NORMALIZED DATA: IMPLICATIONS FOR THE USE OF SAS® SOFTWARE, SQL, AND VIEWS

Patricia L. Garand, Syntex Research

ABSTRACT:

Normalization is the standardly accepted goal of the database design process because it ensures that data changes, additions, and deletions can be performed with a minimum of effort and error. To achieve this goal, however, the normalization process tends to scatter related data into many different tables, resulting in increased effort and potential error for the programmer who must re-connect the data to produce a report. This paper will illustrate the concept of data normalization. It will then compare the impact of normalized vs. denormalized data on the complexity of both SAS® and SQL code written for data analysis and reporting. Finally, solutions will be offered to reconcile the tug-of-war between the needs of data entry and those of data analysis. These solutions are 1) the creation of standard SQL views to make the data appear denormalized to the analysis programmer and 2) the creation of a set of actual denormalized tables from the original normalized ones.

INTRODUCTION:

While database textbooks, professors, and designers all focus on data normalization with the aim of preventing redundant and inconsistent data, the data analyst is left with the job of piecing back together this normalized data to produce reports. If the analyst is using SQL, this piecing together will consist of SELECT statements containing multi-table joins; the more data tables that need to be joined, the more difficult the code is to understand and the greater the chance for an incorrect join, resulting in incorrect reports. If the analyst is using SAS® code, the piecing together will involve many sorts, merges, and renames; the result will be a lot of complicated code.

Data normalization, therefore, while a lofty goal for the prevention of data problems, can represent a substantial headache for anyone trying to retrieve the data. There are some solutions to this problem, however. A normalized database can be maintained to ensure data integrity, but it can be made more accessible to an analyst either by establishing views that would give the impression of denormalization or by actually creating a separate set of denormalized tables from the originals.

This paper will describe the creation of a simple normalized database and will then show the ramifications on both SAS® and SQL code of denormalizing the data. In a nutshell, denormalized data leads to clearer and more accurate analysis code.

DEFINITIONS:

Before creating the database, definitions of a few buzzwords will be helpful:

ENTITY: A noun; a person, place, or thing; a distinguishable object about which data is to be stored.

RELATION: A two-dimensional table containing data for an entity.

PRIMARY KEY: Field(s) in a table that serve as a unique identifier for each record in that table.

FOREIGN KEY: Field(s) in one table that match a primary key field(s) in another table.

THIRD NORMAL FORM: All non-key fields depend on the key, the whole key, and nothing but the key.

NORMALIZED DATA: Data in third normal form.

To simplify things, if the answers to the following questions are all 'YES', the data is normalized:

- if you know the primary key, do you know all the other fields?
- do all fields depend on the entire primary key?
- do all fields depend on only the primary key?

EXAMPLE DATABASE:

Let's say that an outdoors company needs to keep track of information on activities that they sponsor.

What are the likely entities? Although there is no single correct answer to this question, a reasonable list would include potential activities, actual events, and the people who led the events.

What are the relationships between these entities? The standard method for answering this question is to construct an entity-relationship (E-R) diagram such as the one shown below:

![E-R Diagram]

A normalized database pulling together all these entities and listing the various attributes (fields) of the entities would look like this (where '-' identifies primary keys and '™' identifies foreign keys):
### ACTIVITY

<table>
<thead>
<tr>
<th>Activity</th>
<th>Cost per Participant</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hike</td>
<td>15</td>
</tr>
<tr>
<td>Ice Skate</td>
<td>10</td>
</tr>
<tr>
<td>SeaKayak</td>
<td>50</td>
</tr>
<tr>
<td>Snorkel</td>
<td>20</td>
</tr>
<tr>
<td>Spelunk</td>
<td>60</td>
</tr>
<tr>
<td>Ski</td>
<td>40</td>
</tr>
</tbody>
</table>

