Converting Entity-Attribute-Value (EAV) Source Tables into Useful Data with SAS® DATA Steps

Thomas E. Billings, Baskar Anjappan, Junling Ji
Union Bank, San Francisco, California

Abstract

Entity-attribute-value (EAV) tables are a non-relational, row-based data structure in which each variable is recorded on a separate row. The EAV table has columns that identify: a) the associated entity (source table) name, b) the relevant attribute/variable name, and c) the value of the attribute/variable, usually in string (character) form. Such data must be transposed, converted (in many cases) from character to numeric/datetime, adjusted for length, and cleaned to be useful. These operations are complex and cumbersome in SQL, but the SAS® DATA step provides a more elegant way to process EAV data. In particular, the use of RETAIN variables, FIRST and LAST.BYVARIABLE operators, and IF-THEN-ELSE logic are powerful tools for turning complex, messy, unusable EAV data into standard, clean by-column data.

Introduction

To understand *entity-attribute-value (EAV) tables*, consider a single relational database table (entity) named E, with attributes/columns/variables named A1, A2,…, and each of these attributes has the values V1, V2, …, etc. Now take a single row from the relational database and output the data for that row into a different table with multiple rows, one row for each variable in the relational database table, and each output row has the following 3 variables and values:

- **Entity name** = name of entity (table) = E
- **Attribute name** = name of attribute/column/variable = A1, A2, …,
- **Attribute value** = value of attribute/variable, often in character/string form (only) so data conversions are necessary for numeric, datetime/date attributes/variables = V1, V2, etc.

In the cumbersome format described above, metadata and data are commingled and the relational features of an RDBMS are mostly lost. This structure wherein each variable becomes a separate row is referred to as an EAV table. To make matters worse, the data for multiple relational database tables may be converted to EAV format and stored in a single EAV table.

The EAV structure is highly problematic, with the following major issues:

- The row based structure of the data (each variable a separate row) must be transposed -- an arduous task in SQL -- into a conventional column-based structure (each variable a column) to be useful.
- Data type conversions (e.g., character to numeric/datetime) may fail if/when the input format of the data changes.
- For an EAV table, SQL constraints are usually at least one of: impossible to enforce, extremely difficult to write, and/or complex and slow to run.
- EAV tables may contain superfluous records and selecting the correct records may require complex logic.
- Sizing: these tables can quickly grow to become extremely large. An EAV with 200 variables and 50% superfluous rows will have 400 rows for each real record, i.e., 10K real records will be 4 million rows in the EAV table.
- Data dictionaries are useless for EAV tables as the real metadata of interest are expressed as data in the EAV table.
- Attribute_names (variable names) don’t necessarily follow SAS® or SQL naming rules. They also may be inconsistent when multiple related entities feed a single EAV table.
- If they are manually entered, Entity_name and Attribute_name can spawn new variable names (also new variables) via spelling errors, typographical errors, and differences in spelling practices (e.g., U.S. vs. U.K. spellings).
- If users can paste (erroneous or outdated) html into the Attribute_value field, and those data are passed along into downstream output XML files, the result can be a broken/unreadable XML file.

It is very hard to justify an EAV table design from a data modeling perspective, as it violates the basic principles of good design. The mapping of multiple variables from one table into a single variable in another produces variables that are (by design) logically overloaded. Unfortunately, the EAV structure is common and some people defend using the structure. Some designers may see it as a way to handle uncertainty in their data structures; while business units who write programs may see it as a way to change their data structures without the “chore” of going through formal IT production change management procedures. The latter reason is ironic, as the use of EAV tables imposes a significant effort to read/write data. If the choice is to use an EAV or do standard (best practice) change management, then change management is the optimal choice.

Joe Celko, author of numerous books and articles on SQL (2005, 2009, 2012) makes a number of criticisms of the EAV design:

- The EAV is a “design flaw” and in some cases is the end result of code first, think later practices.
- It is a common error among programmers who have an object-oriented programming background and/or who have worked with loosely typed programming languages.
- It is the result of an object-oriented schema forced onto an RDBMS, and often fails.

A similar data structure, OTLT: one true lookup table, shares many of the same issues as EAV tables. OTLT tables are also criticized by Celko and other authors.

