Paper 58-2010

Base SAS® Methods for Building Dimensional Data Models

Christopher W. Schacherer, Clinical Data Management Systems, LLC

ABSTRACT

As the volume of data available from operational systems continues to grow, dimensional models are becoming an increasingly important analytic tool for enterprise reporting and analysis. Although there are a number of software packages specifically designed for transforming data from operational systems into dimensional models, many smaller organizations find themselves unable to make the significant investment in new technology and personnel necessary to utilize these tools. Many of these same organizations, however, do use Base SAS as an analytic tool. Especially for these organizations (but also for larger organizations with more sophisticated business intelligence systems), the current work provides an example of using PROC SQL, SAS/ACCESS®, and SAS hash objects to extract, transform, and load data from an operational system into a dimensional data model.

INTRODUCTION

Across industries, as the volume and complexity of information captured in relation to business processes has continued to grow, the ability of analysts and report writers to extract meaningful information from operational systems has been impacted both by of the complexity of the source data and by performance issues encountered in transforming these data into analytic datasets. Whereas analysts could previously produce analytic datasets with a simple DATA STEP or PROC SQL query against a relatively small number of source system tables (and be able to answer all of the questions that could possibly be answered with those data) now there are often hundreds of database tables that could be used to answer a seemingly endless number of business questions. Moreover, the increasing granularity of these data have resulted in datasets with rows that number in the millions, tens of millions, or hundreds of millions.

One widely accepted approach to dealing with these issues is the creation of Data Marts (or Enterprise Data Warehouses) that focus on modeling data in a way that accurately describes the business processes of interest while optimizing query performance and analytic ease-of-use. Although the classic data warehousing questions revolving around approaches to data warehousing will not be revisited here (see instead, Kimball 1996 and Inmon 1993 for classic data warehousing approaches and Grasse and Nelson (2006), Lupetin (1998), Heinsius (2001), and Rausch (2006) for discussions specific to SAS and star-schemas), the present work focuses on one example of how to make data more usable to analysts and report-writers through the creation of a content-specific data mart that presents data as a dimensional model.

Specifically, the current work describes the creation of a dimensional model used to answer questions related to charges for professional medical services. The source data for our professional billing data mart is centered on a charge transaction table that records each transaction associated with a given billing line-item. Each of these transactions contains a billing id that identifies the specific billing line item to which the current record relates, a transaction type (e.g., new charge, patient copay, charitable reduction, bad debt write-off, etc.), a billing amount relating to the current transaction, and the names of the patient, physician, and the clinic where the service was provided (as well as internal system identifiers that are used to link these names to a fuller description of that entity in the operational system).

