

A Relational Database Primer for SAS® Programmers

Paul Dalberth, Family Health International

ABSTRACT

Information Technology professionals still dispute the need for a relational database management system (RDBMS). The prevalence of these environments as repositories of crucial enterprise data, though, means that familiarity with them is often a prerequisite for successful SAS-based system development. This paper includes discussion of how a relational database differs from a collection of SAS datasets and defines some common Relational Database Management System terminology. Also included is a summary of the theory of data normalization at a sufficient level of rigor to engender conversational familiarity with this practice. From this conceptual foundation the discussion turns to a selective survey of methods for accessing relational structures from within the SAS system. These methods include SAS/ACCESS® access and view descriptors and the DBLOAD procedure, the SQL pass-through facility, and host-based relational database access within the client-server architecture provided by SAS/CONNECT®.

INTRODUCTION

Built-in functionality is what distinguishes a database management system from a simple collection of files. The former accommodates mechanisms for expressing the interrelated nature of data as well as facilities for concurrency control, data integrity, transaction management, security, and backup and recovery. If users desire these qualities from a collection of files, then programmers must provide it. Contemporary DBMS products also include a nonprocedural language called Structured Query Language (SQL) to allow users to define data structures and manipulate data. Intimate knowledge of the physical structures underlying the data, such as linked or inverted lists, is not necessary. Proponents of DBMSs cite all of this power as a major advantage. Detractors argue that these facilities are expensive, both in initial acquisition fees and in hardware upgrades that DBMS products may necessitate. Both points of view are valid, but relational database systems are the dominant vehicle for storing business information so ultimately the discussion is rhetorical. An interesting question, included here as a sidebar, inquires after the location of the SAS System in the preceding spectrum. Users can index SAS datasets, can include password protection, and can use SAS datasets concurrently via the SAS/SHARE® product. Enforcement of referential integrity, crash recovery, and transaction control, however, requires programming effort. Consequently, SAS datasets do not comprise a true DBMS, but they are somewhat more than a collection of 'flat' files.

The code and structured query language (SQL) examples should run on all platforms that support the SAS System

so the following is included for informational purposes. All code samples are written for the SAS System version 6.08 under VAX/VMS. All database examples are for Digital Equipment Corporation's Rdb version 4.2.

RELATIONAL TERMINOLOGY

DBMS software organizes data into tables, where observations are called *rows* or *tuples* and variables are known as *attributes*. A set of attributes that allows programs and users to uniquely identify any given row is called a *superkey*. If the superkey is not reducible to a smaller set of attributes while still preserving uniqueness, it is called a *candidate key*. *Indexes* are database structures that expedite table searches and allow random access. The candidate key that comprises the table's main index is called the *primary key*. An attribute of table 1 that serves as the primary key in table 2 is called a *foreign key*. For example, consider these two tables that store SUGI speaker information:

table 1: (name, papertitle, papertype)

table 2: (papertype, time)

The attribute papertype is the primary key of table 2. The primary key of table 1 is a composite key consisting of the attributes name and papertitle. The underlines above indicate those attributes comprising the primary key. The attribute papertype in table 1 is a foreign key since it is the primary key in table 2.

Normalization Terminology

The process of removing attributes from one table and creating another table to store them is called *projection*. *Normalization* is projection driven by specific criteria. A *functional dependency* in a table occurs when the value of one attribute uniquely determines the value of another. In table 1, above, papertype is functionally dependent on name and papertitle so we write:

name, papertitle -> papertype

Functional dependencies will become important in the next section. Normalization attempts to prevent data operation *anomalies*. These occur when data is lost unintentionally during deletions (delete anomalies) or when extra rows must be inserted to maintain data integrity during insertions or modifications (insert and update anomalies).