The structure described above is the basic EAV format. Variations on the structure are possible and can include:

- Multiple Attribute_value variables in each row: one for string values, one for numeric, one for datetime; this reduces the required data type conversions.
- Entity_name and Attribute_name can be combined in one variable, for an (E.A)V structure.
- Additional information/metadata may be concatenated into Entity_name or Attribute_name, increasing the effort required to parse the data.
- The number/level of superfluous rows in the table can make the transpose more difficult. Similarly, in EAV tables with a large number of superfluous rows, it can be difficult and tedious to identify which is the correct row to pull.

Next, let’s review the different approaches to processing EAV tables.

**Approaches to processing EAV tables**

**SQL**

Most real-world EAV files are not as simple as described above. Depending on how many variables are needed and the circumstances, complex CASE statements with GROUP BY may be used, often with multiple extracts from the file. The multiple extracts are then combined via additional joins. Sample SQL pseudocode for the core logic to process an EAV may look something like the below:

**Part 1:** transpose the data, leaving it in character/string form

```sql
SELECT Customerid, /* ID variables that define the natural data blocks, 
  i.e., multiple rows to be transposed */
  Archiveid,
  MAX(my_raw_variable1) as my_transposed_variable1
```
...additional MAX statements, 1 for each transposed variable...

FROM

(SELECT
    Customerid,
    Archiveid,
    CASE /* 1 CASE statement per variable to be transposed. This maps variable value to variable name. */
        WHEN (Entity_name = 'target_entity') AND
            (Attribute_name = 'target_variable') AND
            (condition #3) AND
            (condition #4) AND
            ....other conditions here...
        THEN SUBSTR(Attribute_value,1,25)
    ELSE ''
    END AS my_raw_variable

    ... additional CASE statements for other variables to be captured ...

FROM &schema..&tablename

    WHERE (Entity_name = 'target_entity') /* a (redundant) efficiency constraint when EAV contains multiple entities */

) GROUP BY CustomerID, ArchiveID

A MAX function with GROUP BY is needed above to eliminate null/missing values and transpose the data, i.e., in this context, produce 1 row per group with multiple variables. (Note that MAX is used even though the data are character.)

Part 2: convert data to numeric/datetime and/or clean the data

This is an SQL step with statements that convert the raw character data and create new variables that are numeric/datetime as required, and also cleans the data. The code required will vary per the variable, EAV, and application/requirements.

It may be possible to combine the SQL pulls above for an entity into a single pull with numerous complex CASE statements. However, the structure above may be preferred as it is easier to maintain and to understand.

Part 3: Additional SQL data pulls may be required

EAV tables combine all of their data into a single table. However, the data in an EAV may be at different levels, e.g., customer data, purchase data, payment data. Separate sets of pulls – parts 1 & 2 above – are often needed to separate the EAV data into tables at the proper level for later use. That is, multiple pulls may be required for the customer-level data, then separate, multiple pulls for the purchase-level data, etc. The end result is that to get data out of an EAV and into a reasonable, relational table structure may require a very large number of SQL pulls and joins.

SAS PROC TRANSPOSE

If the EAV data are:

1. simple and relatively clean,
2. with few or no extraneous rows, and
3. the values of Attribute_name already conform to SAS variable naming standards, or
4. if it is relatively easy to process the EAV table so it satisfies conditions 1-3,

then PROC TRANSPOSE can be an efficient and useful tool. Here the invocation of PROC TRANSPOSE should:
• list the variables that define the natural data blocks in the BY statement,
• the ID statement specifies the Attribute_name variable (or the cleaned version thereof), and
• the VAR statement specifies the name of the Attribute_value variable.

After this, the primary follow-ups needed are data-type conversions and possibly repackaging the variables into multiple tables to reflect the different levels of data.

Extremely complex and messy EAV tables require extensive preprocessing in DATA steps (or with PROC SQL) before they can be used with PROC TRANSPOSE and/or may require numerous invocations of PROC TRANSPOSE with different WHERE= conditions to extract the target data, followed by multiple SQL or DATA step joins to combine the transposed data. Under these conditions, the SAS DATA step is usually the optimal tool for handling EAV tables.

Using SAS DATA Steps to Process Complex EAV Tables

Overview of raw data structure for a loan rating tool (financial scorecard) database

Table 1 below provides a snapshot of a typical entity–attribute–value table structure for commercial loan rating data, i.e., the Risk Analyst system. With only 7 columns/variables, this huge table includes all historical information for loan ratings, e.g., loan details, obligor information, guarantor information, financial factors, qualitative inputs, and rating scores. In this example, CUSTOMERID and ARCHIVEID are the variables that define the block of rows for each loan rating; BLOCKID contains entity name and attribute name making it an (E.A)V structure; IOTYPE, IOID and PROPERTYID provide additional information related to the attribute, and PROPERTYVALUE is the attribute value.

