Paper 6-25

Notes from the Outer Limits of 4GL RDBMS/SQL: The SAS® System to the Rescue

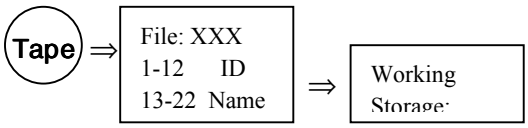
Sigurd W. Hermansen, Westat, Rockville, MD, USA

ABSTRACT

Relational schema and 4th Generation Languages such as the Structured Query Language, SQL, greatly simplify conditional subsetting and join/merges of data tables. But what do you do about unnormalized files that contain duplicates and inconsistencies? Or how do you select the nth item in a group of records? Do some database programming problems fall outside the normal scope of SQL or any other language based on relational algebra or calculus? This tutorial emphasizes the art of recognizing database programming problems with natural and efficient SQL solutions vs. misfits that really do lie outside the range of standard SQL forms. Examples show how the SAS® System fills in gaps in an enterprise RDBMS .

INTRODUCTION

The advent of relational database management systems (RDBMS's) in the 1980's profoundly changed the way programmers viewed programs and data files. Prior to RDBMS's, most programming tasks began with a detailed mapping of physical files to data structures.



The Data Division of a Cobol program and the input statement in a SAS® System data step survive as reminders of that section of a program.

Today in a RDBMS or SAS environment, programs merely refer to an access engine and a database object. The RDBMS has a built in catalog for each database. In a database program, table and column names in the catalog map directly to data elements. The programmer (almost) never has to deal with physical file paths or details of the devices used to read and write data. Properties of the database replace properties of devices and files. The database contains one or more data tables in the form of a "relational rectangle": a fixed-row-length, fixed column-width array of data elements.

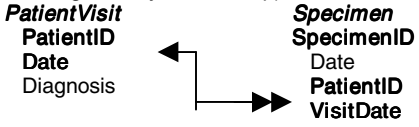
Each column in a database table contains one type of data and may have a domain of values associated with it that further limits the range of values of a type that it can contain. Relations within and among tables, called constraints,

R	E	L	A	T	I	O	N	A	L	row
E	*c*	*c*	*c*	.	.	.	.	.		?
C	*o*	*o*	*o*	.	.	.	.	.		?
T	*l*	*l*	*l*	.	.	.	.	.	.	?
A	*u*	*u*	*u*	.	.	.	.	.	.	.
N	*m*	*m*	*m*	.	.	.	.	.	.	.
G	*n*	*n*	*n*	.	.	.	.	.	.	.
L	*1*	*2*	*3*	.	.	.	.	.	.	.
E	.	.	.	.	.	.	.	.	.	.

implement a data model. At a minimum in each table, a key column has to have a distinct and different key value in the each row (primary key integrity constraint). This means that a value embedded in the data table uniquely identifies each row, and that ID can be used to find the one and only one row of data related to it.

Rows in database tables have no intrinsic order, unlike records in a physical file, though values in a column may have an index that imposes a virtual order on rows. A primary key column in a table may have a 1-1 or 1-many relation with key columns in other tables. A referential integrity constraint between two tables preserves a 1-1 or 1-many relation. For example, the RDBMS would reject an attempt to change or delete a primary key value in table A if that same value also appears in a column in table B that has a many-1 relation to the primary key in Table A. Nor would the RDBMS allow adding values to the column in Table B unless they linked to values in the primary key column of Table A. Instead of having a system flow diagram that shows the physical devices and files used by a program, a database has a data model that shows the columns in tables and the relations among them.

Data Model in the form of an E-R Diagram (small excerpt showing 1-many relationship)



INSIDE THE RELATIONAL RECTANGLE: WHAT YOU GET

Inside the sheltered world of a RDBMS, the programmer can (almost always) rely on the data model as a guide to programming. Consider an example in which a SAS client

wants data from an RDBMS client. The programmer first establishes a SAS LIBNAME connection to a database within the ideal RDBMS R:

LIBNAME DBR <R> "<path>";

The engine <R> acts as the "middleware" between SAS and a database object on a server. It makes programs on the SAS side into programs that the RDBMS R understands. In the case of "pass-thru" SQL programs, SAS merely puts the text of a program written in the SQL dialect used by R (or a common subset of ANSI SQL) on the program queue of an R machine. Then the engine returns the result of the R program to SAS in the form of a SAS dataset. A more advanced version of engine <R> converts SAS programs into R programs, sends the R programs to the R machine, and returns the results as SAS datasets.

