Paper 104-30

Taming Transformation: A Strategy for Metadata within the Warehouse

Katherine Prairie, Stone Country Solutions Ltd, Calgary, Alberta

ABSTRACT

Extraction, transformation and loading (ETL) processes must change over time to accomodate evolving business and transformation rules. Many programmers handle this situation by creating different versions of their programs or routines. However, business analysts and decision-makers who rely on reports drawn from data warehouses must have access to such information as well. One approach is to store metadata relating to business and transformation rules in tables within the data warehouse. If handled properly these tables may be used to quickly generate documentation for business and technical staff alike.

The SAS® System SQL procedure and SAS/ACCESS® can be used to provide an up-to-date record of parameters, and rules relating to transformation processes. These may provide the basis for inputs into the transformation processes themselves, or they may be used to generate current reports suitable for business and technical staff.

To illustrate this concept, a set of tables designed to store parameters and rules are created using PROC SQL. Dates are stored in the tables to provide both a historicial and current record of metadata used in the transformation process. These details are retrieved during the transformation process using a variety of join techniques in PROC SQL. A parameter table is built from the current metadata to optimize performance of the transformation routine. Finally, the tables are used to generate a report for business staff using SAS Output Delivery System (ODS).

This paper is intended for those who have some exposure to PROC SQL. SAS/ACCESS is used to establish a connection to the data warehouse.

INTRODUCTION

There are several advantages to storing metadata relating to transformation routines in the data warehouse itself. First, everyone involved has access to current rules and parameters without having to read the transformation routines. This can be particularly valuable for business analysts and others responsible for setting business rules that ultimately drive data transformation processes. Views may be used to populate standard reports for business and technical staff, eliminating the need for separate documentation. As documentation of metadata often takes a back seat to operational activities, the ability to generate reports from the same tables updated for transformation processes is appealing to many. It also provides an advantage to business staff in that current documentation can be easily made available to assist them in interpreting reports or building models.

Second, a historical view of past transformation parameters can be stored without requiring access to archived programs or routines. In the event data has to be reloaded into the warehouse, transformation rules appropriate to the original loaded data must be applied. Although the past values may be documented, they may not be readily available within the transformation routines themselves. Often the solution is to keep multiple copies of the routines or documentation. However, document and program control in such cases adds extra burden on those working on data warehouse extraction, transformation and loading (ETL) processes. When both historical and current values are stored in tables within the warehouse, transformation processes may be run using a set of rules and parameters that are date-appropriate.

Third, a single documentation system is easier to keep up-to-date than several documents directed at different groups of individuals. The goal is to provide current summary information about the transformation and business rules to business analysts and others involved in the ETL processes, without adding additional documentation burden. Reports generated directly from warehouse tables eliminate the need for additional documentation. An HTML or PDF file, both of which are easily accessed and shared may be generated with current parameters. Views may also be built to provide this functionality, providing those knowledgeable with SAS System procedures such as the REPORT or SQL with the ability to generate their own reports.

Finally, the rules and parameters required by transformation routines can be accessed directly. If the metadata itself is stored within the data warehouse, the transformation routines can self-assign parameters.

Storing metadata within data warehouse tables is not without challenges. Performance must be carefully considered in the design of tables, and programming code used to extract rules and parameters. In addition, the number and nature of the parameters and rules used in the transformation process may place an excessive demand on storage in the data warehouse. However, in our work we have found that in many cases the storage requirement is reasonable and the documentation advantages outweigh other factors.

The following paper sets out one example of storing parameters in a set of data warehouse tables. It steps through the creation of the tables and provides samples of SQL statements that can be used to extract the information. A view is created and used to populate a simple report of current transformation rules and parametesr. Many of the steps shown here may also be accomplished using Base SAS DATA steps and procedures other than PROC SQL.

CREATING THE TABLES

There are many possible table structures that can accommodate metadata in the warehouse. A portion of a set of three metadata tables documenting business and transformation rules are included here.

Each transformation routine or data model is described in a table called *routine*, shown below. A unique code or primary key called *routine_code* is supplied for each entry in the table. An index is automatically built for columns for which a primary key constraint is defined. This index is used to ensure unique entries in the table and facilitate joins or connections between the *routine* table and other metadata tables.

