Paper #1002

Think Thin 2-D: "Reduced Structure" Database Architecture

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

ABSTRACT

In the old days, even before MVS and SAS®, complex record structures ruled tape libraries. When large disk and memory buffers came on the scene, wide "flat file" structures put all of the data related to a single key value in fields of one record, and variable names replaced pointers to physical locations in a file. Big disks and E.F. Codd later inspired a relational database revolution that replaced physical files with basic logical relations and, in the programming environment, pointers to fields with relation.column references. Database systems have evolved steadily toward more abstract but simpler structures. What next? Object-oriented database systems and "data where?houses" regress in the direction of complex structures and file systems. In contrast, a reduced structure database begins with the current standard, a true relational database, and restructures it into a simpler and more abstract form. A structure of dependencies among two or more tables reduces to a single table in which a multidimensional primary key identifies each value. This presentation explores when and why this makes sense, what it takes to create a thin 2-D design, and examples of reduced structure databases in action.

INTRODUCTION

Estimates of the annual sales of database systems currently fall in the $8 billion dollar range. The fact that organizations are paying that kind of money to license and use DB2®, Oracle®, SQLServer®, and other database products suggests that database technology occupies a key role in private and public information systems. Most system experts agree that databases have two critical functions in financial, manufacturing, and administrative organizations:

- as servers for validated transactions entered concurrently by multiple users (including indexing, record-locking, internal security, and data integrity);
- as data storage systems that control access to information and provide an audit trail (automated compression, back-up, transaction roll-back, and external security).

The spectacular successes of some RDBMS's in the marketplace go well beyond the primary niche of database systems. Some information systems have little use for transaction updates or on-line storage. In these situations, the reasons for building an application system around a database product become far less compelling. This, of course, has not slowed the relentless campaigns of certified MS Office® and commercial RDBMS specialists to stamp all information resources with their labels. (It would not surprise me to hear that the Oracle Division of Megacorp has begun a hostile takeover of Megacorp's SQLServer Division.)

In their efforts to impress their colleagues in accounting and market with how easy it is to tap into a data warehouse through a spreadsheet, certified experts have built up transparent structures that add dramatically to the complexity and cost of application system development. In particular, programming by visual interface adds new structures to the existing system architecture.

In the database realm, RDBMS vendors are pushing the idea of an "enterprise database" that wraps their products around many of the data sources available to network clients.

RDBMS product specialists now envision database superstructures, data warehouses, built on multiple RDBMS's and extending across intranets and the Internet. These superstructures link data from different sources within an organization and possibly outside it, to present an integrated and continuous view of operations and progress toward goals. The following quick summary of the advantages and disadvantages of data warehouses actually says more about mindset and scope of product specialists:

DATA WAREHOUSE[1]

View: Enterprisewide
Building time: Years
Cost: Millions of dollars
Pros:
+ Presents a single version of the truth across the corporation
+ Compares apples-to-apples across the entire business
+ Supports indepth decision support analysis.
+ Integrates intellectual islands of knowledge

Cons:
- Project large, cumbersome and difficult to manage
- Must coordinate multiple vendors and products
- Never complete

THE EVOLUTION OF DATA STORAGE AND RETRIEVAL SYSTEMS

A very simple view of the descent of data systems over the past few decades (Table 1) shows successful basic structures (files, tables, databases) and their less successful elaboration (in italics).

Table 1.

DATA	STRUCTURE	ABSTRACTION
files	**field offsets**	**records**
indexed files	*subset offsets*	*hierarchies and networks*
2-D tables	**column, row cells**	**relations**
n-D tables	*indexed cells*	*MDDB*
database	**data-linked tables**	**schema**
object database	*pointer-linked objects*	*container objects*
data warehouse	*linked databases*	*metaschema*

Simple files of records on tape and disk remain important in current information systems. Two-dimensional (2-D) tables also survive in mathematical/statistical programming environments (such as SAS® System datasets).

Databases impose key constraints on 2-D tables. In each table a primary key composed of one or more columns represents a set of (distinct) data elements; the primary key and other elements in a row of a table represent an instance of a relation among observed values. Columns of one table relate to columns in other tables. To link rows in different tables, the data elements in the columns in one table bind to data elements in columns of the other table. This makes the relations among data elements a logical scheme as opposed to a feature of the implementation of a system. A typical feature of an implementation would be a network of memory addresses that contain other memory addresses (pointers). A good logical scheme works across implementations. Pointer schemes don't.

