Paper 49

Relational Database Schemes and SAS® Software SQL Solutions

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

ABSTRACT

A logical data model should do more than help a DBA protect the integrity of a database; it should also make life easier for programmers. A scheme such as that displayed in a simple E-R diagram can serve as a map that a programmer can use to navigate through a database. Primary and foreign keys can represent different entities and their interactions in the real world. Integrity constraints built into a database can guarantee that normal updates will not invalidate a correct program.

This presentation looks at database design from the point of view of a programmer. It begins with examples of alternative database schemes and how they can make programming harder or easier. Following a brief overview of the concepts of relations and dimensions, the emphasis shifts to more complex database schemes and methods. Examples include data-based links among locations, persons, laboratory specimens, and clinical events, and among vendors, clients, and sales. The presentation concludes with illustrations of the power of relational database design and Structured Query Language (SQL) working together on a SAS server.

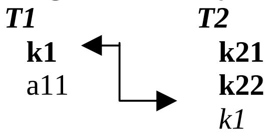
INTRODUCTION

SUGI brings together such a diversity of talents in statistics, computer science, mathematics, operations research, genetics, and other fields that the *ad hoc* programs SAS® programmers customize to content and to computing environment often work as well or better than standard database programming methods. Even so, as database technologies encompass a wider range of problems, standard methods deserve a close look.

Why Relational Database Schemes?

An Entity-Relationships diagram (*e.g.* Diagram 1) of a database scheme gives a programmer a map to use to navigate around a database.

Diagram 1: Entity-Relationships



A database *scheme* (aka *schema*) offers a wealth of information to those who bother to appreciate the cues.

Primary keys (***T1*.k1**, ***T2*.(k21+k22**) and *foreign* keys (***T2*.k1**) represent entities and their interactions in the real world. Primary keys have to have distinct values in each row of a table, and each value in a foreign key has to have that same value in the corresponding primary key column. These two constraints, *key integrity* and *referential integrity*, show a database programmer how to specify a solution to common programming problems. For example, rename ***T1*=*Subjects***, **k1=subjctID**, **a11=caseYN**, ***T2*=*NJnames***, **k21=name**, and **k22=address**, and the SAS SQL solution to the problem of creating a mailing list of NJ cases becomes obvious:

```
CREATE TABLE NJcases AS
SELECT T2.name AS name,T2.address AS address
FROM Subjects AS T1,NJnames AS T2
WHERE T1.subjctID=T2.subjctID and T1.caseYN='Y'
;
```

From the scheme structure and constraints the programmer knows

- each name+address has a different value and corresponds to a unique **subjctID** in ***Subjects***;
- the names and addresses belong to subjects;
- a subject has either one or no name-address on the NJnames table (the connector ◀──▶

means that a distinct table cell (column-row) value in ***Subjects*** matches at most one cell value in ***Njnames,*** as in

Subjects.subjctID		NJnames.subjctID
01101 $\longleftarrow$		
	$\longrightarrow$	01101

- each subjctID in ***NJnames*** has a value in its related caseYN in ***Subjects***.

Equally important, the programmer does not have to know anything about
- the ordering of columns in the tables;
- the ordering of rows in the tables, or indexes;
- which name and address columns to select;
- whether or not a subject has a name+address on ***NJnames*** or another table.

Further, the uniqueness constraint on the primary key name in the ***NJnames*** table presents a more subtle cue. It tells the programmer that the mailing list for the state of New Jersey contains no duplicate combinations of names and addresses.

An alternative to the relational scheme in Diagram 1, the unconstrained scheme

SUBJECTS
subjctID
name
address
caseYN

seems simpler at first glance but leaves out some of the cues that a programmer might need to know. It does not distinguish the subject dimension from the name-address dimension. As a result, the programmer has to guess whether subjctID has unique values, in which case a subject can have only one name-address, or duplicate values, so that different addresses for the same subject become possible but not formally distinguishable from two rows with the same subjctID, name, and address.