Table: routine

routine_code	routine	description	eff_date
1000	Campaign targets	Determines mailing list for catalogues	01JAN2001
1075	Loyalty targets	Determines mailing list for loyalty cards	01APR2003
2000	Specialty targets	Determines mailing list for special events	01APR2004

The PROC SQL statements used to create and populate the table *routine* in an Oracle database are included below in Figure 1. The SAS/ACCESS LIBNAME statement is used here to establish a connection to the *stone* schema within the *datastore* database. A primary key constraint is added to the routine code column in-line with the specification of the column name and data type. Once the table is created, an INSERT statement is used to add rows of data to the table. The SASDATEFMT specification on the INSERT statement ensures that SAS dates are correctly interpreted by the Oracle database system.

Figure 1

```
PROC SQL;
LIBNAME ora ORACLE
user = scott password=tiger path=datastore schema=stone;

CREATE table ora.routine
      (routine_code num primary key,
      routine char(20),
      description char(50),
      eff_date date
      );

INSERT into ora.routine
values (1000,'Campaign targets','Determines mailing list for
catalogues','01JAN2001'D)
values (1075,'Loyalty targets', 'Determines mailing list for loyalty
cards','01APR2003'D)
values (2000,'Specialty targets', 'Determines mailing list for special
events','01APR2004'D);
```

The *d_parameter* table shown below is used to provide details on the parameters themselves as well as information such as the routines in which the parameters are used, a full description of each parameter and the date the parameter was first implemented. The routine codes stored in the *routine* table are used here to link each parameter to a specific routine. The table in the warehouse that stores the values used in transformation processes for this parameter is entered into the *related_table* column. A single parameter table may be created for parameters that require only a few transformation values. However, if there are numerous values associated with a parameter it is often preferrable to store these values in a separate table. In this example, the zip codes used to determine the market regions for the parameter *zip_to_market* are stored in a separate table called *c_zip_market*. Values used to transform credit card and gender codes are stored in a single table called *routine_parameter*. Rules and descriptions may be as complex or simple as necessary. If substantial rules are required or many of the same rules are repeated, the rule column may store a code associated with a specific rule stored in another table rather than the rule itself.

Table: d_parameter

parm_ code	routine_ code	parameter	related_table	description	eff_date	rule
10	2000	credit_card	routine_parameter	Credit card names to in-house codes	15APR2004	Credit card code supplied by Head office
20	1000	zip_to_market	c_zip_market	Table of market regions based on zip code	01JAN2003	Blank zip code uses city as basis of market regions
30	1000	age_group	routine_parameter	Age groups for segment markets	01AUG2004	Age as of March 1
40	1075	summer_age	routine_parameter	Age range targeted for summer campaigns	01JAN2004	Age as of March 1
50	1075	fall_age	routine_parameter	Age range targeted for fall campaigns	01JAN2004	Age as of March 1
60	2000	gender_code	routine_parameter	Recode numeric gender codes to character codes	01DEC2004	Gender codes to character codes

PROC SQL is used in Figure 2 to create and populate the table *d_parameter* in the same Oracle database as the previous example. It is not necessary to repeat either the LIBNAME or PROC SQL statement before executing the statements shown. A composite primary key constraint is added to the table to ensure that each combination of routine code and parameter code occurs only once in the table. Notice that in this CREATE statement, the constraint specification is included as a separate line. This is known as out-of-line constraint specification and it allows named constraints to be applied to a combination of table columns. Again, the SASDATEFMT specification is included in the INSERT statement used to add rows to the newly created table.