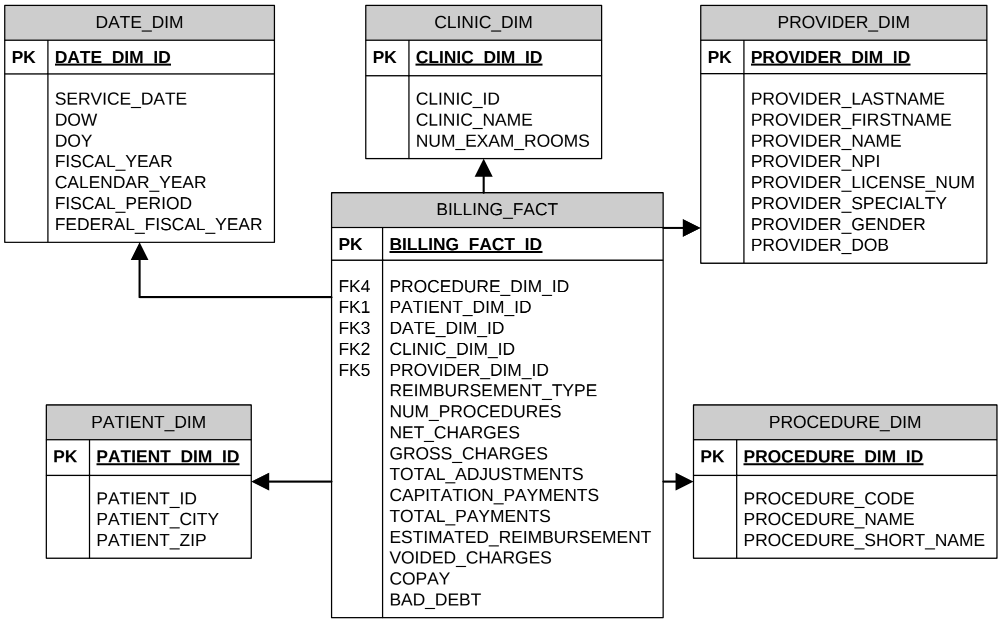
Billing ID	Tran. Type	Transaction Date	Service Date	Patient Name	Provider Name	Billed Amount	Clinic Name	Procedure
81096011	0	10/1/2009	10/1/2009	Smith, John	Jones, Mary	175.20	Clinic XYZ	v70.0
81096011	1	11/12/2009	10/1/2009	Smith, John	Jones, Mary	75.20	Clinic XYZ	v70.0
81096011	2	12/1/2009	10/1/2009	Smith, John	Jones, Mary	10.15	Clinic XYZ	v70.0
81096012	0	10/15/2009	10/15/2009	Smith, John	Stevens, Tom	520.16	Clinic ABC	88.54
81096012	1	11/12/2009	10/15/2009	Smith, John	Stevens, Tom	0.16	Clinic ABC	88.54
81096012	3	12/1/2009	10/15/2009	Smith, John	Stevens, Tom	25.00	Clinic ABC	88.54

As one can see from this example, there is a lot of information stored redundantly (Provider Name, Clinic, etc.). This makes creating reports easier in some ways, because one does not need to join the records from this table to other source system tables to bring (for example) the name of the physician or clinic onto the report. This is especially helpful for inexperienced report writers or those unfamiliar with the larger data model. However, whereas inclusion of these redundant labels can be helpful, the analyst is constrained to the labels made available by the source system. At some point, reports will need to be expanded to include labels not included, by default, in the charge detail table, and at that point, joins to other tables will need to be performed anyway. Furthermore, as the number of records grows into the millions, tens of millions, or hundreds of millions, the cost of this approach in terms of query performance (and even storage space) is significant. This problem is exacerbated by the fact that users may want not only a provider's name included in the analytic dataset, but (for other reporting purposes) their degrees, license numbers, etc. As demand to add additional text labels to this table grows, storage size increases and query performance degrades even further.

Alternatively, analysts and report writers could link to other source system tables (e.g., the "Providers" table, which presumably would store all current data related to a given provider) to return text labels for inclusion on a report, but using these current operational data could result in the loss of information about historical trends. That is, for example, if the new "primary_clinic" on Provider A's "Provider Profile" is Clinic XYZ, and we role up reimbursements by the Clinic at which providers are currently practicing, all of the revenue generated by Provider A would appear to have been generated at Clinic XYZ, even if Provider A spent the last ten years practicing at Clinic ABC. Further, a separate roll-up of the charge transactions table by Clinic would fail to tie-out to that performed using the "primary_clinic" field from the "Provider Profile". This state of affairs can be confusing, and is often regarded with at least a little suspicion even by those who understand how such results come to be in conflict.

The goal of a dimensional model is to minimize this source of confusion and improve query performance when analyzing and reporting on large, complex data models. Dimensional models achieve these goals by transforming the data from the source system into "Fact" tables that contain (a) the numeric performance data of interest (i.e., the facts) such as gross charges, adjustments, and net charges in the current example and (b) foreign keys to tables (the "Dimensions") that describe the context within which the facts were generated. The second part of this description is particularly important because it hints at the historical nature of the dimensions. We did not define the dimensions as "a description of the entity (e.g., the provider) as it is currently described in the operational source system". Instead, we want to describe the provider, clinic, patient, etc. as they would be described at the time the facts were generated. In reality, not all dimension changes are worth tracking—as some characteristics of the dimension change rapidly (generating extraordinarily large dimension tables) and some are not pertinent to our analyses (e.g., a patient's name change). Also, there are dimension fields (e.g., Provider's Name) that we want to keep in synch with the operational source system for reporting purposes. However, when changes to dimensions are important, we must have a way of tracking those changes so that the analysis of the facts is done in a way that is historically accurate.

