STRUCTURING SAS’ DATA SETS FOR USE WITH THE SQL PROCEDURE
Lewis Church, SAS Institute Inc., Cary, NC

INTRODUCTION

When designing a database for use with the SQL procedure, it is vital that the data be decomposed, or properly structured, into tables. The designer is often faced with many choices concerning how best to do this. Naturally, some choices are much better than others. From a retrieval viewpoint, a bad design may result in better execution-time performance than a good one. It is during data modification, for example, insertion, deletion, or updating, that the negative effects of a bad design are most noticeable. Although the literature on the subject of logical database design in relational systems contains many possible approaches to producing well-structured tables, there are basically only two approaches: an analytic approach (Fagin 1977) and a synthetic approach (Bernstein 1976; Bernstein and Beeri 1976; Fagin 1977a). This paper does not attempt to cover all the possible approaches, in particular it does not attempt to cover even one methodology. Rather, it concentrates on certain common aspects of the analytic and synthetic approaches and provides the reader with some guidelines and a background to understanding the current literature on this topic.

Central to the design of database tables is the notion of a data dependency, that is, a constraint on the possible values that a table can have because of semantics associated with whatever real-world items it is modeling. There are several kinds of dependencies that can exist between the columns of a table:

- functional dependencies (Bernstein 1975, 1976; Fagin 1977a; Beeri et al. 1978; Ullman 1980)
- multivalued dependencies (Fagin 1977b)
- join dependencies (Agaig 1986)
- hierarchical dependencies (Deobiol 1975)
- mutual dependencies (Nicolas 1978a,b; Mendelzon and Maier 1979)

The basic type of dependency and the one that this paper concentrates on is the functional dependency. Why? In practice, this is the type of dependency used most often in decomposing a relation because

- multivalued dependencies depend on context and are thus very hard for the database designer to visualize and get right in the first place (Beeri 1979; Sciore 1981)
- join, hierarchical, and mutual dependencies are also hard to visualize and even harder to use during the analysis and design phase (Beeri 1979; Sciore 1981)
- some of the better relational systems allow the designer to specify functional dependencies that follow from the fact that a primary key determines the other columns of a table and automatically enforces them. Even fewer relational systems will allow the specification and enforcement of arbitrary functional dependencies.

Briefly, if a set of column names in a table uniquely determines another, as PRODNAME does for PRODLIST in the Product table of the Seaside database, then PRODLIST is said to be functionally dependent on PRODNAME. Another way to state this is to say that PRODNAME functionally determines PRODLIST.

Before delving into issues involved with the good design of a set of tables for a database, it would be very instructive to show an example of a bad design and point out why it is bad. After doing so it will be much easier to explain certain concepts and their relevance to the desired goal.

The tables in the Seaside database are listed in Appendix 1, “Sample Tables,” in the SAS Guide to the SQL Procedure: Usage and Reference, Version 6, First Edition. This book does not have a table that contains information on the suppliers of the customers. Suppose that for each supplier you want to keep information concerning the supplier’s address, products and prices for those products. One possible instance of a table that has this type of information is the following:

<table>
<thead>
<tr>
<th>SUPPNAME</th>
<th>SUPPADDR</th>
<th>ZIP</th>
<th>PRODNAME</th>
<th>PRICE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Watermania</td>
<td>Treherneville</td>
<td>23319</td>
<td>flippers</td>
<td>15</td>
</tr>
<tr>
<td>Watermania</td>
<td>Treherneville</td>
<td>23319</td>
<td>jet ski</td>
<td>2150</td>
</tr>
<tr>
<td>Watermania</td>
<td>Treherneville</td>
<td>23319</td>
<td>kayak</td>
<td>190</td>
</tr>
<tr>
<td>Watermania</td>
<td>Treherneville</td>
<td>23319</td>
<td>raft</td>
<td>5</td>
</tr>
<tr>
<td>Watermania</td>
<td>Treherneville</td>
<td>23319</td>
<td>snorkel</td>
<td>12</td>
</tr>
<tr>
<td>Watermania</td>
<td>Treherneville</td>
<td>23319</td>
<td>surfboard</td>
<td>615</td>
</tr>
<tr>
<td>Watermania</td>
<td>Treherneville</td>
<td>23319</td>
<td>windsurfer</td>
<td>1090</td>
</tr>
<tr>
<td>The Coop</td>
<td>Birdsnest</td>
<td>23307</td>
<td>flippers</td>
<td>17</td>
</tr>
<tr>
<td>The Coop</td>
<td>Birdsnest</td>
<td>23307</td>
<td>kayak</td>
<td>200</td>
</tr>
<tr>
<td>The Coop</td>
<td>Birdsnest</td>
<td>23307</td>
<td>snorkel</td>
<td>9</td>
</tr>
<tr>
<td>The Coop</td>
<td>Birdsnest</td>
<td>23307</td>
<td>surfboard</td>
<td>630</td>
</tr>
<tr>
<td>Supplies 'R Us</td>
<td>Exmore</td>
<td>23405</td>
<td>flippers</td>
<td>16</td>
</tr>
<tr>
<td>Supplies 'R Us</td>
<td>Exmore</td>
<td>23405</td>
<td>jet ski</td>
<td>2000</td>
</tr>
<tr>
<td>Supplies 'R Us</td>
<td>Exmore</td>
<td>23405</td>
<td>kayak</td>
<td>190</td>
</tr>
<tr>
<td>Supplies 'R Us</td>
<td>Exmore</td>
<td>23405</td>
<td>snorkel</td>
<td>19</td>
</tr>
<tr>
<td>Supplies 'R Us</td>
<td>Exmore</td>
<td>23405</td>
<td>surfboard</td>
<td>720</td>
</tr>
<tr>
<td>Supplies 'R Us</td>
<td>Exmore</td>
<td>23405</td>
<td>windsurfer</td>
<td>1125</td>
</tr>
</tbody>
</table>

You can quickly see several problems with this table.

- Redundancy - The address and zip code of each supplier are repeated for each product supplied. This causes problems, not because it is wasteful (some systems are smart enough to represent logical redundancy without using physical redundancy), but because redundant information should always stay consistent.
- Update Anomalies - As a consequence of the redundancy, if Watermania moves from Treherneville to Onancock, it is possible to forget to change the values in the address column and zip code columns in all of the rows that correspond to products that Watermania supplies. This introduces potential inconsistencies.
- Insertion Anomalies - Suppose a new supplier, say XYZ of Cheriton, comes on the scene but it is not known what they supply. It would probably still be desirable to place this supplier’s name and address on file. Furthermore, assume that all columns for each row are required to have non-null values. Since you don’t know what XYZ supplies, it may be tempting to insert a single row in the table with the name and address portions filled in and set the corresponding ZIP, PRODNAME, and PRICE values to null. Assume that you eventually get pricing information on all the products that XYZ supplies. Will you remember to remove the incomplete row?
In the information in tables The first, SSZ, records the address which happens to be the most expensive operation in relational sys­ 
tem. This wasn't the case previously. The rest of this paper present in the supplier table, you now have to perform a join, which happens to be the most expensive operation in relational sys­ tems; this wasn't the case previously. The rest of this paper explores how to find a good replacement for a table or collection of tables that were badly structured.

The concepts of redundancy and dependency are closely related. As mentioned earlier, a dependency is mainly an assertion that of all the possible combinations of values that a table can be instanti­ ated with only a subset of these are valid. Basically, the ones that are valid are those that reflect a possible state of whatever the table is trying to model. If all states are not possible, then that means there may be some redundancy in each valid table. Therefore, if you know certain things about the table, other things can be deduced.

If the dependency is a functional dependency, then the form that the dependency takes is rather straightforward. In the Supplier table, if SUPPNAME functionally determines SUPPADDR, then it is impossible for that table to have two rows with the same value for the SUPPNAME column but have different values for their SUPPADDR column. Therefore, if you look at a row with a value of Watermania for SUPPNAME and Treherneville for SUPPADDR, then you know that another row that has Water­ mania as a supplier also has Treherneville as the value for SUPPADDR. So, if Watermania has 50 rows in the table, 49 of them contain redundant address information and have the potential for causing a lot of trouble when the data have to be modific­ ed.

As was stated in the second paragraph of this paper, there are dependencies other than functional ones. These more general types may also have redundancies associated with them. The form that these redundancies have are not as easy to spot as the functional ones. However, in the case of functional dependencies, the de­ pend­ ency seen in the Supplier table enables it to be decomposed into SS and SPP and then be recomposed, without any loss of informa­ tion, by a join.

