost of asthma-related care (for treatment vs. control patients)?

Unit of Analyis: Patient

-------- Dosage Table --------

PatID	Date	Time	Dose (mg)
001	01Feb2002	8:00 AM	10
001	01Feb2001	8:00 PM	20
001	02Feb2002	8:00 AM	10
001	02Feb2001	8:00 PM	20

Example aggregations:

- Average daily dose
- Number of dose titrations

------- Adverse Effects Table -------

PatID	Date	Adverse Effect
005	01Feb2002	Nausea
037	02Feb2002	Sleepiness

Example aggregations:

- Adverse effect flag (Did patient experience one or more adverse effects-Yes/No)
- Days to first adverse effect

Figure 3. Tables *below* the patient level must be *aggregated up to the patient-level* for analysis.

Unit of Analyis: Patient

-------- Hospital Table --------

HospID	Name	Size	Type
H01	Univ D. Hosp	large	university
H02	Univ A. Hosp	medium	university
H03	County Gen	small	county
H04	SW Services	small	rural

Fields to denormalize (if required for analyses):

- Hospital size
- Hospital type

----------- Region Table -----------

Region	Wealth Index	Description
NE	103	ME, VT, NH, CT, MA, NY, PA
SW	90	AZ, NM, UT, NV

Fields to denormalize (if required for analyses):

- Regional wealth index

Figure 4. Tables *above* the patient level must be *denormalized to the patient-level* for analysis. Note that you should not denormalize descriptive or irrelevant fields (e.g., region description, hospital name).

DETERMINE WHICH QUESTIONS ARE ANSWERABLE

The database structure dictates which questions are and aren't even potentially answerable. Watch out for research questions that are not answerable with the given data. For example, a health care claim may have multiple diagnoses associated with it (as represented in a separate diagnosis table), but only one dollar amount associated with the claim. In this case, it is not possible to directly compute the average cost of claims for a specific diagnosis (Figure 5). In cases such as this, you need to think of other ways to frame the research question to make it answerable, or perhaps impute missing data.

- diagnosis-level table -

Claim Number	Diagnosis Number	ICD-9 code
000001	1	310.0
000001	2	433.0
000001	3	254.9
000002	1	98.2

Each claim can have multiple diagnoses.

--- claim-level table ---

Claim Number	Patient ID	Billed Charges
000001	123	$12,033.00
000002	456	$ 300.00
000003	789	$ 1,500.00

Billed charges available only at the claim level

Figure 5. Sample claims database including a diagnosis-level table and a claim-level table. Billed charges are not available at the diagnosis level of detail.

Keep database design issues in mind when you discuss analyses with project managers and clients, because they are not always aware of the data format and how that might affect their proposed analyses. For example, a project manager might ask you to run the correlation between disease severity and the number of days in hospital. If disease severity was measured on multiple occasions during a single hospital stay, then you need to summarize the disease severity data to the patient level before you can run a correlation. Depending on the clinical question, you might compute the average disease severity, the worst disease severity, or the number of days with severity reaching some clinically relevant cutoff.

AUDIT PRIMARY KEYS

Database schemas depend on keys to define the relationships among the tables. A key uniquely identifies each row in the table, either with a single column or with a combination of columns. Each table must have its own primary key[4], and may also contain foreign keys, or links to other tables. In Figure 2 above, the dosage table's primary key is the combination patid and date, while the demographic table's primary key is the single field patid.

[4] A *primary key* can be a single field or a combination of fields, but it must uniquely identify rows in a table.

Verifying the uniqueness of each table's primary key is a critical audit. Not only are you testing for database errors[5], but you are also testing your understanding of the database structure. For example, it might seem obvious that each patient in a clinical trial would have only one entry in the "completers" table (a table showing the study completion status, date of study completion, and status at study completion). However, if the clinical trial involves both a regular study period and an extended follow-up, patients participating in the extension period might have two entries in the completers table, representing their statuses for both phases of the study.

Verify uniqueness of primary keys

To test the primary key, create a temporary table that is unique by the primary key, and check the log to see if the sample size is the same as the original table. To create a dataset that is unique by a column, use the DISTINCT option in the SQL procedure or use the NODUPKEY option in the SORT procedure:

```
/* DISTINCT option in SQL procedure */
proc sql;
  create table work.uniqtest as
    select distinct patid, date
    from work.doses;
quit;

/* NODUPKEY option in SORT procedure */
proc sort nodupkey
          data=work.doses
          out=work.uniqtest(keep=patid date)[6]
          nodupkey;
  by patid date;
run;
```

The SORT procedure does tell you how many records were deleted (in the log file) but use it with caution. The NODUPKEY option permanently deletes the duplicate records from the data set, so always use it with the OUT= dataset option.