The programmer has access to the R database's catalog. Combined with a description of the data model, say in the form of a compete E-R diagram, these tools give the programmer enough information to write programs that extract data, modify the R tables, and even modify the structure of the R database on the server. This notion of a client/server RDBMS works well over a local network, or even over the Internet. Smarter R engines determine which parts of programs should run on the server and which on the client. The distribution of work between client and server may minimize the volume of data that has to pass over the network, minimize the stress on the client machine, and disconnect clients as quickly as possible.

Set-logic languages like SQL have deep roots in the two thousand year old foundations of mathematics. Relatively few logical forms suffice for virtually all operations on databases:

/* Column Projection*/
Select a,d as r,b, ….
/* Intersect tables A⊗B (or reflexively, A⊗A)*/
from a as t1 [inner | left | full] join a as t2 ….
/* Row Subsetting */
on t1.a=t2.a ….
where d in (12,24,18) ….
/* Grouping and Summary */
group by b having count(*) >5 ….
/* Union */
select f from g union corresponding
select k from l ….
/* Difference [subquery variant] */
select x from t1
where x not in (select x from t2) ….

Basic insert, update, delete, and table maintenance operations complete the operations needed for normal programming within the confines of a RDBMS database.

OUTSIDE THE NORMAL SCOPE OF THE RELATIONAL RECTANGLE

Things get really ugly really fast when database programmers have to cope with real world problems that lie outside the scope of the relational rectangle. A few general types of problems of this sort seem both commonplace and important:

- Slicing and cooking large volumes of "raw" data (prior to entry into a RDBMS);
- Selecting the mth to nth rows in a RDBMS table;
- Key expansions;
- Fuzzy matching;
- "Reduced structure" databases.

The SAS System comes to the rescue when basic solutions lie outside the relational rectangle. Not that better planning and training of programmers might prevent these problems. Nor that RDBMS's lack methods of working around problems that do not conform to the logic of SQL and other 4GL languages. More that the SAS System offers a wider range of better tools for acquiring, enumerating, classifying, and restructuring the large masses of "before" data that need to be cleaned and repackaged for entry into a nice, neat database within a RDBMS.

SLICING AND COOKING "RAW" DATA

More than a few research databases consist of a collection of text files. Few of these have data structures that fit the relational rectangle. In the case of the Landauer database, header and detailed records in the same file have different formats (including lengths and column specifications). A field *rtype* has a value that distinguishes header from detail records.

```
The Landauer data (Header record)
@1    rtype       $char1.  /* type (H=Header) */
@2    id          $char7.  /* westat id */
@9    tapesc      $char1.  /* tape source */
@10   sn          $char9.  /* ssn  */
@19   lanpn       $char5.  /* landauer number */
@24   ln          $char25. /* last name */
@49   fn          $char25. /* first name */
@74   mn          $char1.  /* middle initial */
@75   sx          $char1.  /* sex */
@76   bd          $char8.  /* date of birth */
@84   acct        $char6.  /* UI account #  */
```

The detail records belongs to the header record they follow. While none of this would shock an experienced SAS programmer, a RDBMS programmer does not normally see column specifications that change from row to row, or groups determined implicitly by sequences in a physical file.

Say also, to make the problem more interesting, that first and middle name tokens may appear either in the ln column following a lastname token or in the fn and mn columns.

This SAS data step (Unix SAS) streams decompressed data from zipped files into a view:

```
filename lfil pipe "gzcat f1 f2";
/* Normalize database (3NF). */
data detail (keep=other id) head/view=head;
    retain id;
    infile lfil end=eof;
    input  @1 rtype $char1. @ ;
    if rtype='H' then do;
       input <header record> ;
       output head;
    end;
    else do;  input <detail record>  ;
              output detail;
         end;  <.....>
/* Parse header record. */
proc sql;    create table header as
   select id,scan(ln,1) as ln,
         coalesce(fn,scan(ln,2)) as fn,
         coalesce(mn,scan(ln,3)) as mn
   from head   ; quit;
```

In a typical RDBMS, the text read, select, parsing, and table import operations would have to be programmed outside the RDBMS, or in a RDBMS language (say, Oracle's PL/SQL).