Relational design has during the last two decades evolved into an international standard for database systems.[1] A relational design presupposes a logical mapping among key variables. From the viewpoint of the user a data model defines links through queries into the database. This abstract view of the database remains constant across many different physical implementations. SQL programs operate only within the logical layer.

The logic behind relational database models proves simple and elegant. Any mapping between two sets has three main relations between sets of key values. One we have seen earlier:

1 $\longleftrightarrow$ 1 (one-to-one).

Two others complete the set.

1 $\leftarrow\!\!\twoheadrightarrow$ m (one-to-many)

denotes a functional dependency from possibly more than one distinct cell values in one table to only one cell value in another table. Recall that a function {a,b} has unique values of a paired with possibly repeating values of b; in other words, given the value of b, one knows the value of a in {a,b}. A strict hierarchy or *tree* has a 1-m relation between a parent node and each child node (mixed metaphors intended);

m $\twoheadleftarrow\!\!\twoheadrightarrow$ m (many-to-many)

denotes no particular constraint on the values (a,b); if every possible combination of {a,b} can appear as key values, a m-m relation provides little information for the programmer. The fewer the distinct combinations of {a,b} that appear in a m-m relation, the more information it provides.

Date and Darwen[2] highlight a feature of a relational data model. A logical proposition in the form of a *query* yields a “relational variable” or *relvar*. Date and Darwen coin the term *relvar* to serve as a handle for properties of table references in queries, including maximum dimensions, or in simple terms, whether data fit in a *cell*, *column*, or *table*. An inner join or equijoin (in SQL), SELECT * FROM A,B WHERE A.a=B.b; where A.a and B.b have a 1-1 relation, pairs each A.a with a single B.b. It produces a value that fits in a cell. The same query on a 1-m relation may produce a value that fits in a column but not a cell. On a m-m relation, it multiplies {a,b} pairs so that repeating sets of values of B.b fit only in a table and not in a column or cell.

It obviously helps to know in advance the maximum dimensions of a relevar produced by a a query on a scheme . A query that produces more than one age per person does a lousy job of calculating an average age. On the other hand, a query that can only produce one value per person would not produce an accurate list of the names of the children of some parents.

Programmer Camps

Programmers often take pride in being able to work around ambiguities in a database scheme. But why not start with a better scheme in the first place? This question divides database programmers into two camps. The "cueless" camp takes whatever structures and schemes data sources might have as given, and writes programs that combine data and transform them directly into solutions. The "data modellers" camp transforms data into meaningful schemes, and take advantage of the knowledge built into the schemes when specifying solutions.

Data modellers spend more time on formal database design, and they prefer the so-called logical or declarative programming languages. Most program in SQL. Though not a perfect language for declarative programming and limited by implementations within software systems and platforms, SQL represents a major departure for database programming.

Why SQL?

At SUGI 22 I presented *Ten Good Reasons to Learn SAS System PROC SQL*[3]. At SUGI 23 Paul Lafler raised the ante with *Ten Great Reasons to Learn SAS...SQL*[4]. For SUGI 24 I had planned to present *The Ten Best Reasons to Learn SAS SQL*. Fortunately, our session Chairpersons insisted that we call off the battle of superlatives. I changed the title, but have maintained much of the enthusiasm that led Paul and me to champion SI's implementation of the Structured Query Language[5].

The Ten Good Reasons … presentation emphasized the formal design of SQL. The developers of SQL gave it a concise, logical, and intuitive structure. A few basic forms hold open lists of table definitions, column names, and expressions containing column names. Basic forms such CREATE TABLE <table-name> AS SELECT <column-name1>[,<column-name2>…] FROM <table-def1>[,… leave little room for doubt about the placements of keywords and variable names. A few basic forms cover the essential set-relation operations on key values (by variables in SAS data steps) that one needs to declare (specify) a SQL solution to a wide variety of practical database programming problems:

Select and Project
```
SELECT <>[,<>,<>] FROM S1 WHERE <>
```

Intersection S1XS2
```
SELECT <> FROM S1 AS t1,S2 AS t2 WHERE
t1.<>=t2.<>
```

Difference S1–S2
```
SELECT <> FROM <>
WHERE <x> NOT IN
        (SELECT DISTINCT <x> FROM <>)
```

Union S1+S2
```
SELECT <> FROM     (SELECT <> FROM <>
          UNION CORR
        SELECT <> FROM <>)
```

A programmer has to learn more SQL features to become fully proficient in the language. Nonetheless, the basic forms in the space of the preceding paragraph will handle a wide variety of practical programming tasks. Further, SQL has become an ANSI standard and the prevailing lingo of database servers everywhere.

Logical languages such as SQL combine with logical database schemes to detach the data access language from the physical implementation of a software system. That means that SAS SQL looks much like other flavors of SQL, and that it allows various independent and efficient implementations under MVS, Unix, MS Windows/NT, and other operating system.

A Nagging Question

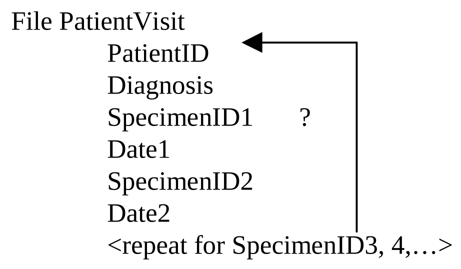
In light of its practical advantages, I have long speculated about the reasons why SAS SQL has not largely replaced the older SAS data-step language. (And for the sake of SAS programmers everywhere the transition cannot happen too soon: SI's SQL Guy, Paul Kent, has taken on a larger role in SAS upgrades, but is reportedly holding back new SAS features until he is sure that his monster has taken over the village.) In fact SAS SQL is steadily gaining popularity among mainstream SAS programmers, but at a slower pace than some of us expected. Why would anyone continue to use a MVS vestigial data step program instead of a structured query?

Although a Bit Slow on the Uptake, It Gradually Dawned on Me That….

Some database schemes do not accommodate themselves to SQL solutions. Some of the questions posted to the SAS listserver, SAS-L, do not fit all that well into any of the basic SQL forms. Perplexed programmers often struggle with data sources organized in a way that bear little resemblance to the relational model that SQL programmers would find, for example, on an RDBMS server. Instead, data structures appear to be modelled after paper forms, tape files, print files, or (gasp) even IBM cards. Without the logical layer between program and the physical implementation that a relational model provides, SQL does not work nearly as well if at all.

Data Models and Database Schemes that Confound SQL

To see why a SQL programmer (almost certainly in the "data modeller" camp) feels more comfortable working within the framework of a relational data model, it may help to take a look at other database schemes and models that do not fit SQL constructs. For example, consider this highly simplified, not-all-that-hypothetical data structure that someone (almost certainly in the "cueless" camp) apparently copied off a log sheet at a clinic:

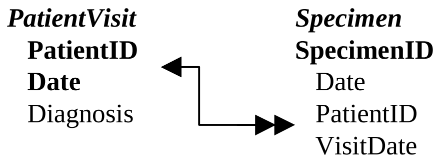


This crude diagram suggests that all of the SpecimenID's and Date's in one record belong to the person identified by the PatientID. Also true, but not easy to show, the second, third, etc. SpecimenID represents the first follow-up specimen, second, etc. related to each visit, and the order of the records in the file represents the order of visits.

This data model gives a SQL programmer nightmares. SQL does not recognize the order in which records appear in a physical file or dataset. On purpose. If it did, the SQL programmer would have to pay attention to the various ways different platforms and devices store data on physical media. SQL shields a programmer from the details of an implementation, but also prevents him or her from varying operations on a row of data depending on the set of values that precedes or follows it, or when it happens to be the first or last record in a physical file. Further, SQL does not offer variable arrays for repeating groups of SpecimenID's and Date's, nor any particularly efficient means of determining the number of specimens related to patient visit.