Programmers understand the concepts if not all of the reasoning behind them. For example, tables implemented as SAS® datasets can link on pointer values or link on values in columns named A1.ID and A2.PID.

```
/* Create table. */
data test;
input A1ID A1x A1y A1z A2PID A2x A2y A2z
      A3PID A3x A3y A3z
;
cards;
3 4 3 2 2 1 1 3 3 2 2 4;
run;
/*******************************/
/* Program One (below) uses a system pointer
(variable name) to select an instance of
patient ID (PID)and an outcome y.
*******************************/
data test1 (keep=PID y);
  set test;
   select (A1ID);
     when (2) do;
               PID=A2PID;
               y=A2y;
              end;
     when (3) do;
               PID=A3PID;
               y=A3x;
              end;
     otherwise put "error";
   end;
run;
title 'datastep';
proc print;
run;
/*******************************/
/* Restructure views to A(ID,x,y,x)
   and P(PID,x,y,z)
*******************************/
proc sql;
 create view Avw as
 select A1ID as ID,A1x as x,A1y as y,A1z as z
 from test
 ;
 create view Pvw as
 select *
 from  (select A2PID as PID,A2x as x,
               A2y as y,A2z as z
        from test)
                union corr
       (select A3PID as PID,A3x as x,
               A3y as y,A3z as z
        from test)
  ;
/*******************************/
/* Program Two (below)joins the
restructured tables based on data
values to select a PID and outcome
*******************************/
  create table test2 as
  select t2.PID,t2.y
  from Avw as t1 left join Pvw as t2
    on t1.ID=t2.PID
  ;
quit;
title 'sql';
proc print;
run;
title;
```

Both programs produce the same result, but the second achieves a significantly higher degree of logical independence. For instance, adding new instances of PID's does not require new column names and consequent restructuring of tables to add them. Also, interchanging rows in the table P (as seen in the virtual table Pvw) never changes the result, while interchanging data in A2PID,A2x,A2y,A2z and A3PID,A3x,A3y,A3z may. Moreover, the SQL join solution simplifies programming by linking data values directly and avoiding a link through a specific column name. Adding new instances of PID's requires adding more lines to Program One, but does not require changes to Program Two. Though contrived, this example shows some of the advantages of designing data systems around logical relations among data elements instead of around features of the implementation.

Just how a given system on a given platform actually implements the table restructuring and Program Two does not really matter so long as the SQL compiler conforms to ANSI standards. In this sense the program and data transport cleanly from a SQL compiler on one platform to another SQL compiler on another platform.

THIN 2-D "REDUCED STRUCTURE" DATABASES

If restructuring one table allows us to achieve a higher degree of data independence and simplify programming, could further restructuring buy us something more? We have seen one instance in which data kept in user programs, in this case variable names, shift to the system environment. Can we embed even more of a database structure in data?

In a commercial RDBMS, table names, column names, and key relations, main elements of the structure of a database, appear in tables that programs can view as just another form of data. The RDBMS uses these "metadata" to keep track of the tables and columns in a database. By changing the contents of the metadata tables, a system administrator restructures a database. Programmers generally do not have rights to update a database, but do have rights to create views of the database that amount to a virtual restructure. Either by database redesign or within construction of a database view, a database programmer has many options for setting up new keys and shifting columns from one table to another.

Production databases in RDBMS's tend to have many tables and many keys defined to link data from multiple tables in one query. The question of which of many distinct relational database schemes might work best for a specific database has no accepted answer. In fact, experts disagree on just about every facet of the question. Much of the debate involves the issue of many tables, fewer column names, and a more robust structure of key links ("normalized") vs. larger tables, more column names, and higher risk of errors creeping into the database ("denormalized").

Rather than add yet another opinion to the archives of an old debate, let us consider what might constitute a generic database scheme of minimum structure. The scheme has to have key dimensions that identify data elements. These may include standard dimensions such as location and time, but may also include other classifiers, such as a person ID, or an attribute. As a generic scheme, it must have at least one relation for each key dimension that has attributes related only to it. If not, combining data for any two key dimensions into one relation would force them to be either redundant or completely independent, since a partial relation between two key dimensions imposes a side condition on column values related to the pair. For example, relating the birth date, gender, and other demographic attributes of a person to both a person ID and a place name would imply that one person could have different birth dates and genders at different locations! Further, a generic scheme needs at least one other relation to classify values related to more than one key dimension.