NORMALIZATION

Consider the following table containing SUGI speaker information (see appendix figure 1):

(name, papertitle, degree1, degree2, degree3, company, papertype, maxtime, section)

The primary key is comprised of the name and papertitle attributes. This table is in what relational theorists disdainfully refer to as 'flat file' format. The table contains a repeating group of degree attributes to store each participant's college and graduate school degrees. We hope that three positions is enough to store all of the degrees for the most educated speaker. If not, we would need to add more attributes. Also, if we wish produce a listing of participants and the number of degrees they hold, we would need to count the number of non null degree attributes in each row. This is not particularly difficult with array processing in a third generation language or in the SAS System:

```
array degree degree1-degree3;
count = 0;
do over degree;
  if degree ^= '' then count = count + 1;
end;
```

But if we remove the repeating group so that the table looks like this (see appendix figure 2):

(name, papertitle, degree, company, papertype, maxtime, section)

then the non-procedural alternative is conceptually simpler and saves a few lines of code:

```
proc freq data = <dataset name>;
tables name;
```

Of course, this simplicity has come at the expense of redundant data in the table, since we now have one row for each degree that each speaker holds.

Definition: A table is in first normal form if and only if it contains no repeating groups.

This form of the table has insertion anomalies since if we want to add another paper for Joe we have to add two rows, one for each degree that he holds. It has modification anomalies since if Joe's employer changes we must modify four rows in the table. The problem is that the table contains attributes that do not depend on all of the primary key (name and papertitle). Specifically, attributes company and degree depend only on the value of name. These attributes should be moved, or projected, to two other tables. Now we have three tables (see appendix figure 3):

speaker (name, papertitle, papertype, maxtime, section)
company (name, company)

degree (name, degree)

Definition: A table is in second normal form if and only if the values of all non-key attributes depend on the value of the entire primary key.

The speaker table still has a modification anomaly, though: if the maximum time for an invited paper changes then we have to modify all of the rows that store data for invited papers. The table also has deletion anomalies: if we were to delete the last invited paper we would lose the fact that an invited paper is allotted 50 minutes. The table contains a functional dependency (papertype -> maxtime) where the determinant is not a superkey and the right side of the dependency is not a member of a candidate key. Database practitioners also refer to this as a transitive dependency since these two functional dependencies hold:

name, papertitle -> papertype
papertype -> maxtime

To eliminate the modification and deletion anomalies we must remove the transitive dependency via projection.

Definition: A table is in third normal form if and only if it is in second normal form and for all non-trivial functional dependencies of the form A -> B one of two conditions hold: either A is a superkey or B is a member of a candidate key.

Note: a trivial functional dependency is of the form: XA->A.

The projection to third normal form will leave us with four tables (see appendix figure 4):

speaker (name, papertitle, papertype, section)
company (name, company)
degree (name, degree)
maxtime (papertype, maxtime)

A more informal, if somewhat cryptic, summary of third normal form is this: a table is in third normal form if every non-key attribute depends on the key, the whole key, and nothing but the key.

Third normal form is sufficient for many database applications, but even third normal form does not guarantee immunity from all anomalies. Consider a table that looks like this:

(name, section, sectionchair)

Suppose that this table is subject to these constraints: one section can have more than one section chair, each speaker is only advised by one chair from each section, and a person can only serve as sectionchair for one section. The table contains no repeating groups so it is in first normal form. the only nonkey attribute is sectionchair and it depends on the whole key so the table

satisfies the conditions for second normal form as well. The only transitive dependency is this one:

name, section -> sectionchair
sectionchair -> section

This transitive dependency is allowed since section is a member of a candidate key so the table is in third normal form. Unfortunately, there are still insert anomalies since if we need to add a new section chair we cannot until a speaker is assigned to one. Also, if we remove a speaker who is the only assignee to a particular chair, we lose the data indicating that the section chair is assigned to a particular section.

Definition: A table is in Boyce-Codd normal form if and only if for every non-trivial functional dependency of the form $A \rightarrow B$ attribute A is a superkey.

Application of this principal would result in the projection of our original table into these two tables:

(name, section, sectionchair)
(section, sectionchair)

Normalization through BCNF prevents all anomalies arising due to functional dependencies and is sufficient for most database applications. There are two other normal forms (fourth and fifth normal forms) that deal with other constraints (multivalued and join dependencies) and are not discussed here.