A SQL programmer cannot, for example, select the 2nd visit by a patient. A data step programmer handles this problem easily. The data step programmer can set up an array of variable names and search for the last non-empty specimenID in a record. The SQL programmer has to write a series of CASE .. WHEN conditions.

How would a relational database scheme help the SQL programmer specifically, and perhaps in general simplify programming? First, it would require an explicit ordering of records of patient visits. Also, it would represent the dependency of specimenID's on patientID's as a relation between a patient and a specimen table and a constraint on the columns values that link the two.



Multiple specimen records ordered by Date link back to patient visits uniquely identified by PatientID and Date. A referential integrity constraint protects this link by preventing changes in the values of PatientID and Date in the PatientVisit table while the corresponding values exist in the PatientID and VisitDate columns of the Specimen table.

Relational Databases: Where SQL Rules

The SQL compiler frees the programmer from having to keep track of row counts, field pointers, or the order or physical location of data items in rows. To realize these benefits, a programmer has to store data properly for the program, just as a violinist has to have strings in proper order and tuned before a programmed series of rapid movements of the bow produces a *Flight of the Bumblebee*.

Data must appear in the form of tables with fixed column definitions but unlimited numbers of rows. Each column should contain only simple and primary types of data. Beyond all else, classifiers used to select a subset of rows in a table, identifiers used to link a row in one table to a row in another table, and information used to group or write data in a specific order must appear as column values in tables. In a relational, data-based system (to use the original description), the system only has to maintain pointers to tables, columns and rows. The database architect embeds all other identifying and classifying information in the columns of tables.

Any column in a data table can serve as an identifying (key) or classifying variable. Each table has to have a primary key consisting of one or more columns. This key has to distinguish any one row from any other row in that table. The database system has to block any attempt to insert or update a row and duplicate a primary key value. This precaution guarantees the ability of the system to link unambiguously to a single row in a data table. Tables may also contain columns that link to columns in other tables. These links have special meanings in relational systems. For example, a table of patient ID's may contain a Physician ID column, and may use the values in this column to link to the primary key in a Physician table. If included in the overall database design to express a many-to-one relation between Patients and Physicians, the so-called "foreign key" reference requires the special protection of a referential integrity constraint. An explicit constraint on changes in data protects against deletions or changes in a Physician ID value in the Physician table that might inadvertently sever or change the relation between patient and physician.

Where a data model accurately reflects the reality it is trying to represent, and where a programmer understands and appreciates the data model, SQL rules. The SQL compiler reduces a programming task to the simpler task of declaring a logical solution in terms of elements of a data model. Say a programming task requires that we find all of the other patients of the physician of a given patient. First, we must find the unique identifier for the physician of a specific patient. Then we must use the unique physician identifier to find all rows in the Patient table that contain that unique physician identifier as a reference or "foreign key", excluding the patient ID that we already know. The SQL solution

```
SELECT *
FROM (SELECT *
          FROM Patient
          WHERE PhysID=(SELECT
                              DISTINCT PhysID
                              FROM Patient
                              WHERE UPCASE(PatID)
                              =UPCASE("&PatID")
                              )
     )
     WHERE UPCASE(PatID) ne
               UPCASE("&PatID")
     ;
```

has a "nested query" that initially selects the PhysicianID from the row in the Patient table that contains the unique primary key value (say &PatID="JonesB1"). It then selects all of the patient names from rows containing that PhysicianID.

Why not use the patient's name instead of the PatientID? SQL would have no difficulty with a program that substitutes the patient's last name for PatientID provided that the name is unique in the PatientLstName column of the Patient table. If not the solution will fail with a run-time error (see discussion of *relvar* forms above). A proper data model guarantees that the SQL solution based on a PatientID will succeed. The data model complements the SQL program perfectly.