In addition to how descriptive dimensions are to be captured and used in a dimensional model, another design consideration is the "granularity" of the data. Whereas there are certainly valid reasons for retaining our professional billing data at the finest level of analysis (i.e., the individual <u>line item transaction</u>), the present work will focus on a data model that rolls up the transactions to the level of the charge <u>line-item</u>. In other words, each record in the central "billing_fact" table will represent the aggregate data describing a unique billing line-item. In reality, the more granular data would likely also be preserved in another fact table that uses the same conformed dimensions to add context to the results, but because the vast majority of the reports and analysis done with these data are performed at the billing line-item level (and for simplification of the code examples), the remainder of the paper will focus on creating a dimensional model that has the billing line item as the most granular unit of analysis. This simplified dimensional model is depicted below.



In this data model, each record in the "billing_fact" table contains data (facts) about a specific billing line-item as well as foreign keys that link the fact table records to the appropriate dimension record (e.g., clinic, patient, provider). Using the data in this model, one could easily sum all charges or calculate the reimbursement rate (e.g., total_payments / gross_charges) for a given Provider or Clinic over a given Calendar Year, Fiscal Year or specific date range. Because (a) the relationships between the fact table and the dimensions are defined in terms of simple numeric keys, (b) the contents of the descriptive dimensions are limited to the relatively small number of unique historical dimension records, and (c) the redundant descriptive information has been moved from the performance data of interest and placed in the dimensions, query performance is usually improved significantly compared to the "wider" relational source system table. Further, because the dimension tables are smaller and easier to traverse, one can add a great many additional descriptors to each of these records without impacting performance—increasing the number of ways in which the facts can be sliced or rolled up. For example, instead of just being able to query the relational source system table by the "service date" associated with a billing line-item, the "Date" dimension (DATE_DIM) provides a number of alternative representations of a given date that can be used to summarize data using multiple time descriptors (e.g., Day of Week, Fiscal Quarter, Federal Fiscal Year, etc.) without any additional coding. By providing these additional descriptive attributes to the dimensions, the model enhances the ability of analysts and report writers to provide their customers with a broader range of analytic products that are more likely to remain consistent when performed by different analysts.

In order to achieve the benefits of this powerful model, one must first transform the source system relational model into a dimensional model. The following sections describe how to achieve this transformation using Base SAS DATA STEPS, SAS/ACCESS, PROC SQL, and SAS Hash Objects.

EXTRACTING AND TRANSFORMING SOURCE SYSTEM DATA

As described by Kimball and Ross (2002), there are a variety of different types of fact tables that serve different business purposes. What is described in the following example is an accumulating snapshot of the billing line-items. With each rebuild of the model, all of the transactions that are associated with a given billing line-item are used to update the calculations that result in the facts.

DATABASE ACCESS. The first step in creating the fact table, is extracting data from the source system. In the current example, the source system is an Oracle 10g® database containing an electronic health record vendor's proprietary reporting tables. In order to connect to the database, SAS/ACCESS is used to create a SAS library that references a connection to the database. As described elsewhere (SAS Institute Inc., 1999, 2004; Schacherer, 2008), using native Oracle SQL*Net connections to the database, you can create SAS libraries that directly reference the database schemas to which you have access. The pre-requisites for such connections include having Oracle's SQL*Net client software installed on the server or workstation from which the you are running SAS as well as access to the "tnsnames.ora" file containing the network references and connection parameters for the database(s) to which you are trying to connect.