READING FROM AND WRITING TO A DBMS

The flexibility of the SAS System extends to this arena as well; there are alternative methods for accessing a DBMS. All of them except one involve SAS/ACCESS. The exception involves using a program, probably written in a third generation language, that can create ASCII copies of database tables that the SAS System can subsequently read. The analogous method for moving SAS data into a DBMS involves creating ASCII copies of SAS datasets that a 3GL program can read and create or append to database tables. This is an excessively tedious process that is only feasible if the need to communicate with a DBMS is an occasional one. For this reason, the remainder of this discussion will focus on SAS/ACCESS alternatives. The sample data consists of two tables from the preceding section on normalization:

speaker (name, ptitle, ptype, section)
maxtime (ptype, ptime)

Note that the attributes have been renamed to comply with the SAS 8 character limit to clarify the remaining discussion.

Access and view descriptors play roles in extracting from and writing to a DBMS. A SAS/ACCESS access descriptor contains data about one particular database

table. There are, though, multiple methods of joining data in database tables; they are outlined in the next section. SAS/ACCESS view descriptors are windows into the database and enable DBMS access. In most instances, view descriptors behave exactly like a SAS dataset. In order to communicate with the database using descriptor files (there are other ways) both types of descriptor are required. The code fragment in appendix figure 5 creates access and view descriptor files in noninteractive mode for the speaker and maxtime tables, above.

Extracting Data from a DBMS

There are two cases of DBMS data extraction: extraction involving a single DBMS table, and extraction involving data contained in more than one table. In each case, there are two options: using view descriptors, or using the SQL pass-through facility.

The use of view descriptors for extraction from a single table is trivial because of the similarity between view descriptors and SAS datasets. This code fragment creates a SAS dataset containing all of the data from the speaker table for invited speakers:

```
libname sasdata '<library name>';

data work.invited;
set sasdata.speaker
  (where=(ptype='INVITED'));
run;
```

The SQL pass-through utility allows users to pass SQL statements directly to the DBMS for processing and does not require any descriptor objects. To create the same dataset as above, the syntax would be:

```
proc sql;
connect to rdb (database='<dbms name>');

create table work.invited as
select * from
connection to rdb
(select * from speaker);

disconnect from rdb;
quit;
```

For the multitable case, where a join is required, users have these 3 options:

Create a SQL view by joining two view descriptors, one for each table. The code looks like this:

```

libname sasdata 'library name';

proc sql;
create view sasdata.sasjoin as
select name, ptitle, s.ptype, ptime,
       section
from sasdata.speaker s,
     sasdata.maxtime m
where s.ptype = m.ptype;
quit;

```

Create a DBMS view with access and view descriptors defined on it to enable user access.

Pass SQL code that joins the tables directly to the DBMS using the SQL pass-through utility.

```

proc sql;
connect to rdb (database='<dbmsname>');

create table work.extract as
select * from connection to rdb
(select name, ptitle, s.type, ptime,
       section
from speaker s,
     maxtime m
where s.ptype = m.ptype);

disconnect from rdb;
quit;

```

There are many factors influencing decisions about which of the preceding methods to use. Some of these are the SQL knowledge of the user community, the desirability of maintaining view and descriptor files, and the availability of computer resources like disk space and CPU time. The following observations, based on experience under the operating systems and software versions described in the introduction, might provide some very general guidelines.

The SQL pass-through facility requires SQL knowledge. View descriptors behave like SAS datasets so SAS System programmers need not negotiate a learning to curve to make immediate use of them. Also, view descriptors can provide full-screen access to DBMS tables via SAS/FSP® software.

Use of access and view descriptor files may impose some maintenance responsibilities especially if the structure of the DBMS tables is volatile. In this case, the access and view descriptors must be recreated when the DBMS table structure changes.

In informal performance metrics, the SQL pass-through option appears to require more CPU time. Sometimes the increase is on the order of 30 - 40%.

Writing Data to a DBMS

To write data to a DBMS from the SAS System, we have essentially the same options above, based on view descriptors and the SQL pass-through facility, with an additional option provided by the DBLOAD procedure.

The usefulness of the SQL pass-through facility is limited when writing to a DBMS, though, since it can only insert a row at a time. The syntax looks like this:

```

proc sql;
connect to rdb (database='<dbmsname>');

execute
(insert into speaker
 (name, ptitle, ptype, section)
 values
 ('JOE', 'A GOOD PAPER', 'INVITED',
  'BEGINNING TUTORIALS'))
by rdb;

disconnect from rdb;

quit;

```

Although it is possible, through the macro language, to invoke the SQL procedure repeatedly to handle multiple inserts, this method would not yield acceptable performance and would be considerably more complex than any of the following alternatives.