Perhaps better alternatives would allow users to select on names but guard against multiple matches and entry errors. For example,

```
SELECT DISTINCT COMPRESS
          (CASE (SELECT COUNT(*)
                    FROM Patient
                    WHERE UPCASE(PhysID)
                              =UPCASE("&PhysID")
                    GROUP BY PhysID
                    )
                    WHEN (1)   THEN "              "
                    WHEN (0)   THEN
                    "None: Check name."
                              ELSE      "Many: Use PatID  "
               END) AS Note,PatID
FROM Patient
WHERE UPCASE(PhysID)=UPCASE("&PhysID")
;
```

More on Dimensions

A typical trap for programmers gets set when a database developer, intentionally or not, tries to fold more than two dimensions into the two dimensions that fit on a computer screen or a printed report. It can be done, and in fact most database work requires it, but it has to be done carefully and thoroughly. Database developers may choose to or inadvertently omit dimensions and assume that programmers understand the context of data collection and can reconstruct the dimensions from related data or from sequences of data elements.

The worst examples seem inspired somehow by data entry screens or reports. (Some appear to believe that one should always store data either as collected or as displayed.) For example, a data warehouse may have data stored as tabulated from monthly sales records:

MonthlySales98

Retailer
ZestCola
BoltCola
DoltCola
VoltCola

How would a programmer calculate the percentage increase or decrease per month?

After a quick look at the data stored on ***MonthlySales98***, a clever programmer might figure out that the *Cola columns hold a month of sales for the retailer named in the first column. Since "monthly" implies a specific ordering of twelve rows per Retailer, the programmer could infer that the physical ordering of rows of data by Retailer follow a January .. December sequence. Provided that sorting by Retailer preserves this implicit ordering, it proves fairly easy in a SAS data step to SET ***MonthlySales98*** BY Retailer; and IF FIRST.Retailer THEN month=1; ELSE month+1;. With a month variable in place, it becomes fairly straightforward to RETAIN prior month's sales and calculate a percentage increase or decrease.

Almost anyone looking at this example would argue that it makes sense to put a month column in the data table. Besides saving the trouble involved in making explicit an implicit sequence, the month column would help control the quality of reporting, Say a Retailer has only eight rows of sales data. Does that imply January through August, May through December, or some other sequence? Assuming a continuing sequence of months, the monthly increases or decreases do not vary, but the comparisons with other Retailers does not hold.

The same principle applies in less obvious situations. What if the Retailer column contains this subset of cells:

Barts
Safeway
7-11
CSmiths

While easy to check that each Retailer has not more than twelve rows of sales figures in ***MonthlySales98***, to guard against ambiguous classifications, nothing in this context precludes confusion of values of Retailer in different years. "Safeway" in 1997 might refer to a different location than in 1998, or to a group of stores, or to the whole chain. A data model that links Retailer to ***Companies***.company,

Companies ***MonthlySales98***
location ⟷ **retailer**
company etc.

makes it clear that ***MonthlySales98*** contains a specific store identifier, and that **retailer** links to ***Companies*.location** (note that linked keys do not have to have the same name).

While a database developer cannot make all implicit information explicit in data, he or she should separate each important dimension into its own separate column, and should strive to populate each column with distinct class values. This prescription leads us in the direction of very simple table structures in which most columns combine in a composite primary key representing dimensions. Each set of key dimension values, necessarily unique, classifies a small set, perhaps just one, value.

The simple tracking model,

Events
siteID
personID
event_type
date
outcome

constrains each date and outcome to a specific combination of site, person, and type of event. A further constraint on person might limit any one person to a single site. This simple model and its table structure not only proves easy to understand and implement, it also holds a surprisingly rich set of information. Even better, it proves robust as well, in that inserting new values in the column **event_type** makes it capable of tracking new or different events. A programmer can expand the capabilities of the database system simply by expanding the domain of **event_type** but without having to restructure the logical or physical layers of the system.

I first proposed simple "vertical" data models of this type over a decade ago, primarily because I found it much easier to export data from vertical structures with few columns than from horizontal "spreadsheet" structures with many columns. More experienced programmers argued against them on grounds of efficiency. They pointed out the fact that, instead of having one simple key and many attribute fields per record, a vertical structure has many partially-repeating key values and few attribute fields per record. A vertical file has to have ten or more times the number of records to hold the same amount of data!