Having successfully configured Oracle's proprietary networking software, creating the library definition that connects SAS to the database can be accomplished as shown in the following example.

```
LIBNAME billing ORACLE
        USER = hca_etl
        DBPROMPT = yes
        PATH = "ehr"
        SCHEMA = billing_owner;
```

The preceding code creates a library named "billing"; the ORACLE option specifies that the library is referencing an Oracle database. The USER option specifies that the connection will use the credentials of Oracle database user "hca_etl", and the "PATH" specifies the entry within the tnsnames.ora file that points to the database instance to which the user wishes to connect. Regardless of the database system in which the source data are stored, however, once these database-referencing libraries are established, the database tables and views to which the user has access can be used as source data for DATA step programming, SQL procedure queries, or analytic procedures (e.g., GLM, ANOVA, FREQ, etc.).

One final note on database access is provided here for SAS users coming from an environment where database interaction is performed through a vendor's proprietary access tool (e.g., Oracle's SQL*Plus or Microsoft SQL Server's Management Studio). By specifying the PATH and SCHEMA in your library definition, you are constraining the library reference to search that particular database and schema for the database objects you reference. As a result, you cannot directly query a table in another schema through the connection defined by the "billing" library.

Whereas the following query may allow you to access the "patients" table in the "coding" schema using your vendor's proprietary access tool:

```
  SELECT * FROM coding.patients
```

a similar attempt to pull these data through the "billing" library results in an error:

```
PROC SQL;
  SELECT * FROM billing.coding.patients;
QUIT;
```

```
758  PROC SQL;
759  SELECT * FROM billing.coding.patients;
                                 -
                                 22
                                 200
ERROR 22-322: Syntax error, expecting one of the following: a name, ;, (, ',', ANSIMISS, AS,
              CROSS, EXCEPT, FULL, GROUP, HAVING, INNER, INTERSECT, JOIN, LEFT, NATURAL,
              NOMISS, ORDER, OUTER, RIGHT, UNION, WHERE.

ERROR 200-322: The symbol is not recognized and will be ignored.
```

If you are working in an environment where public synonyms are used, you could define the "billing" library without a schema specification:

```
LIBNAME billing ORACLE
        USER = hca_etl
        DBPROMPT = yes
        PATH = "ehr";
```

and you would have access to public synonyms, but you may lose access to database objects in the "billing" schema that do not have public synonyms associated with them.

Therefore, it is recommended that in creating libraries that access relational database systems, you create a library for each database and schema to which you need a connection. This best-practice not only helps avoid issues associated with the design of your back-end security practices, but also helps keep you from being confused about "which" tables (possibly with the same name) you are accessing. For example, there may be a "patients" table in both the billing schema and the ehr schema. In order to get access to both of them, you would create two separate libraries—one for each schema.

```
LIBNAME billing ORACLE                        LIBNAME ehr ORACLE
        USER = hca_etl                                USER = hca_etl
        DBPROMPT = yes                                DBPROMPT = yes
        PATH = "ehr"                                  PATH = "ehr"
        SCHEMA = billing;                             SCHEMA = ehr;
```

INITIAL DATA EXTRACTION. As described previously, this particular dimensional model has, as its main fact table, a table that summarizes all charges, payments, and adjustments applied to each line-item billed for professional medical services. For example, for a given patient visit, there may be multiple procedures performed, and each procedure would generate a line-item on the professional billing statement. Associated with each of these line-items, there will be multiple records describing the original charge transaction (record_type = 0), payments (record_type = 1), voided charges (record_type = 2), copays (record_type = 3), etc. The goal of the current dimensional model is to arrive at a fact table that contains one record for each new charge generated (i.e., each billing line-item) and summarizes the charges, payments, and adjustments associated with that charge.

The initial step in creating the fact table for this model is to create the dataset comprised of all new charges generated (where "record_type = 0"). This will define the granularity of the "billing_fact" table. PROC SQL is used to create a dataset that contains a natural primary key (billing_id, which is part of the complex primary key in the operational system), natural foreign keys (e.g., provider_id, procedure_id) that link the table to lookup tables in the operational system, charge modifiers (which contribute to some of the calculations performed in deriving the facts), gross charges, and placeholders for the summary variables (e.g., "net_charges", "total_adjustments", etc.) and surrogate foreign keys (which will be used to link the fact table records to the dimensions) .