SELECTING THE MTH TO NTH ROWS IN A RDBMS

Since a relational database may have to maintain more than one index for a table, it does not require an RDBMS to preserve a physical order of records. As a result, finding the top three most frequent ICD9 codes in a RDBMS database table proves tricky.

A generic method seems easy enough:

- *Group and summarize by code;*
- *Select 1st to 3rd code groups in descending order of count.*

The first step has a simple and quick SQL solution:

```
/* It takes about 1 second under MS W'95  */
/* to summarize about 16,000 obs. */
create view ICD9 as
select distinct code,count(*) as NofCode
from test
group by code    ;
```

Joe Celko's SQL solution for a top n problem (provided over SAS-L by Boyce G. Williams, Jr.) uses a reflexive join to rank the counts and limit those selected to the top three:

```
select distinct t1.code, t1.NofCode
   from ICD9 AS t1, ICD9 as t2
where (t1.NofCode <= t2.NofCode)
   group by t1.NofCode
having count(*) <= 3    ;
```

Although ingenious, SAS SQL cannot optimize the solution , as the log shows:

```
NOTE: The execution of this query involves performing one
or more Cartesian product joins that cannot be optimized.
NOTE:The query requires remerging summary statistics
back with the original data.
NOTE: The PROCEDURE SQL  used 25.16 seconds.
```

By comparison, an alternative to pure SQL, based on suggestions by Robert Krajcik and Ian Whitlock, does not need to generate a Cartesian product and works much faster:

```
create view ICD9S as
select * from ICD9
order by NofCode descending ;  quit;
proc print data=ICD9S (obs=3); run;

NOTE: SQL view …  has been defined.
NOTE:   PROCEDURE SQL used 0.05 sec ..
NOTE: PROCEDURE PRINT used 0.33 sec ..
```

The enhanced SQL completes the task about twenty times faster than the pure SQL solution.

KEY EXPANSIONS

Compared with the SAS, RDBMS's offer far fewer functions, formats, data informat and formats, and other data transformation tools. An actual SAS-L problem requires matching the last four characters of a ID in one data source to any 4 contiguous characters in a 9 character ID in another data source. It has a simple and natural solution in SAS but not in a typical RDBMS.

In a SAS data step, the function INDEX(ID,ID4) yields a value of TRUE when the condition stated above holds. Unfortunately, ID4 consists of six string values. Although SAS SQL allows joins on values of functions, formats, etc., it and other SQL or SAS datastep merges do not handle key expansions in WHERE or ON conditions. Nonetheless, a view can supply a table of expanded key values for the match:

```
data a4s/view=a4s (drop=i);   set a;
  do i=1 to 6;
     a4=SUBSTR(PUT(a,z9.),i,4);  output;
  end; run;
```

A simple SQL program then executes the view and matches the expanded set of key values to the last four characters of an ID:

```
CREATE TABLE AB AS
SELECT distinct a /* other */,B.*
FROM a4s as A, B
WHERE a4=SUBSTR(PUT(B,Z9.),6,4);
```

This implementation combines a SAS datastep view with SAS SQL. A program written by the SAS SQL man, Paul Kent, creates an in-line view in SAS SQL that uses the undocumented argument *offset* to accomplish the same result:

```
  CREATE TABLE AB3 AS
  SELECT distinct a /* other */,B.*
    FROM
     (select a.*, substr(put(a,z9.),offset,4) as a4
         from a,six ) as A , B
   WHERE a4=SUBSTR(PUT(B,Z9.),6,4);
```

Before too long Paul will retreat to the underground laboratory where he keeps secret SAS functions and figure out a way to optimize this form of query. For now the SAS workarounds provides a solution for key expansions.

FUZZY MATCHING

This problem actually breaks down into related problems:

- *Using a Function of Classifiers as a Key:*

A real problem arises in trying to find a set of the most similar case/control pairs. A naïve method (one of several possibilities) entails an iterative search for complete set of pairs of keys with a minimum of differences in a similarity index. Three logical rules guide the search:

- s=min(difflcase ID) union min(difflcontrol ID contains solution;
- singular Case ID in s belongs in solution;
- min(diffICaseID) in s belongs in solution unless linked to min(diffIControl ID) .