SOME NECESSARY DEFINITIONS

An application environment is a collection of interrelated objects and activities (or events) that involve those objects (Algic 1986). The basic form of abstraction required by the relational model of data is classification of objects in an application environment into object types. Objects are considered to be of the same type if they have the same set of properties. An attribute specifies a single, static property of an object and cannot exist independently of that object. An object type is characterized by a set of attributes that are common to all objects of that type.

In the relational model, relationships among objects are represented in the same way as the objects themselves, that is, as objects. Both of these are called entities. Since, there are two types of entities, there are two kinds of dependency (Beeri 1980).

In the relational model, an entity is represented by a tuple of values and its attributes. A set of entities of the same type is represented by a set of tuples, where different entities are represented by different tuples. Such a set of tuples is formally called a relation.

In SQL, a relation is represented as a table whose columns corre­ spond to attributes and whose rows correspond to tuples. In addi­ tion, a table that represents a relation can not have duplicate rows.

Here are some examples of these concepts. Both the Supplier table and the Invoice table represent a collection of entities. Each Supplier entity has 5 attributes: SUPPNAME, SUPPADDR, ZIP, PRODNAME, and PRODCOST. The two rows

<Watermania, Treherneville, 23319, flippers, 16>

and

<Watermania, Treherneville, 23319, jet ski, 2150>

represent different entities. The Invoice table in the Seaside data­ base represents a relationship among objects, specifically a relation­ ship between the CUSTOMER, EMPLOYEE, and PRODUCT objects.

SOME BASICS OF LOGICAL DESIGN

Every entity should be represented by a single relation. Having more than one entity represented in a single relation causes update, dele­ tion, and insertion anomalies as demonstrated above. The precise way to structure relations so that there is exactly one entity per rela­ tion is to use normal forms and dependency information. Normal forms are formal rules for determining whether or not a relation rep­ resents a single entity. Two extremely desirable properties of any decomposition are

• it preserves all dependencies

• the information in a table that was decomposed into several smaller tables can be completely reconstructed by simply joining those tables together.

The first property is known in the relational database literature as dependency preserving decomposition and the other is known as the lossless join property.

There are quite a few normal forms:

• first (1NF)

• second (2NF)

• third (3NF)

• Boyce-Codd (BCNF)

• fourth (4NF)

• fifth (5NF), which is also known as project-join (PJNF).

These form a hierarchy where 1NF is the weakest in terms of requirements and 5NF is the strongest. The relationship is proper in that 5NF is stronger than 4NF, which is stronger than BCNF, and so on. Therefore, if a relation satisfies the requirements of, say, 4NF, then it also satisfies the requirements of 1NF, 2NF, 3NF, and

597
BCNF. Generally, the higher the normal form, the more desirable it is to have tables satisfy it. However, in practice, a designer rarely normalizes past 3NF. The reasons for this are as follow:

- tables that are in 3NF but not in 4NF or 5NF, though theoretically possible, are very unlikely to occur in practice (Date 1985)
- only 2NF, 3NF, and BCNF are based on functional dependencies
- getting a table to satisfy BCNF may result in a design that has fewer anomalies than 3NF but may result in a decomposition that cannot be dependency preserving (Tschirtzis 1982)
- it has been conjectured that all functional dependencies that satisfy 3NF but violate BCNF are in a sense irrelevant (Ullman 1982) as they tell the designer something about the structure of the real world that is of no use.

For example, assume that the zip code functionally determines the SUPPADDR in the Supplier table. This information is certainly interesting, but is not really useful because the primary application of this table is not to relate zip codes to addresses but to retain zip codes so that mail gets to an address quicker.

1NF is the basis for all the other normal forms; it and 2NF are mainly used as stepping stones to more desirable normal forms (Date 1986). When a relation or table is said to be normalized it is trivially in 1NF. Frequently, when a table is said to be normalized, it means that the table satisfies 3NF. It is important to note that this is often true, but does not necessarily have to be.