Loss of information due to restructuring would, of course, offset any benefits that restructuring might yield. To assure us that the reduced scheme does not distort or lose information in the original scheme, some form of restructuring of the reduced database has to restore the original database.

A realistic example shows how a more complex database scheme reduces to a simpler scheme. A standard codebook describes the results of optical-mark recognition (OMR) scanning of a questionnaire form. In this example we include only the first part of a long questionnaire:

QUEX	Column Name	Domain
	ID	preassigned (unique)
	rec	1-2
	subrec	1-99
	(rec=1, subrec=1)	
	Age	18-99,M,E
	Sex	1-2,M,E
	(rec=2, subrec=1-99)	
	Site	1-5,M,E
	Day1	0-3,M,E
	Day2	0-9,M,E
	Month	01-12,MM,EE
	Year1	8-9,M,E (1980-1999)
	Year2	0-9,M,E
	Q1	Y,N,M,E
	Q2	minor,moderate,severe,M,E
	Q3	good,OK,bad,M,E

Coders complete the ID/rec/subrec header. The respondent responds to the range of questions between Age and Q3. The Day1 question presents a grid of three bubbles; Day2 a grid of 10. If a respondent fails to mark any of the choices in a grid or the scanner fails to pick up a mark, the scanner writes "M" for missing to the output file. If the scanner detects more than one mark in a grid, possibly due to an incomplete erasure, it outputs an "E" for error. Each respondent responds once to the questions under rec=1 and one or more times to the questions under rec=2; each response has a different subrec number. Q2 and Q3 response codes translate to the levels shown. One file holds data in this scheme. It has the form of an table with one rec=1 header block followed by subrec=n repeats of the rec=2 block.

This scheme decomposes cleanly into a relational scheme:

Persons
- ID*
- Age
- Sex

Events
- ID*
- Date*
- Site*
- Q1
- Q2
- Q3

The asterisks identify primary keys. Events occur no more frequently than once a day per Site, so the Date and Site columns distinguish repeating events for the same person. This new scheme has close to minimal structure. It leaves a relation for the Persons dimension and a relation of persons to Date and Site. The decomposition into two tables eliminates the rec and subrec columns (and the extra coding required to break a single table into repeating blocks).

The motive for reducing further the structural complexity of the database becomes compelling when Q1-Q3 becomes Q1-Q200 and the structure includes skip patterns of the form, "*If you are male, skip to question 30; else, please respond to questions 25-30*". Programming data cleansing rules becomes a chore, and dependencies among questions become pervasive and more difficult to control. Say that responses to Q19, Q28, Q111, Q151, and Q176 have the same domain: % of total five-year dose of drug x taken during a year. The questions ask for annual %'s for the first through the fifth year. A range and sum check program has to contain references to the five variables. If for some reason the names change, the program has to change as well.

Restructuring by shifting column names to data elements helps reduce the system-dependent structure of the database. SAS® programmers recognize the automated procedure that remaps column names related to columns and rows of data to parallel columns of label and data values: PROC TRANSPOSE. In its simplest form, it functions as a transpose operator in a matrix language (such as APL or SAS®/IML). Consider parallel row vectors name[i,1] and value[i,1]. The database catalog contains name; a data table contains value. Transpose operations, name[i,1]` = name[1,i] and value[i,1]` = value[1,i], exchange the values of the [row,column] indexes to change the row vectors to column vectors. Since the index i still links the same elements in the two vectors, it does not change information that the parallel vectors contain.

A relatively simple extension of a transpose of vectors "pivots" each row of column names and values in a table and replicates one or more identifiers in each row across the columns created from the values in a row.

This structure,

ID* Date* Site* Q1 Q2 Q3 …. Q200,

courtesy of a general purpose SAS®/BASE procedure,

```
proc transpose data=events out=reduced
   (where=(compress(Response)
             not in ("","."))
and Qn ne "ID" and Qn ne "Date"
and Qn ne "Site")
    rename=(_name_=Qn col1=Response)
    keep=ID Date Site _name_ col1);
by ID Date Site;
  var _char_ _numeric_ ;
run; ,
```

becomes

ID* Date* Site* Qn Response,

where Qn has the domain Q1-Q200 and TRANSPOSE converts numeric responses to questions to character values.

[CAVEATS: sort or index data on BY variables prior to TRANSPOSE; KEEP= executes before RENAME= executes before WHERE= options; automatic conversion of numeric to character values in TRANSPOSE leaves numbers and missing values right-justified in field. Use (COMPRESS() to remove leading blanks; not valid in States where prohibited.]