This process implements the search:

```
/*    1. add singulars to solution  */
/*    2. delete control ID dups from solution  */
/*    3. add unlinked mins to solution  */
/*    4. delete ID dups from solution    */
/*    5. delete IDs in solution from candidates */
/*    6. repeat w/ updated  candidates  */
/*    7. stop after exhausting candidates */
/*********************************/
```

The SAS Macro language offers the option of iterating through a sequence of data steps, procedures, or queries until a table attains a desired property or the number of loops reaches an upper limit:

```
%macro repeat(limit);
  %do %until (&nP=0 | &limit <=0);
            %let limit=%eval(&limit - 1);
            < more program >
            %main /* global (&nP -1)  */
            %put iterations left: &limit ;
  %end;
%mend repeat;
```

This algorithm finds the best matches first. Howard Schreirer has argued convincingly that this "greedy algorithm" usually fails to find a solution that minimizes the sum of differences in similarity index values between cases and controls. Again the SAS System comes to the rescue. PROC ASSIGN implements the classic assignment method in linear programming to find case/control pairs that minimize the sum of differences.

- *"Unduplicating" Databases:*

Primary key constraints in an RDBMS generally block attempts to add the same key value to more than one row in a data table. This prevents joins of tables in which "cross-linking" of duplicated key ID's combines data from unrelated rows. When rows of data that belong to the same entity have different ID's, however, nothing in the database links the rows. A search on one of the ID's finds only part of the data related in fact to the entity.

All or part of the non-key attribute data in each row may form the same pattern for each row that belongs to the same entity, but different patterns for rows that belong to different entities. If so, a pattern of attributes can serve as a "virtual primary key". Substantial and accurate personal information, say date of birth, street address and gender, would for example uniquely identify most records related to persons.

Errors introduce incidental differences in rows of data that belong to the same entity (and may explain how data for the same entity might come to have different primary key values). Coincidence or errors may leave rows of data so similar that unrelated records have the same key pattern. Practical solutions have to balance the costs of errors that lead to failures to link rows of data belonging to the same entity against costs of errors that lead to false matches.

RDBMS/SQL solutions tend to fall into the dreaded Cartesian product trap. A reflexive join of the form,

```
select * from T1,T2
where (T1.surname=T2.surname and T1.DOB=T2.DOB)
    or (T1.DOB between T2.DOB-180 and T2.DOB+180
and T1.SSN=T2.SSN)
    or ..... ; ,
```

implements the necessary strategy, but the usual strategies fail to optimize the search. Sorting or indexing by SSN, for example, limits a SSN search to above or below a midpoint, above or below a midpoint of the remaining range, etc., as in B-Tree searches. But a concurrent search on surname requires a different sort order or index. For a dataset of N rows, a brute force search has to make N^2 sets of comparisons.

Efficient solutions for unduplicating large datasets boil down to different implementations of an intrinsically inefficient method:

- build multiple indexes on sets of persistent identifiers;
- scan the dataset for hits on one or more indexes;
- select a relatively small subset of possible matches;
- score and rank index hits and review close calls.

Database unduplication adds another complication to already complex multiple indexing methods. In an index created from a database, each row of data should match each of the indexes created from it. Only subsequent matches to the same index value count as possible duplicates.

A thorough database unduplication effort may require as many indexes as identifying fields to eliminate match failures due to errors in a single field. Fuzzy indexing may increase that number further.

SAS-L archives include many recent examples and tests of multiple indexing and searching methods. Options include SAS dataset indexing, optimizing SQL, “big formats” (K. Self and self); data step hash indexing in temporary arrays (Paul Dorfman); commercial fuzzy or probabilistic matching programs ($$$); or custom procedural programs

For the indexing and searching of very large volumes of data, Dorfman’s hashing work best pretty much across the board (see his presentation “Private Detectives ….” in the SUGI 25 Data Warehousing Section). For smaller volumes of data, big formats may prove simpler to implement.

- Unreliable ID’s:

Another very real problem requires linking a relatively small set of records of interest (say, selected attributes of a few thousand DWI cases) to a huge set of records (say, selected attributes of all drivers in the USA). This proves straightforward given standardized and reliable ID’s. It becomes much more difficult when ambiguities and errors creep into key values.

The multiple indexing methods used in database unduplication work best in this case as well. Indexes for the smaller set of records pull key data into memory. A scan of the larger dataset then pulls one or more rows of data into memory, looks up key values on the indexes formed from the smaller dataset, and copies data into a results dataset if it finds a hit on any index. The scan of the larger dataset against the indexes typically reduces the data in the larger file to a small enough subset to rematch to the smaller dataset.