### EVENT

<table>
<thead>
<tr>
<th>Event</th>
<th>Leader</th>
<th>Date</th>
<th>Participants</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hike</td>
<td>Chuck</td>
<td>3/16/90</td>
<td>16</td>
<td>Big Sur</td>
</tr>
<tr>
<td>Spelunk</td>
<td>Melissa</td>
<td>3/28/90</td>
<td>10</td>
<td>Moaning Caves</td>
</tr>
<tr>
<td>SeaKayak</td>
<td>Jan</td>
<td>6/17/90</td>
<td>15</td>
<td>SF Bay</td>
</tr>
<tr>
<td>Ski</td>
<td>Melissa</td>
<td>2/19/90</td>
<td>24</td>
<td>Mount Rose</td>
</tr>
</tbody>
</table>

### EMPLOYEE

<table>
<thead>
<tr>
<th>Name</th>
<th>Phone #</th>
<th>Address</th>
<th>Favorite Sport</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jan</td>
<td>6439</td>
<td>San Jose</td>
<td>Snorkel</td>
<td>3000</td>
</tr>
<tr>
<td>Melissa</td>
<td>2829</td>
<td>Sunnyvale</td>
<td>SeaKayak</td>
<td>3000</td>
</tr>
<tr>
<td>Chuck</td>
<td>5484</td>
<td>Palo Alto</td>
<td>Hike</td>
<td>3000</td>
</tr>
</tbody>
</table>

A visual inspection of this database leads to the conclusion that a query against it would probably involve combining data from several different tables; this takes time and effort for the programmer. However, it also seems apparent that updating values would be fairly easy since all fields except primary and foreign keys would need updating in only one place.

A version of this database in denormalized form is shown below:
In this example of denormalization, most of the data is conveniently located in one table, but a couple of problems exist. How will the company store information on potential activities that have not yet been sponsored? It would be possible to put them into the EVENT table and just leave a lot of blank spaces where actual information does not exist; this would waste storage space and would also lead to query results that were somewhat difficult to interpret due to large amounts of missing data.

Another problem with this database is that fields in addition to the primary and foreign keys are duplicated, a situation which may lead to update anomalies. For example, if Jan changes her phone number, the database updater may mistakenly change it only in one table rather than in both; the result will be that data concerning Jan’s number will no longer be linkable since the match has been broken. This situation would lead to inaccurate query results.

**QUERYING THE TWO DATABASES:**

While it appears that updating data values is more accurate with a normalized database, the following code will show how querying a normalized database is more difficult than a denormalized one. First the query against both databases will be shown in SAS and then in SQL.

**SAMPLE QUERY:** Which employees have led outings of their favorite sport that resulted in income of at least $200?

**SAS**

**Normalized Data**

```sas
PROC SORT DATA=ACTIVITY; BY ACTIVITY; RUN;
PROC SORT DATA=EVENT; BY EVENT; RUN;
PROC SORT DATA=EMP; BY FAV_SPT; RUN;
DATA REQUEST (KEEP=LEADER EVENT INCOME DATE);
MERGE ACTIVITY (RENAME=(ACTIVITY=EVENT)) EVENT
  EMP (RENAME=(FAV_SPT=EVENT));
  BY EVENT;
INCOME = COSTPER * NUM_PART;
IF (INCOME >= 200) AND (LEADER = NAME);
RUN;
PROC PRINT DATA=REQUEST; TITLE 'SPORTS'; RUN;
```

**Denormalized Data**

```sas
PROC PRINT DATA=EVENT; VAR LEADER EVENT INCOME DATE;
WHERE (EVENT=FAV_SPT) AND (INCOME >= 200);
TITLE 'SPORTS'; RUN;
```
Using SAS® against the normalized data, you must invoke a PROC SORT for every dataset needing to be combined and must then combine them in a data step, using appropriate RENAME statements. The code is somewhat long and complex. In an industrial situation involving many tables, the length of the code and the trouble it takes to write it is greatly increased. In contrast, when accessing the denormalized data, essentially all that's needed is to specify the two simple conditions and print out the result.

Below is a similar exercise using PROC SQL.