The CUSTOMERID and ARCHIVEID variables are created when the data for a new loan are archived in the system. There are numerous values for BLOCKID, to support the loan information and rating scores. For each value of BLOCKID, there are multiple rows with differing values of IOTYPE, IOID and PROPERTYID. Only one row with a specific combination of values for BLOCKID, IOTYPE, IOID and PROPERTYID is valid for a particular attribute. The PROPERTYVALUE corresponding to that specific combination provides the correct attribute value.

The EAV table has a very simple design with “generic” interface components, e.g., read and parse BLOCKID and the other variables to get the variables of interest. This interface makes it easy (perhaps too easy) to add new attributes/columns/variables to the table. However, that same design makes statistical calculations and report creation into very complicated processes.

Sample data follow in Table 1. Note that the data below are artificial: ID variables, names, numbers are all fictitious.

Table 1: Risk Analyst -- Layout: sample (artificial) data

<table>
<thead>
<tr>
<th>CUSTOMERID</th>
<th>ARCHIVEID</th>
<th>BLOCKID</th>
<th>IOTYPE</th>
<th>PROPERTYID</th>
<th>IOID</th>
<th>PROPERTYVALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1234</td>
<td>1</td>
<td>UBOCFF.Approvername</td>
<td>1</td>
<td>text0</td>
<td>Text</td>
<td>Jim Forbes</td>
</tr>
<tr>
<td>1234</td>
<td>1</td>
<td>UBOCFF.Approvername</td>
<td>1</td>
<td>bool0</td>
<td>TextIsMapped</td>
<td>FALSE</td>
</tr>
<tr>
<td>1234</td>
<td>1</td>
<td>UBOCFF.Recommendername</td>
<td>1</td>
<td>text0</td>
<td>Text</td>
<td>Cary Morgan</td>
</tr>
<tr>
<td>1234</td>
<td>1</td>
<td>UBOCFF.Recommendername</td>
<td>1</td>
<td>input</td>
<td>Score</td>
<td></td>
</tr>
<tr>
<td>1234</td>
<td>1</td>
<td>UBOCFF.Recommendername</td>
<td>1</td>
<td>score</td>
<td>Score</td>
<td>1</td>
</tr>
<tr>
<td>1234</td>
<td>1</td>
<td>UBOCFF.Recommendername</td>
<td>1</td>
<td>weight</td>
<td>Score</td>
<td>0</td>
</tr>
<tr>
<td>1234</td>
<td>1</td>
<td>UBOCFF.Recommendername</td>
<td>1</td>
<td>weightedscore</td>
<td>Score</td>
<td>NaN</td>
</tr>
<tr>
<td>1234</td>
<td>1</td>
<td>UBOCFF.Recommendername</td>
<td>1</td>
<td>bool0</td>
<td>TextIsMapped</td>
<td>FALSE</td>
</tr>
<tr>
<td>1234</td>
<td>1</td>
<td>UBOCFF.ManagementQuality</td>
<td>1</td>
<td>text0</td>
<td>SelectedCategory</td>
<td>B</td>
</tr>
<tr>
<td>1234</td>
<td>1</td>
<td>UBOCFF.ManagementQuality</td>
<td>1</td>
<td>numeric0</td>
<td>OrigSysPD</td>
<td>FALSE</td>
</tr>
</tbody>
</table>
Tables 2 below is from a PROC CONTENTS on the input EAV files. Comparing tables 1, 2 illustrates how the EAV structure commingles metadata and data in a single structure, making extraction of data very difficult.

Table 2: Part of PROC CONTENTS for input (EAV) table:

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>Len</th>
<th>Format</th>
<th>Informat</th>
<th>Label</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARCHIVEID</td>
<td>Num</td>
<td>8</td>
<td>11</td>
<td>11</td>
<td>ARCHIVEID</td>
</tr>
<tr>
<td>BLOCKID</td>
<td>Char</td>
<td>50</td>
<td>50</td>
<td>50</td>
<td>BLOCKID</td>
</tr>
<tr>
<td>CUSTOMERID</td>
<td>Num</td>
<td>8</td>
<td>11</td>
<td>11</td>
<td>CUSTOMERID</td>
</tr>
<tr>
<td>IOID</td>
<td>Char</td>
<td>50</td>
<td>50</td>
<td>50</td>
<td>IOID</td>
</tr>
<tr>
<td>IOTYPE</td>
<td>Num</td>
<td>8</td>
<td>11</td>
<td>11</td>
<td>IOTYPE</td>
</tr>
<tr>
<td>PROPERTYID</td>
<td>Char</td>
<td>50</td>
<td>50</td>
<td>50</td>
<td>PROPERTYID</td>
</tr>
<tr>
<td>PROPERTYVALUE</td>
<td>Char</td>
<td>1024</td>
<td>1024</td>
<td>1024</td>
<td>PROPERTYVALUE</td>
</tr>
</tbody>
</table>
Outline: Logical structure of a SAS DATA step to input and transpose EAV data

The financial scorecard data above provides an example of a complex EAV table with numerous superfluous rows. The relevant scorecard data must be extracted from an RDBMS table which has stored the data elements in the EAV row structure. The data for a single customer record might be stored in more than 200 rows. The DATA step has been used to extract the data and transpose the rows to columns/variables. The logic required here has the following structure:

/* Step 1: sort the file so data are in their natural blocks */
PROC SORT DATA=raw_scorecard_file OUT=scorecard_extract_file;
   BY CUSTOMERID ARCHIVEID;
RUN;

/* Step 2: transpose, convert, and clean the data in a single DATA step */
DATA my_scorecard_output;
   SET scorecard_extract_file;
   BY CUSTOMERID ARCHIVEID;
   ATTRIB statements to define: variable length, label, and format of output/transposed variables;
   RETAIN list of output variables;
   IF FIRST.ARCHIVEID then CALL MISSING (list of output variables);
   ** Conditional IF-THEN-ELSE logic to set output variables, based on;
   ** BlockID, ioid, etc. goes here. This is the core of the program;
   ** Each variable is processed in this part, transformed, cleaned, and created in the most efficient way;
   IF LAST.ARCHIVEID then OUTPUT; * write out a record ONLY on last record of natural data blocks;
   DROP intermediate variables/work variables not needed in output file;
   -or- KEEP statement for output variables;
RUN;

In the above, the output variables are defined using the ATTRIB and RETAIN statements. The CALL MISSING sets the output variables to missing on the first row of each block of rows for a loan rating record, triggered by using the FIRST.ARCHIVEID indicator variable. Then the conditional statements IF-THEN are used to: 1) identify the correct rows of interest, 2) clean, convert and process the raw values to derive the target output variables. Appropriate logic to transform the values is required and varies according to the variables being processed. Once the row for LAST.ARCHIVEID has been read and processed, we have processed all rows in a data block, and issue an OUTPUT statement to write out 1 row for the entire block of rows per customerid, archiveid pair. This single output row contains the transposed variables.

Sample SAS DATA step source code to input data from an EAV:

data file2;
set file1;
   attrib
      Recommender_Name length=$100 label ='Recommender Name'
      Approver_Name length=$100 label ='Approver Name'
      Management_Quality length=$1 label ='Management Quality'
      Final_PD_Risk_Rating length=8 label ='Final Mapped Risk Grade'
      Past_Raised_Capital length=8 label ='Past Raised Capital'
      CIF_Num length=$19 label ='CIF Num';
by customerID ArchiveID;
   retain _all_;
if first.ArchiveID then
    call missing(Recommender_Name, Approver_Name, Management_Quality, Final_PD_Risk_Rating, Past_Raised_Capital, CIF_Num);
if IOTYPE = 1 then
    do;
    if PropertyID = 'text0' then
        do;
        if IOID = 'Text' then
            do;
            if BlockID = 'UBOCFF.Approvername' then
                Approver_Name=PropertyValue;
            if BlockID = 'UBOCFF.Recommendername' then
                Recommender_Name=PropertyValue;
            end;
        else if IOID = 'SelectedCategory' then
            do;
            if BlockID = 'UBOCFF.ManagementQuality' then
                Management_Quality=PropertyValue;
            end;
        /* other conditions */
        end;
    else if PropertyID = 'numeric0' then
        do;
        if IOID = 'OrigSysPD' then
            do;
            if BlockID = 'UBOCFF.FinalGrade' then
                Final_PD_Risk_Rating=input(strip(PropertyValue,32.));
            end;
        /* other conditions */
        end;
    else if PropertyID = 'input' then
        do;
        if IOID = 'Score1' then
            do;
            if BlockID = 'UBOCFF.PastRaisedCapital' then
                Past_Raised_Capital=input(strip(PropertyValue,32.));
            end;
        /* other conditions */
        end;
    end;
if IOTYPE = 2 then
    do;
    if PropertyID = 'text0' then
        do;
        if IOID = 'Text' then
            do;
            if BlockID = 'UBOCFF.CIFNumber' then
                CIF_Num=PropertyValue;
            end;
        end;
    end;
if last.ArchiveID then output file2;
Drop BLOCKID IOID IOTYPE PROPERTYID propertyvalue;
run;
Table 3: Part of PROC PRINT for output (transposed, derived) file, showing artificial data:

<table>
<thead>
<tr>
<th>CUSTOMERID</th>
<th>ARCHIVEID</th>
<th>CIF_Number</th>
<th>Approver_name</th>
<th>Recommender_name</th>
<th>Management_Quality</th>
<th>Final_PD_Risk_Rating</th>
<th>Past_Raised_Capital</th>
</tr>
</thead>
<tbody>
<tr>
<td>1234</td>
<td>1</td>
<td>204001234</td>
<td>Jim Forbes</td>
<td>Cary Morgan</td>
<td>B</td>
<td>9</td>
<td>18420680.74</td>
</tr>
<tr>
<td>1234</td>
<td>2</td>
<td>204001234</td>
<td>Jim Forbes</td>
<td>Cary Morgan</td>
<td>B</td>
<td>8</td>
<td>20435465.32</td>
</tr>
<tr>
<td>1235</td>
<td>1</td>
<td>201256789</td>
<td>Mary Lin</td>
<td>Mark Moore</td>
<td>A</td>
<td>6</td>
<td>3265655.45</td>
</tr>
<tr>
<td>1235</td>
<td>2</td>
<td>201256789</td>
<td>Mary Lin</td>
<td>Mark Moore</td>
<td>A</td>
<td>4</td>
<td>3575656.46</td>
</tr>
<tr>
<td>1235</td>
<td>3</td>
<td>201256789</td>
<td>Mary Lin</td>
<td>Mark Moore</td>
<td>B</td>
<td>5</td>
<td>3634239.13</td>
</tr>
</tbody>
</table>

Advantages of the DATA step vs. SQL approach

Overall, the source code to process an EAV structure in the DATA step is usually less voluminous and less complex than the SQL code required to perform the same function. Other advantages of the DATA step approach include:

- A single DATA step - in one pass through the raw data - can process EAV data that would require multiple pulls if done in SQL. The DATA step can create multiple output files in a single step, and is more efficient in this situation.
- For many users, the DATA step code is easier to understand and maintain than the complex SQL required.
- Adding new variables to a job that extracts variables from an EAV and then pushes them downstream into output files is usually easier with the DATA step than SQL. The inherent limitations of SQL ("SELECT *" vs. "SELECT variable1, variable2, etc.) may require that you insert the names of new/added variables in the SELECT clauses of multiple downstream joins.

Automation of source code generation

For a given EAV, if the processing required to create all of the output variables is not too complex, the coding step can be automated and driven by metadata. The properties of the input rows to be pulled can be described in metadata, and PROC FREQ may be helpful in creating the metadata. Similarly, other metadata can be created to describe the output variables. A cross-reference table can be constructed that maps input variable metadata to output variable metadata. Once the metadata are available, the entire source code set required to extract tables from an EAV could be generated by a computer program. If you have to process a large number of derived tables extracted from a smaller set of EAV tables, this approach may be worth considering.

Epilogue

If you have to input data from an EAV table, the optimal approach to use depends on the quality of the EAV table. If the EAV table data are simple and clean, then PROC TRANSPOSE may be the optimal approach. For more complex data that requires extensive selection logic and cleaning, the SAS DATA step with the approach described above is optimal. Finally, if you are writing output files, it is strongly recommended that you avoid using the EAV structure for your output files.

References:


Acknowledgment:

Thanks to Jack Hamilton (of Kaiser Permanente) for valuable comments and suggestions. Any errors herein are solely the responsibility of the authors.

Contact Information:

Thomas E. Billings  
Union Bank  
Basel II - Retail Credit BTMU  
350 California St.; 9th floor  
MC H-925  
San Francisco, CA 94104  

Phone: 415-273-2522  
Email: tebillings@yahoo.com

SAS and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.

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