Multiple indexing fails only in cases of multiple key errors or poor discriminating information. Standardization of key values and “blurring” of incidental discrepancies further reduce the effects of unreliable keys. For that purpose SAS offers a rich selection of functions and operators. SOUNDEX(), SPEDIS(), INDEX(), and LIKE (SQL), enriched with numeric functions, data conversion tools, and frequency analysis procedures.

For example, errors in SSN fields may occur at a rate that ranges from 1 in 20 to 1 in 7. “Scoring” of a pair of SSN yields a fraction that measures in some sense the degree of similarity of the two strings of digits. An expression based on the SPEDIS(s1,s2) function in SAS scores a comparison of strings s1 and s2. The SPEDIS() function computes the “cost” of rearranging and substituting characters in s1 so the result equals s2. For SSN, the expression,

```
MAX(1-LENGTH(t1.SSN)*SPEDIS(t1.SSN,t2.SSN)/500,0.1),
```

computes a score between 1 (exact match) and 0.1 (not close). A product of scores from comparisons of persistent attributes of entities, as recorded in databases, indicates the combined degree of similarity of the database records.

The SAS language and procedures provide many of the special functions and procedures that search engines and data mining systems offer. Those and the statistical analysis tools in SAS give SAS an advantage over standard RDBMS’s when it comes to linking data tables on fuzzy

keys.

"REDUCED STRUCTURE" DATABASES

Databases designed to capture data on sequences of events and outcomes tend to have either too many tables of data or too many column variables in tables, or both, especially when different entities experience different sequences of events. For example, a database architect might separate results of diagnostic tests on specimens into a different table for each type of test. Adding new types of tests to the database would both increase the number of tables and force changes to its structure, Simple reports and updates might then require too many joins of tables to link data elements on different tables and too much programming effort to distinguish events that did not happen from missing data.

As structures of tables and column variables become more and more complex, database architects often recommend "denormalization" for the sake of efficiency. In the test results example, that might involve putting all test results for person in one row of a results table. At that point, one should step back and consider reducing the structure of the database.

Abstract table structures,
AA : ID testrslt1 date1 testrslt2 date2 testrslt3 ….
AT: ID date testtype result
can store the same information when testtype in AT contains the column labels in AA. The simpler structure AT, on the other hand, preserves the original degree of normalization of the database and allows adding or changing of test types with no need to restructure.

SAS PROC TRANSPOSE provides a more powerful version of the RDBMS/SQL "pivot" operation. The procedure call,

```
proc transpose data=lib.results out=lib.tresults
        (where=(compress(col1) ne "0" and col1 ne "" and
col1 ne "."));   by ID;   var _char_ _numeric_ ; run;
```

serves as a generic method for transposing form AA to AT.

CONCLUSION

RDBMS/SQL rules the relational rectangle, but substantial programming problems exist that lie outside the normal limits of 4GL/RDBMS/SQL methods. For these problems, solutions can become very difficult to develop within a RDBMS or highly inefficient. SAS data steps, procedures, functions, informats/formats, and other features complement 4GL/RDBMS/SQL programming.

AUTHOR'S E-MAIL ADDRESS:

hermans1@westat.com

NOTICE

- SAS is a registered trademark of SAS Institute Inc. in the USA and other countries.

REFERENCES

Riba, S.D. and E. Riba, ODBC*: Windows to the Outside World*, Proceedings of the Twenty-First Annual SAS Users Group Conference (1996), 548-557.

Riba, S.D., *More Windows to the Outside World: Configuring and Using ODBC in Version 7 of the SAS System*. Proceedings of the Twenty-Fourth Annual SAS Users Group Conference (1999), Paper 293.•

Raithel, M.A., *SAS Data Set Metadata*, A Westat White Paper (1999).

Celko, J. **SQL for Smarties: Advanced SQL Programming**, Morgan Kaufmann (San Francisco, CA, USA), 1995.

ACKNOWLEDGMENTS

SAS-L contributors provided the inspiration for this tutorial, as well as many of the problems (and solutions). Those cited by name made special contributions. Ian Whitlock, Mike Rhoads, and other members of the Westat SAS User's Group helped improve content and presentation. Linda Pierce contributed editorial support. The author alone takes responsibility for remaining defects. Views presented do not necessarily represent those of Westat, Inc.