Find the duplicates by primary keys

If your table does have duplicates by the primary key, you will need to research them. Create a data set containing each instance of the duplicated key, using an embedded query in the SQL procedure, or the FIRST. and LAST. flags in the DATA step:

```
/* HAVING clause in embedded query */
proc sql;
  create table work.dups as
    select *
    from work.doses as a
    where patid in
      (select patid from work.doses as tmp
       group by patid, date
       having count(*)>1 and date=a.date)
    order by patid, date;
quit;

/* FIRST. and LAST. flags in DATA step */
proc sort data=work.doses;
  by patid date;
run;
data work.dups;
  set work.doses;
  by patid date;
  if sum(first.date, last.date) < 2;
run;
```

Both duplicated and missing primary key values can cause data problems. You must eliminate duplicated key values, to avoid Cartesian products and other join problems. When key values are missing, you might consider imputing the missing data.

BUILD TABLES AT THE PRIMARY UA

The end-goal for reporting and analysis is to have a table that is unique by the primary UA. Statistical analyses require one row per observation unit, and most reports templates require the same (particularly if the reports include standard deviations or standard errors). Keep this goal in mind and work backwards from there to create analysis variables from the raw data tables.

Denormalize lookup tables as needed

Some analytical variables are formed by straight denormalization, i.e., a simple merge with a lookup table. Use lookup tables to denormalize columns that will be required for reporting and analysis. For example, in a clinical trial conducted at multiple hospitals, hospital characteristics would be stored in a lookup table, not on every patient record. If you have a regression model predicting inpatient charges and you wish to control for hospital size and hospital type, you will need to bring those columns in to your patient-level table (Table 2).

Table 2. Sample patient data with denormalized hospital characteristics.

PatID	Age	Female	HospID	HospSize	HospType
0001	32	1	H01	large	university
0002	44	0	H01	large	university
0003	27	0	H02	medium	university
0004	19	1	H02	medium	university
0005	26	1	H03	small	county
0006	30	1	H04	small	county
0007	24	0	H05	small	county

There are two cautions regarding denormalization. First, you must ensure that the lookup table does not have duplicates by your merge key. Failure to spot duplicate entries in a lookup table can result in duplicated patient records if you join tables with the SQL procedure. (This type of join, which produces duplicated records, is known as a *Cartesian product*.) Duplication will not occur if you use a MERGE statement in the DATA step, but you still run the risk of having a hospital coded differently for different patients. Second, you must watch for missing data in your lookup table. Depending on how you join the tables, you may lose patient records if they contain missing/invalid hospital identifiers (e.g., if you use the default inner join in the SQL procedure). Watch for both of these errors by checking your log file against your actual sample size (the number of unique patients).

Create flags (dummy variables) from detail data

Often the only level of aggregation needed is a flag, a yes/no field indicating the presence or absence of an event. Store these values in standard boolean notation, numeric 1 for yes and 0 for no, rather than character values "Y" and "N." This allows you to use the flag variables directly as dummy variables in statistical models.

[5] Database software generally enforces the uniqueness of primary keys, but you should never underestimate the possibility for bad things happening to data tables before they reach you. Also, as the SAS System is not a relational database application, it will not enforce the referential integrity among tables once they are read in.

[6] Using the NODUPKEY option can get you into trouble later unless you are careful, because it picks one complete instance of any duplicate records, including columns that vary across records. To avoid future data errors and debugging woes, add a KEEP statement whenever you sort with the NODUPKEY option. Keep only the fields used on your BY statement, to prevent yourself from using other columns that may be meaningless when randomly selected at the patient level.

There are numerous ways to create flags in the primary UA table. A straightforward way is to create the flags at the detail level, then use the MAX option on a MEANS procedure to summarize the flag by patient. Use the OUT= option on the OUTPUT statement to create a table containing the results by patient.