Since view descriptors behave like SAS System datasets, appending data to existing DBMS tables is not difficult. Suppose, for example, that a user has a SAS System dataset called 'toadd' that she wishes to append to the existing DBMS speakers table using the speaker view descriptor created in appendix figure 5. The code is identical to a program dealing exclusively with SAS System datasets:

```

libname sasdata '<library name>';

proc append
base = sasdata.speaker
data=sasdata.toadd;
run;

```

If the objective is to create a new table, then the DBLOAD procedure is the simplest alternative. Suppose, for example, that all of the speaker data currently resides in a SAS dataset called sasspeak and a user wishes to create the DBMS speaker table discussed in the preceding sections. The DBLOAD procedure provides a single step solution:

```

libname sasdata '<library name>';

proc dbload dbms=rdb
data=sasdata.sasspeak;
database='<dbmsname>';
table=speaker;
commit=20;
limit=0;
load;
run;

```

Note that the commit option specifies the number of insert transactions to buffer prior to issuing a commit command. Large numbers for this parameter will result in large journal files under Rdb. The limit option specifies the maximum number of rows to load; a value of 0, as above, signals the procedure to load an unlimited number of rows.

The DBLOAD procedure will not run if the target DBMS table exists prior to procedure invocation. One easy solution is to use the SQL command with the DBLOAD procedure to delete the target table prior to reloading it:

```

proc dbload dbms=rdb data=work.test;
database='<dbmsname>';
sql drop table psdtest;

table=psdtest;
commit=20;
limit=0;
load;
run;

```

It is easier to provide guidance about how to write to a DBMS than it is to suggest how to read from one.

Access and view descriptors are most useful when the DBMS table the user wants to load already exists. When the objective is to append multiple rows to an existing DBMS table, view descriptors offer the best alternative since they behave quite similarly to SAS datasets. If the DBMS table does not exist and the objective is to create and load a table with as little effort as possible, then the DBLOAD procedure offers an attractive alternative. This is particularly true since most DBMS products do not contain procedural languages with the flexibility and breadth of the SAS System.

A BRIEF LOOK AT CLIENT-SERVER SOLUTIONS

If SAS/CONNECT is installed on the local machine (often a PC) and on the remote CPU (e.g., a mini or mainframe CPU), and if SAS/ACCESS is licensed on the remote host as well, then users can enjoy client-server communication with a remote DBMS with just a few syntactical changes to the preceding code examples.

The following code will extract data from a remote DBMS, for example, and create a dataset called fromhost on the local machine. This dataset contains speaker data for all speakers whose names begin with a 'B.' A successful signon to the remote host via SAS/CONNECT is assumed.

```

proc sql;

connect to remote
(dbms=rdb server=<server name>
dbmsarg=(database='<dbmsname>'));

create table work.fromhost as
select *
from connection to remote
(select * from speaker
where name like <'B%');

disconnect from remote;
quit;

```

As another alternative, it is also possible to take advantage of remote library services to access view descriptors that were previously created and reside on the remote machine. Suppose that a user wishes to append data that exists in a local SAS System dataset to the speaker DBMS table that resides on the remote machine. Here is a previous example with a few changes to accommodate a client/server environment:

```

libname lochost '<library name>';

libname remhost REMOTE '<library name>'
server=<server name>;

proc append
base = remhost.speaker
data=lochost.toadd;
run;

```

CONCLUSION

This discussion is a survey of relational database theory and of the methods of communicating with contemporary database management systems in the current versions of the SAS System. It is not an exhaustive treatment of either subject. Some obvious areas for further exploration are performance of alternative access methods, client server access to remote databases, and SAS Institute's relatively new implementation of Open DataBase Connectivity (ODBC).

Many database theorists advocate normalization to third normal form with almost evangelical zeal and look on any collection of files without indexes, triggers, and primary keys as not worthy of consideration. Normalized databases are an excellent goal, but some database applications will not run in third normal form due to

performance limitations. Some databases, in fact, should not be normalized or should at least exist in denormalized form to support ad-hoc queries. Realizations like these have led to current discussions of 'data warehousing.'

Regardless of personal views on the need for a DBMS or the need for data normalization, database products and practices are too widespread for contemporary SAS System programmers to ignore.