This initial dataset "billing_fact1" will be created in a SAS library "etl" that serves as the work area where the dimensional model will be built prior to being loaded to the business intelligence system as a refreshed version of the Professional Billing model.

```
LIBNAME etl '\\hca\etl\product\probill\data\';
LIBNAME stage '\\hca\etl\dimstage\data\';

PROC SQL;
   CREATE TABLE etl.billing_fact1 AS
   SELECT billing_id, service_date, service_fiscal_year, procedure_id,
          provider_id, patient_id, clinic_id, billing_amount AS gross_charges,
          charge_modifier1, charge_modifier2, charge_modifier3, charge_modifier4,
          . AS payments, . AS voided_charges,
          . AS copays, . AS adjustments,
          . AS net_charge, . AS balance,
          . AS svc_date_dim_id, . AS provider_dim_id, . AS procedure_id,
          . AS clinic_dim_id, . AS patient_dim_id
     FROM billing.billing_details
    WHERE detail_type = 0;
QUIT;
```

After the initial data extract, additional PROC SQL statements are used to create a series of summaries (e.g., payments, voided charges, capitation payments, and bad debt write-offs) that will be applied to the fact table by joining on the billing id.

```
PROC SQL;
   CREATE TABLE etl.payments AS
   SELECT billing_id, SUM(billing_amount) AS payments
     FROM billing.billing_details
    WHERE detail_type = 1
 GROUP BY billing_id;
QUIT;

PROC SQL;
   CREATE TABLE etl.voided_charges AS
   SELECT billing_id, SUM(billing_amount) AS voided_charges
     FROM billing.billing_details
    WHERE detail_type = 2
 GROUP BY billing_id;
QUIT;

PROC SQL;
   CREATE TABLE etl.copays AS
   SELECT billing_id, SUM(billing_amount) AS copays
     FROM billing.billing_details
    WHERE detail_type = 3
 GROUP BY billing_id;
QUIT;

PROC SQL;
   CREATE TABLE etl.adjustments AS
   SELECT billing_id, SUM(billing_amount) AS adjustments
     FROM billing.billing_details
    WHERE detail_type = 4
 GROUP BY billing_id;
QUIT;
```

Once these various summaries are computed, they need to be added to the shell of our "billing_fact" table "billing_fact1". One approach to doing these joins would be to use PROC SQL to perform a left join from the billing fact table to each of the summary tables. With large datasets like the professional billing model, however, each of these joins can take a significant amount of time to complete. To more efficiently perform these joins, SAS hash objects will be used to speed the addition of these summary statistics to their appropriate billing line-item.

USING SAS HASH OBJECTS TO ASSIGN SUMMARY VALUES. As described by Dorfman (2000), Dorfman & Vyverman (2006, 2009), Dorfman & Snell (2003), Snell (2006), and Parman (2006), the SAS hash object provides an extraordinarily fast way to look up data from one dataset and assign those values to records on another data set in the context of the DATA STEP. Using the hash object achieves its performance gains by (1) holding the look-up data in memory—obviating the need for repeated disk access and (2) allowing data to be joined without first being sorted (Secowsky & Bloom, 2007). As described by Secowsky and Bloom, the SAS hash object is:

> *"an in-memory lookup table accessible from the DATA step. A hash object is loaded with records and is only available from the DATA step that creates it. A hash record consists of two parts: a key part and a data part. The key part consists of one or more character and numeric values. The data part consists of zero or more character and numeric values".*

Because the hash records (key/data combinations) are held in memory, finding the data value that corresponds to a given key happens much faster than if the record needs to be read from disk. In the following DATA STEP, hash objects are used to assign the summary values in the tables "payments", "bad_debt", "copay", and "voided_charges" to the table "billing_fact1".

```
DATA etl.billing_fact2;
  IF _N_ = 1 THEN DO;
```