Figure 2

```
CREATE table ora.d_parameter
      (parm_code num,
      routine_code num,
      parameter char(20),
      related_table char(20),
      description char(50),
      eff_date date,
      rule char(50),
      constraint parameter_pk primary key(routine_code, parm_code,eff_date)
      );

INSERT into ora.d_parameter
values(10,2000,'credit_card','routine_parameter','Credit card names to in-house
codes','15APR2004'D,'Credit card code supplied by Head office')

values (20,1000,'zip_to_market', 'c_zip_market','Table of market regions based on
zip code','01JAN2003'D,'Blank zip code uses city as basis of market regions')

values(30,1000,'age_group',’routine_parameter','Age groups for segment
marketing','01AUG2004'D,'Age as of March 1')
```

```
values (40,1075,'summer_age','routine_parameter','Age range targeted for summer
campaigns','01JAN2004'D,'Age as of March 1')

values (50,1075,'fall_age','routine_parameter','Age range targeted for fall
campaigns','01JAN2004'D,'Age as of March 1')

values(60,2000,'gender_code','routine_parameter','Recode numeric gender codes to
character codes','01DEC2004'D,'Gender codes to character codes ');
```

A third table called *routine_parameter* holds transformation values for some parameters. For each parameter there may be one or more rows of values in this table. If historical values are also available, such as with the *summer_age* parameter, separate entries are made for each effective date. Selected entries for each parameter in the *routine_parameter* table are shown below.

There are a number of strategies for storing parameters of different data types. In the table below, string and integer values are stored in separate columns and a *value_type* indicator is provided. The data type indicator may be retrieved by transformation processes to determine the appropriate set of columns from which parameter values may be drawn.

Table name: routine_parameter

routine_ parm_code	parm_ code	eff_date	outcome	string_value	num_ value	low_ value	high_ value	value_type
100	10	15APR2004	AMEX	AMERICAN EXPRESS				STRING
101	10	15APR2004	MC	MASTER CARD				STRING
102	30	01JUN2004	1			18	24	INTEGER
103	30	01JUN2004	2			25	35	INTEGER
104	30	01JUN2001	1			20	45	INTEGER
105	30	01JUN2001	2			46	60	INTEGER
106	40	01JAN2004	1			25	45	INTEGER
107	40	01JAN2004	2			46	55	INTEGER
108	60	01DEC2004	F		1			STRING
109	60	01DEC2004	M		2			STRING

The SQL statements required to create and populate the displayed table are shown in Figure 3. Notice that there are two constraints added to this table. The first is a primary key constraint that prevents duplicate entries of a parameter and outcome for a specific effective date. It is specified out-of-line because it involves a combination of three columns. The second constraint, a check constraint called *value_ck* is used to verify that incoming values to the high and low cutoff value columns are reasonable. It has been added out-of-line to allow a name *value_ck* to be specified for the constraint. If you refer back to Figure 1 you will notice that the primary key constraint associated with the routine_code column does not have a name associated with it.

Figure 3

```
CREATE table ora.routine_parameter
      (routine_parm_code num,
      parm_code num,
      eff_date date,
      outcome char(10),
      string_value char(20),
      num_value num,
      low_value num,
      high_value num,
      value_type char(10),
      constraint routine_pk primary key(parm_code,eff_date,outcome),
      constraint value_ck check(high_value >=low_value)
      );
```

```
INSERT into ora.routine_parameter
values(100,10,'15APR2004'D,'AMEX','AMERICAN EXPRESS',.,.,.,'STRING')
values(101,10,'15APR2004'D,'MC','MASTER CARD',.,.,.,'STRING')
values (102,30,'01JUN2004'D,'1','',.,18,24,'INTEGER')
values (103,30,'01JUN2004'D,'2','',.,25,35,'INTEGER')
values (104,30,'01JUN2001'D,'1','',.,20,45,'INTEGER')
values (105,30,'01JUN2001'D,'2','',.,46,60,'INTEGER')
values (106,40,'01JAN2004'D,'1','',.,25,45,'INTEGER')
values (107,40,'01JAN2004'D,'2','',.,46,55,'INTEGER')
values (108,60,'01DEC2004'D,'F','',1,.,.,'STRING')
values (109,60,'01DEC2004'D,'M','',2,.,.,'STRING');
```

RETRIEVING PARAMETERS FROM THE TABLES

In many cases the parameters required for transformation processes may be drawn directly from the *routine_parameter* table using an SQL equi-join or a Base SAS DATA step. In the example provided here, values of different data types are stored in the same table. There are several options for dealing with such a situation. The PROC SQL COALESCE function provides one solution, extracting the first non-null value from any set of columns of the same data type.