REFERENCES

Teorey, Toby (1994), *Database Modeling and Design*, Second Edition, San Francisco: Morgan Kaufman Publishers, Inc.

Kroenke, David M. (1983), *Database Processing: Fundamentals, Design, Implementation*, Second Edition, Chicago: Scientific Research Associates, Inc.

SAS Institute Inc. (1990), *SAS/ACCESS Interface to Rdb/VMS*, First Edition, Cary, NC: SAS Institute Inc.

SAS, SAS/ACCESS, SAS/CONNECT, and SAS/FSP are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.

Other brand and product names are registered trademarks or trademarks of their respective companies.

AUTHOR CONTACT INFORMATION

Reader comments or suggestions are welcome:

Paul Dalberth
Family Health International
2224 Chapel Hill-Nelson Hwy.
Research Triangle Park, NC 27709
919-544-6979 ext. 532

Figure 1 - Flat Data

SPEAKER TABLE:

name	papertitle	degree1	degree2	degree3	company	papertype	maxtime	section
JOE	PROC TABULATE	BA	MS		ST&T	CONTRIBUTED	20	ADV TUT
JAN	P-VALUES	BA	MA	PHD	QUARTILES	INVITED	50	STATISTICS
JOE	GRAPH OPTIONS	BA	MS		ST&T	INVITED	50	INF VIS

Figure 2 - Normal Form 1

SPEAKER TABLE:

name	papertitle	degree	company	papertype	maxtime	section
JOE	PROC TABULATE	BA	ST&T	CONTRIBUTED	20	ADV TUT
JOE	PROC TABULATE	MS	ST&T	CONTRIBUTED	20	ADV TUT
JAN	P-VALUES	BA	QUARTILES	INVITED	50	STATISTICS
JAN	P-VALUES	MS	QUARTILES	INVITED	50	STATISTICS
JAN	P-VALUES	PHD	QUARTILES	INVITED	50	STATISTICS
JOE	GRAPH OPTIONS	BA	ST&T	INVITED	50	INF VIS
JOE	GRAPH OPTIONS	MS	ST&T	INVITED	50	INF VIS

Figure 3 - Normal Form 2

SPEAKER TABLE:

name	papertitle	papertype	maxtime	section
JOE	PROC TABULATE	CONTRIBUTED	50	ADV TUT
JAN	P-VALUES	INVITED	20	STATISTICS
JOE	GRAPH OPTIONS	INVITED	50	INF VIS

COMPANY TABLE:

name	company
JOE	ST&T
JAN	QUARTILES

DEGREE TABLE:

name	degree
JOE	BA
JOE	MS
JAN	BA
JAN	MS
JAN	PHD

Figure 4 - Normal Form 3

SPEAKER TABLE:

name	papertitle	papertype	section
JOE	PROC TABULATE	CONTRIBUTED	ADV TUT
JAN	P-VALUES	INVITED	STATISTICS
JOE	GRAPH OPTIONS	INVITED	INF VIS

COMPANY TABLE:

name	company
JOE	ST&T
JAN	QUARTILES

DEGREE TABLE:

name	degree
------	--------

JOE	BA
JOE	MS
JAN	BA
JAN	MS
JAN	PHD

MAXTIME TABLE:

papertype	maxtime
-----------	---------

CONTRIBUTED	20
INVITED	50

Figure 5 - Creating Access and View Descriptors

```
libname sasdata '<library name>';

proc access dbms=rdb;

create sasdata.speaker.access;    * CREATE ACCESS DESCRIPTOR FOR THE ATTENDEE TABLE *;
  database = '<database name>';
  table = speaker;
  assign = yes;

create sasdata.maxtime.access;    * CREATE ACCESS DESCRIPTOR FOR THE MAXTIME TABLE *;
  database = '<database name>';
  table = maxtime;
  assign = yes;

create sasdata.speaker.view;      * CREATE VIEW DESCRIPTOR FOR ATTENDEE TABLE *;
  select all;                     * SELECT EVERY COLUMN IN TABLE *;

create sasdata.maxtime.view;      * CREATE VIEW DESCRIPTOR FOR MAXTIME TABLE *;
  select all;                     * SELECT EVERY COLUMN IN TABLE *;

run;
```