With this thin 2-D structure in place, it becomes possible to write a data cleansing program that contains no references to the variable names/labels Q1-Q200. Say that a table or dataset Edits has this structure and content:

Edits

Type	label
pctX100	Q19
pctX100	Q28
pctX100	Q111
pctX100	Q151
pctX100	Q176

By referencing Edits, this cleansing program subsets Events (as defined above) and filters out instances in which Reponses to specific questions add up to less or more than 100%:

```
proc sql;
  create table Errors as
  select ID,Date,Site,sum(input(
  compress(Response),3.)) as pctsum,
  'pctX100' as Error
  from reduced
  where Qn in (select label from Edits)
  group by ID,Date,Site
  having calculated pctsum ne 100
  ; quit;
```

[CAVEAT: conversion of right-justified number will not work as expected unless leading blanks removed; $500. fine for removing tag from upholstery.]

This demonstration gives only a hint of the extent to which a thin 2-D database facilitates integration of distributed and heterogeneous data. Other virtues of “reduced structure” databases include

- the option to systematically delete rows containing “dead space” such as missing or null values or default values;
- ease of further restructuring by pivoting thin 2-D structure and summarizing into cross-tabs;
- minimal programming required to subset or summarize values;
- improved normalization of database scheme.

DATA WHERE?HOUSING AND BEYOND

Requirements for data systems vary substantially in details across organizations, but all should include

- a sound, logical scheme or model for linking corresponding data elements;
- independence of data and methods of implementing a system that will give the target users access to the data that they need.

Once teams of RDBMS product specialists acquire special data modelling tools and invest time and resources in logical database schema, we might expect to see them work together to integrate database catalogs into a grand data warehouse “metaschema” or catalog of catalogs. That does not seem to be the pattern. Enterprise data warehousing projects more typically draw on the expertise of outside consultants and begin as scavenger hunts for sources of information. This “data where?house” phase typically occupies much of the time dedicated to data warehouse development.

Data Warehouse (DW) developers extract data from multiple sources (centralized DW) or obtain data by request from different owners (federated DW). A “star schema” model for enterprise data warehousing, as close to a standard as currently exits, restructures transaction records into a common structure and stacks huge quantities of records into one “fact” table[2]. The composite primary key to the fact table has one column in it for each of the dimensions of the database. A minimal fact table contains little more than dimension values (ID’s, classifiers, place, time, etc.) and a numeric value (counts, $’s, amounts, or summary rates, means, etc.). Links from the dimension keys in the fact table radiate in all directions to the dimension tables (thus the star analogy).

A star schema structure of a data warehouse closely resembles a thin 2-D “reduced structure” database. Both achieve high degrees of normalization. Both suffer from massive redundancies in keys. Minimal structure implies n-dimensional key values related to a single atomic value. Both deviate from pure relational database designs, in that the type of data in a column determined by a set of dimensions may depend on the values of those same dimensions, but for good reason. If it makes sense to reduce structure at the level of a data warehouse, it would make even better sense to apply the same system architecture principle to database design. It would make life far easier data warehouse developers.

If RDBMS products and complex database structures work best, then why do organizations that have invested financially and strategically in RDBMS technology continue to pay license fees for the SAS® System? A few reasons seem obvious and ubiquitous. SAS® has the tools that basic tools that programmers need to

- extract data from multiple systems on different platforms;
- restructure data into common and useful structures;
- clean and standardize “dirty data”;
- deliver information in meaningful forms.

Expect to continue to see informats and type conversion functions, input from records and parsing of text lines, and frequencies and transposes for a number of years more. Visual interfaces and RDBMS’s need clean and consistent data, and, as we know too well, you don’t buy them off the shelf.

CONCLUSION

Packaged solutions and product labels have hefty price tags. A thin 2-D “reduced structure” database embeds more information in the data themselves. A logical and pragmatic data model neither depends on nor constrains a database system. Data system architects who focus on achieving close correspondence of data model to reality and independence of data and implementation have found a good formula for success.

ACKNOWLEDGMENTS

Special thanks go to Richard Gadol, Lillie Stephenson, Robin McEntire, and Yiling Xu.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Sigurd W. Hermansen
Westat: An Employee-Owned Research Company
1650 Research Blvd
Rockville, MD 20850
(301) 251-4268
hermans1@westat.com
www.westat.com

[1] http://www.healthcare--formatics.com/issues/1998/09_9bullet.htm
[2]http://www.informix.com/informix/solutions/dw/redbrick/wpapers/star.html