The second challenge is related to the storage of both historical and current values within the same table. Correlated subqueries may be used extract the current parameter values by selecting the values associated with the maximum date for each parameter in the *routine_parameter* table. A temporary table *current_parameters* has been created in Figure 4, eliminating the need to retrieve the current set of parameters more than once. An index has been added to this table to ensure optimal performance.

Figure 4

```
CREATE table ora.current_parameters as
(SELECT    parm_code, outcome, num_value, string_value, value_type,
           low_value, high_value
FROM       ora.routine_parameter p
WHERE      eff_date = (SELECT max(eff_date)
                       FROM ora.routine_parameter r
                       WHERE r.parm_code = p.parm_code)
);

CREATE index parm_code on ora.current_parameters(parm_code);
```

A Base SAS DATA step could also be used to extract this information from the *routine_parameter* table. A table stored in a database can be treated exactly like a SAS data set and processed using the SAS programming language and passed to SAS procedures such as PROC SUMMARY or PROC TABULATE.

Transformation processes working with high and low cutoffs to assign parameter values can retrieve the needed values through SQL non-equi joins. In Figure 5 two non-equi-joins between the incoming data set *dataset1* stored in a native SAS library and the *current_parameters* table are used to compare the incoming age code to the cutoff values for each age group. A non-equi-join uses a comparison operator rather than an equals (=) operator when comparing a column of values in one table to a column of values in another table. In this example, the age is compared to the low and high values for each age group to determine the appropriate age group for that age. The report generated by the SELECT statement has been included in the example.

An INPUT function has been included on the SELECT clause to convert the character version of the outcome in the transformation table to a numeric value. Conversion functions may also be included on a WHERE clause to handle any necessary conversions in the incoming data to be loaded. Automatic data conversion does not occur in PROC SQL and columns involved in a join must have the same data type.

Figure 5

```
LIBNAME ora ORACLE
user = scott password=tiger path=datastore schema=stone;

LIBNAME loading 'c:\temploading';

CREATE table loading.dataset1
(id num,agecode num);

INSERT into loading.dataset1
values (1,18)
values (2,23)
values(3,43)
values(4,33)
values(5,30);


TITLE      'Age Group for each incoming Age Code';
SELECT     id, agecode,input(c.outcome,2.)'Age Group'
FROM       ora.current_parameters c, loading.dataset1 d
WHERE      c.parm_code = 30 and
           d.agecode >= c.low_value and
           d.agecode < c.high_value;
```

```
Age Group for each incoming Age Code

                          Age
      id   agecode     Group
_____________________________

       1        18         1
       2        23         1
       4        33         2
       5        30         2
```

Alternatively, the query can be run directly from the *routine_parameter* table as shown in Figure 6. The WHERE clause of this SELECT statement includes a correlated subquery that retrieves the current values from the *routine_parameter* table using the maximum effective date for the selected parameter.

Figure 6

```
LIBNAME ora ORACLE
user=scott password=tiger path=datastore schema=stone;


SELECT     id, agecode,input(r.outcome,2.)'Age Group'
FROM       ora.routine_parameter r, loading.dataset1 d
WHERE      r.parm_code = 30 and
           d.agecode >= r.low_value and
           d.agecode < r.high_value and
           eff_date = (SELECT max(eff_date)
                 FROM   ora.routine_parameter p
                 WHERE  r.parm_code = 30
                        and p.parm_code =r.parm_code
                 );
```

REPORTING

The metadata tables within the data warehouse may be used as the basis for a variety of reports suitable for both business and technical staff. SELECT statements may be issued directly against one or more of the tables providing custom reports for technical staff. Views are often an ideal solution when information must be current but complex queries or formatting is required to generate reports.

Views are referenced in SELECT statements in the same fashion as tables. However, they are updated with the current information in the source tables each time they are accessed. Only the query used to construct the view is stored, and it is rerun when the view is referenced in a SELECT statement. All reports generated from the view will be assigned the labels and formats associated with each column in the CREATE VIEW statement.

In Figure 7 a view is created with two of the tables shown earlier contributing columns. Each time subsequent procedures such as REPORT, TABULATE or SQL are run against the view, the view is generated using the current values in the *current_parameters* and *d_parameter* tables. An embedded LIBNAME statement has been added to the CREATE view statement, allowing an automatic connection to the database at the time the view is accessed.