Aggregate other detail data into analysis variables

If you spend the time determining which questions are answerable, you should have a good idea how you are going to build your analysis variables from the detail data. For many variables you merely need to capture the average or total of some detail-level column (e.g., total costs of concomitant medications per patient, average daily dosage of study medication). As with flags, you can create these columns in a straightforward way using a MEANS procedure[7], as in the example below.

```
/* from work.dosage (dose data stored */
/* at one row/patient/day), use proc  */
/* means to compute each patient's    */
/* average daily dose, and store the  */
/* result in work.dosept table.       */
proc means data=work.dosage;
  var daydose;
  by patid;
  output out=work.dosept
        mean=avgdose;
run;
```

Use interim-UA tables as needed

As mentioned above, some cases require an interim-UA table at a different level of analysis. A good example of this is dose escalations. Suppose that in a clinical trial, patients are titrated on an analgesic until they achieve an acceptable level of pain control. One potential outcome of interest is dose escalations (assume that daily dosage data are available in their own table). You can identify and count dose escalations in the dosage table; however, you will need to report escalations at the patient level, e.g., the average number per patient. Working with the dosage table alone you could count the total number of escalations and divide by the sample size (similar to the example outlined in the introduction).

If instead you create a table at the patient-escalation level (one row representing each escalation for each patient), you can then compute other simple statistics at the patient level to describe dosage escalations: the standard deviation, the maximum number of escalations per patient, the number of patients who required no dose escalations, etc.

CONCLUSION

Source data rarely come in the ideal structure for the analytical programmer. To create data for reporting and statistical analysis, you need to understand the raw data structure, then rebuild it at the level of your UA. At first, the need to aggregate detail data to the patient level might seem like a database issue, but it is truly a methodological question. Using a UA-oriented approach forces you to make these methodological issues explicit in your programming.

APPENDIX A. OLTP VS. OLAP

Online transaction processing (OLTP) refers to the process of storing and processing data for the purpose of administering business operations. For example, an insurance company's claims processing system both stores data and performs functions based on data input (e.g., approving claims, processing payments, sending rejection letters). Online Analytical Processing (OLAP) refers to statistical and business analyses performed on existing data. The requirements of OLTP vs. OLAP dictate very different database structures.

OLTP requires fast processing and efficient storage. These are achieved with a normalized database schema, i.e., a design in which repetitive data are split out into separate tables. For example, clinical trial data may track the actual medication doses patients take each day. These can be stored in a table with one row per patient per day, but there is no need to replicate all the patient's baseline demographic information (which never changes) on each row of this table. A normalized design removes the demographic data from the dosage rows and creates a separate table containing the demographic data for each patient. The two tables are joined by a unique patient identifier (Figure 6).

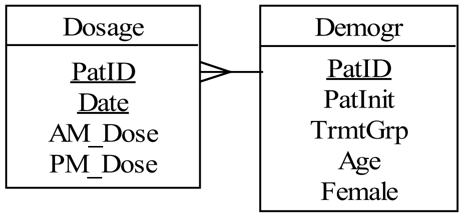

Figure 6. Daily medication data normalized into a dosage table and demographic table. Primary keys are underlined.

Data warehouses and archival systems also tend to use normalized database schemas, as they store the most detail using the least amount of disk space. Unless you are working with survey data, it is likely that your source data are at least partially normalized.

OLAP requires meaningful, relevant data. To run a logistic regression model on patient morbidity, you need a table with one row per patient, containing columns for each of the dependent and independent variables. The database programmer's job is to create these analysis variables from the raw, normalized data. This step often takes more time than the actual analysis, and requires you to make a number of (often subtle) methodological decisions. Focusing on the primary UA facilitates this process.

REFERENCES

Academic Computing and Instructional Technology Services, University of Texas at Austin. (2000). Introduction to Data Modeling. ACITS on the Computing Web.

Klein, D.F. (1998). Listening to meta-analysis but hearing bias. *Prevention and Treatment*, *1*, Article 0006c.

ACKNOWLEDGMENTS

The author thanks Fidelity Investments® for its support in attending this conference, and Alexis Hayden for her review and comments.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Please contact the author with any feedback, or for copies of the presentation handouts, which contain in-depth examples.

Vanessa Hayden
Fidelity Investments
82 Devonshire Street, R3D
Boston, MA 02109

[7] You can also aggregate detail data in a DATA step using the RETAIN statement, but that strategy is inadvisable unless you absolutely require the efficiency. Look for upcoming writing by the author on this topic.

Work Phone:	(617) 563-3178
Email:	vanessa.hayden@fmr.com