Essentially, 1NF says that every attribute (column) in a relation (table) must be simple. That means that there are no composite attributes in a relation nor set-valued columns in a table. Consider, for example, the table schema

\[
\text{SUPPLIER} \rightarrow (\text{SUPPNAME}, \text{ADDR}(\text{SUPPADDR}, \text{ZIP}), \text{PRODNAME}, \text{PRICE})
\]

This schema has a composite column named ADDR that consists of the sub-columns SUPPADDR and ZIP. A composite column may be even more complicated in that one or more of its constituents may be composite. This could very easily go on through several more levels. Also, the table schema may have more than one composite column. It is easy to see the complexity that would have to be inherent in any language that manipulated these tables if this sort of thing were allowed. Fortunately, the relational model prohibits this. As a consequence, tables have a simple and regular conceptual structure. Each row has the same number of columns and the columns are simple. The data for an arbitrary column of the table must all come from the same domain. This means that if a specific column of the table holds information about the days of the week, then that particular column of each row must contain information which is either null or is about a day of the week. It is this regularity and uniformity that the relational model derives a lot of its power and simplicity from as there are only five basic operators used in querying:

- union
- set difference
- Cartesian product
- projection
- selection.

Others such as intersection, quotient, join and natural join can be expressed in terms of the basic five. Furthermore, there only need be one form of each of these operators. Contrast the small number of basic operators with the number and different flavors of each needed in systems based on other approaches.

In a table, some of its columns are part of candidate keys, whereas others are not. Those that participate in the candidate keys are called prime; the remainder, if any, are termed nonprime. A table that is in 1NF is said to be in 2NF if all of its columns that do not participate in any candidate key are fully functionally dependent on every candidate key. To be fully functionally dependent means that no proper subset of a candidate key determines a nonprime column. If at least one nonprime column does not fully functionally depend on every one of these keys in the table, then a partial dependency is said to exist. This indicates that the table represents at least two entities and that certain update, insertion, and deletion anomalies can occur.

The Supplier table above is obviously in 1NF since its columns are simple and each row has the same number of columns as the table. However, that table is not in 2NF since the combination of SUPPNAME and PRODNAME is the primary key for it and SUPPADDR and ZIP are partially dependent on it. But, if you decompose it into SS and SPP then they are both in 2NF since the primary key of SS is SUPPNAME, which only has one part and the primary key of SPP is two parts (SUPPNAME and PRODNAME), where both parts are needed in order to functionally determine PRODCOST.

Even when a table satisfies 2NF it is still possible for it to have the anomalies mentioned above as 2NF is not sufficiently rigorous to get rid of all of them.

Take the table SSZ above and now assume that the semantics associated with it say that SUPPADDR functionally determines ZIP. This table is in 2NF but still has update, insertion, and deletion anomalies. The insertion anomaly is present since it is impossible to put the zip code associated with an address into it if there is no supplier with that address. Because SUPPNAME is a primary key, you cannot set a row's SUPPNAME column to null and fill in the rest of the columns with the desired values. The deletion anomaly is present because if the last row with a specific address is deleted, the information about the associated zip code is lost. Finally, the update anomaly is present because the same zip code is repeated for every supplier who has the same address. If, because of growth in an area, the post office changes the zip code associated with an address, then you have to update all supplier rows with that address in order to prevent inconsistencies that would violate the functional dependency of ZIP on SUPPADDR.

A table is said to be in 3NF when it is in 2NF and none of its non-prime columns is transitively dependent on any key of that table. A transitive dependency is a dependency such that a set of columns denoted by X functionally determines another set denoted by Y; Y does not functionally determine X, but Y functionally determines another set called Z with the proviso that the set of columns in Z cannot be a subset of those in Y.

To get rid of these anomalies, SSZ should be decomposed into

\[
\text{SS1} (\text{SUPPNAME, SUPPADDR})
\]

and

\[
\text{SS2} (\text{SUPPADDR, ZIP})
\]

**RECOMMENDATIONS AND CONCLUSION**

A database designer should not view the tables of a relational data base as self-contained, stand-alone items. There are both intra- and intertable dependencies that must be taken into account when modeling the entities of interest. Careless attention to these dependen-
cies can result in a design that is hard to maintain or modify because of update, insertion, and deletion anomalies. You can control this to a large degree by discovering the relevant dependencies during an analysis phase and making use of normalization theory to obtain a collection of tables that are properly structured or decomposed.

REFERENCES


Date, C.J. (1986), Relational Database: Selected Writings, Reading MA: Addison-Wesley.


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