Whether compelling or not in its day, that argument has little persuasive value in this age of powerful database servers, cheap disk storage, and logical and declarative programming languages. From the mid-range of database sizes and below that, access to fields on different data records, linked by key values, works about as fast as access to fields within a physical record. Further, a programmer may actually find it easier to compare and combine values on different records.

Two program segments illustrate some of the power of simple relational database design and a logical and declarative programming language (SQL in this case):

To determine which subjects have consent for further testing after an initial diagnosis (an inter-record, reflexive query):

```
SELECT t2.site,t2.person
FROM Events AS t1,Events AS t2
WHERE t1.site=t2.site
    AND t1.person=t2.person
    AND t2.date>t1.date
    AND t1.event_type=”<diagnosis>”
    AND t2.event_type=”<consent>”
;
```

To report whether customers who have a payment due have a payment date this month, and if so the date, or do not have a payment date recorded:

```
SELECT t1.site,t1.person,
  case when t2.date ne .
      then put(t2.date,mmddyy10.)
      else "No Payment"
  end  as PayDate format=$char10.
FROM
          (SELECT * FROM Events
          WHERE event_type="d") AS t1
          LEFT JOIN
          (SELECT * FROM Events
          WHERE event_type="p") AS t2
ON t1.Site=t2.Site AND t1.person=t2.person
AND t2.date>t1.date
;
```

Notice the absence of arrays, loops, and other features of procedural programs. I recall Harlan Mills, one of the original software engineers, telling a small group “ …arrays are dangerous …” At the time I did not understand the significance and profound wisdom behind this remark. Since then I have come to understand it all too well. Once an application programmer reaches down into the physical layer of a database system and begins defining special data containers and pointers to them (e.g. arrays), he or she becomes responsible for maintaining and disposing of them. I appreciate more and more each year Harlan’s insistence that simple, proven logical models, consistently applied to divide and conquer complex problems, offer the best hope for successful programming. This strategy underlies the best features of relational design and object-oriented programming.

Relational database design and SQL programming operate almost independently. It does not take too much effort to protect the integrity of a relational database of SAS datasets. In Version 7 the SAS® System offers integrity constraints in PROC SQL. These new features make it even easier to design and administer databases that support logical, declarative programming.

CONCLUSIONS

Relational data models and good logical schemes for a database set the stage for programming in a logical, declarative mode. A relational design for a database gives programmers a roadmap to help them evade traps, focus on the simple logic of sets and relations, and resolve potential ambiguities. SQL combines with good design and powerful database servers to encourage quick, simple, and precise programming.

TRADEMARKS

ACKNOWLEDGEMENTS

Mike Rhoads, Robin McEntire, Kellar Wilson, Stella Wang, Marianne Whitlock, Steven Schweinfurth, and Ian Whitlock contributed suggestions and comments that led to substantial improvements in content. Greg Barnes Nelson encouraged the author to develop the topic and present it.

AUTHOR CONTACT

Sigurd W. Hermansen
Westat, An Employee-Owned Research Co.
1650 Research Blvd.
Rockville, MD 20850 USA
(301) 530-6851
FAX: (310) 315-5936
e-mail: hermans1@westat.com

REFERENCES

[1] Maier, David, (1983) **The Theory of Relational Databases**, Computer Science Press, p. 94.

[2] Date, CJ and H. Darwen, (1998) **Foundation for Object/Relational Databases: The Third Manifesto**, Add-Wes.

[3] Hermansen, S.W., (1997) *Ten Good Reasons to Learn SAS Software's SQL Procedure*, Paper 35, **SUGI22 Proceedings**

[4] Lafler, Kirk P., (1998) *Ten Great Reasons to Learn SAS Software's SQL Procedures*, Paper 131, **SUGI23 Proceedings**.

[5] SAS Institute, Inc. (1989) **SAS Guide to the SQL Procedure, First Edition**, Cary,NC, SAS Institute, Inc.