Figure 7

```
CREATE view report as
(SELECT    r.routine 'Routine', p.parameter 'Parameter',
           p.eff_date 'Effective date',
           p.description 'Description', p.rule 'Rule',
           p.related_table 'Related Table', c.outcome 'Outcome',
           coalesce(c.string_value,put(c.num_value,10.-L)) as value 'Value',
           c.low_value 'Low', c.high_value 'High'
FROM       ora.d_parameter p, ora.current_parameters c, ora.routine r
WHERE      p.parm_code = c.parm_code
           and r.routine_code =p.routine_code)
ORDER BY   r.routine, p.parameter ,c.outcome
USING      LIBNAME ora ORACLE
           user=scott password=tiger path=datastore schema=stone;
```

The SAS Output Delivery System (ODS) is used here to generate a simple html report using the default style. This report can then be distributed to others or published on an intranet site communicating transformation rules and parameters to all involved parties.

Figure 8

```
ODS html;

PROC PRINT data=report noobs label;
      by routine;
      var parameter description related_table outcome value low_value high_value;
      format description $50. ;
      title 'Current transformation parameters';
run;
ODS html close;
```

Current transformation parameters

Routine=Campaign targets

Parameter	Description	Related Table	Outcome	Value	Low	High
age_group	Age groups for segment marketing	routine_parameter	1	.	18	24
age_group	Age groups for segment marketing	routine_parameter	2	.	25	35

Routine=Loyalty targets

Parameter	Description	Related Table	Outcome	Value	Low	High
summer_age	Age range targeted for summer campaigns	routine_parameter	1	.	25	45
summer_age	Age range targeted for summer campaigns	routine_parameter	2	.	46	55

Routine=Specialty targets

Parameter	Description	Related Table	Outcome	Value	Low	High
credit_card	Credit card names to in-house codes	routine_parameter	AMEX	AMERICAN EXPRESS	.	.
credit_card	Credit card names to in-house codes	routine_parameter	MC	MASTER CARD	.	.
gender_code	Recode numeric gender codes to character codes	routine_parameter	F	1	.	.
gender_code	Recode numeric gender codes to character codes	routine_parameter	M	2	.	.

CONCLUSION

Storing metadata within the warehouse is a valuable strategy that is advantageous to business and technical personnel alike.

Although it may take some effort to design an appropriate set of metadata tables, they allow you to embed business and transformation rules directly in the warehouse. Historical and current parameter and business rule documentation and access becomes easier tasks because a single set of tables rather than several documents or programs must be kept up-to-date. The metadata tables provide a simple, consistent method of documenting important aspects of the transformation process.

Metadata tables are created and populated using PROC SQL or Base SAS through a warehouse connection specified on a SAS/ACCESS LIBNAME statement. Transformation routines may read current or historical parameter values directly from metadata tables. Alternatively, views or current parameter tables can be constructed for the routines using PROC SQL.

Performance issues must be considered when embedding ETL-related metadata in the warehouse. Careful table construction, primary key constraints and effective indexing all help to optimize performance. The association of a primary key constraint with one or more table columns results in an automatic index on the constraint columns. Primary key constraints are also an important element of data integrity as they ensure unique entries in the table. Other constraints such as the check constraint also assist in data integrity, allowing the comparison of incoming values against a criterion.

Just as important is the ability to generate reports for business and technical staff from the current parameters and rules accessed by the transformation processes. Base SAS procedures such as PROC SQL and PROC PRINT may

be used to retrieve information from the metadata tables or views built on these tables. The Output Delivery System (ODS) can be used to generate HTML or PDF reports that can easily be shared or published to the company intranet.

It is a strategy worth considering if you find yourself with extensive transformation routines and parameters. Bringing all of the information together into a single set of tables tames the data into a useful record, valuable to all.

CONTACT INFORMATION

Katherine Prairie
Stone Country Solutions Ltd.
730, 736 8th Avenue SW
Calgary, Alberta. T2P 1H4

Phone: (403) 270-3157
Email: kprairie@stonecountry.ca
Web: www.stonecountry.ca