**SQL**

**Normalized Data**

```
PROC SQL;
TITLE 'SPORTS';
SELECT E.LEADER, E.EVENT, (E.NUM_PART * A.COST_PER) AS INCOME, E.DATE
FROM EVENT E, ACTIVITY A, EMPLOYEE P
WHERE ((E.NUM_PART * A.COST_PER) >= 200) AND (A.ACTIVITY = E.EVENT) AND (E.EVENT = P.FAV_SPT);
```

Using the denormalized data, you need access only one table. With normalized data, you must identify the needed common fields from all three tables and then correctly join the tables together. While this may not be prohibitively difficult when only three tables are involved, the task becomes much more complex when an industrial database consisting of 50 or more tables is being accessed. Table joins become very difficult and errors may result in the production of Cartesian products, where every row in one table is joined with every row in another, rather than just joining on a match.

For example, if the programmer mistakenly left out the join between E.EVENT and P.FAV_SPT in the example above, each employee would end up having as many favorite sports as there are events in the event table, rather than just one.

In some cases incorrect joins may yield such subtle inaccuracies that the errors go undetected; in all cases of large multi-table joins, considerable time must be taken by the programmer to join correctly.

In addition to the complexity of writing code for normalized data is the issue of computer efficiency. As the number of data files and the amount of data being accessed and manipulated increases, computer performance decreases and the amount of work space needed increases.

**QUERY SOLUTIONS:**

One fairly easy way to solve the query complexity issue caused by normalized data is to construct standard views. Views are SQL statements that describe a SELECT of data from one or more tables; in essence, they provide virtual denormalized tables to the user.

Before constructing a view, the programmer should first determine which fields typically need to be accessed together. In the case of our normalized outdoors database where only 3 tables exist, all fields in all three tables could easily be combined connecting ACTIVITY from the ACTIVITY table with EVENT from the EVENT table and connecting LEADER from the EVENT table with NAME from the EMPLOYEE table. Two full table joins are performed here to allow for the fact that some join fields may not find a match in the other table; with the full join, records with no match will still be selected.

**SQL VIEW**

```
PROC SQL;
CREATE VIEW OUTDOORS.ALL AS
SELECT A.ACTIVITY, A.COST_PER, E.LEADER, E.DATE, E.NUM_PART, E.LOCATION, M.PHONE_NO, M.ADDRESS, M.FAV_SPT, M.SALARY
FROM OUTDOORS.ACTIVITY A
FULL JOIN OUTDOORS EVENT E ON A.ACTIVITY = E.ACTIVITY
FULL JOIN OUTDOORS.EMPLOYEE M ON E.LEADER = M.NAME;
```
The analyst now needs to access only one view, OUTDOORS.ALL, for any information that might be needed; complex joins are no longer necessary.

While substantial effort generally must go into the creation of standard views, the advantage to this approach is that in general the code is written, de-bugged, and validated once. Without standard views, each analyst will need to spend time writing merges or joins on-the-fly which may or may not be correct.

The view solution, however, ceases to be practical as the volume of the data increases since performing join after join to create virtual tables is much more costly in computer performance time and space than accessing a table directly.

When the amount of data being queried is so great that the use of views makes querying prohibitively slow, an alternative solution is the creation of a second set of actual tables that are denormalized. To do this, simply replace the word 'VIEW' with the word 'TABLE'. Alternatively, of course, the SAS® data step approach could be used.

Drawbacks to this are the need to purchase and maintain extra storage space and the logistics of creating the denormalized tables. It is advisable to create (and recreate after data updates) the denormalized tables in a consistent, routine manner. Although some effort must go into maintaining the denormalized tables, the payoff is that analysts can obtain answers to questions very quickly and reliably.

CONCLUSION:

Although normalized data provides for data integrity, it results in difficult and complex data retrieval. Possible solutions to this problem are either the creation of SQL views that virtually denormalize the data or the creation of a set of denormalized tables in addition to the originals. In these ways the analyst can write clear, simple, accurate code to obtain quick, accurate results.

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

Patricia L. Garand, Syntax Research, 3401 Hillview Ave., Palo Alto, CA 94303, U.S.A.