At the beginning of this DATA STEP (where the automatic variable "_N_" equals one), the hash objects are initialized using the DECLARE statement and the keyword "HASH" and are assigned a name (e.g., "payment"). As part of the declaration statement, each hash object is associated with a dataset from which the values that fill the hash table will be read. The key value(s) and data element(s) are then defined for the hash records using the DEFINEKEY and DEFINEDATA methods, respectively. You can think of the key and data elements as two distinct parts of a record in the hash record—two columns in a database table, if you will. At the time a look-up is performed against the hash object, the KEY identifies how the hash table will be searched to identify a match and the DATA represents the element(s) that will be returned as the result of the look-up operation. The declaration of the hash object is completed with the DEFINEDONE method.

```
    DECLARE HASH payment(dataset:'etl.payments');
      payment.DEFINEKEY ('billing_id');
      payment.DEFINEDATA('payment_amount');
      payment.DEFINEDONE();

    DECLARE HASH voided(dataset:'etl.voided_charges');
      voided.DEFINEKEY ('billing_id');
      voided.DEFINEDATA('voided_charges');
      voided.DEFINEDONE();

    DECLARE HASH adjust(dataset:'etl.adjustments');
      adjust.DEFINEKEY ('billing_id');
      adjust.DEFINEDATA('adjustments');
      adjust.DEFINEDONE();

    DECLARE HASH copay(dataset:'etl.copays');
      copay.DEFINEKEY ('billing_id');
      copay.DEFINEDATA('copay');
      copay.DEFINEDONE();

  END;
```

At this point, the structures of the hash objects are defined, and they are filled with data from the specified datasets.

With the hash objects loaded with their KEY and DATA elements, the raw data from the first stage of the fact table build ("billing_fact1") is SET and the FIND method is called for each hash object for each record in the dataset.

```
     DO UNTIL (eof_fact1);
      SET etl.billing_fact1 END = eof_fact1;
          rc1 = payment.FIND();
          IF rc1 ne 0 THEN payments = 0;
          rc2 = voided.FIND();
          IF rc2 ne 0 THEN voided_charges = 0;
          rc3 = adjust.FIND();
          IF rc3 ne 0 THEN adjustments = 0;
          rc4 = copay.FIND();
          IF rc4 ne 0 THEN copay = 0;
```

As each record is read from "billing_fact1", each FIND call uses the values of the KEY variable(s) in the current record to search for a match in the associated hash object. For example, when the call to payment.FIND() is made, the value of "billing_id" from the current record is used to search for a matching "payments" record in the hash object "payment". If a match is found, the value corresponding to the data element of the hash record is returned from FIND() and assigned to the variable "payments" in the current record. Also, the return code rc(i) is assigned a value of "0"—indicating that a match was found. If a match is not found, a non-zero value is returned from the FIND() call—indicating that there were no records in the hash object that had a "billing_id" matching the "billing_id" in the current record. In this case, we set the value of "payments" to 0.

The relationship between the definition of the hash object and the existing dataset variables deserves some additional explanation. Specifically, you cannot name a hash object with the same name as a variable you are going to encounter later in the dataset "billing_fact1". This may not seem obvious at first, but consider what is happening behind the scenes. In the first portion of this DATA STEP, we are defining a SAS OBJECT called "payment". For the duration of the DATA STEP, "payment" now refers to the hash object we created. Therefore, when we open a dataset that contains a variable named "payment", there is a conflict, and we receive the following error:

```
   3248  DO UNTIL (eof_fact1);
   3249    SET etl.billing_fact1 END=eof_fact1;
   ERROR: Variable payment has been defined as both object and scalar.
```

Of course, this error can simply be avoided by making sure variable names found in the dataset being read are not used as hash object names, but the first time this error is encountered, it can be a bit puzzling.

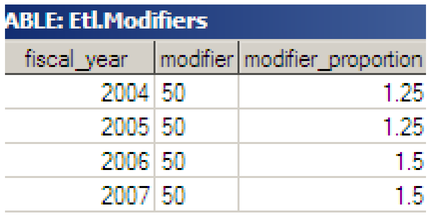
After values for the summary variables are written using hash objects, there are a number of other transformations that we want to perform on our billing line item facts. First, we compute the "net_charges" and "total_adjustments" for each line-item using simple arithmetic expressions.

```
   net_charges = gross_charges + voided_charges;
   total_adjustments = charity_adjustments + other_discounts;
   OUTPUT;
   END;

   RUN;
```

The calculation of some variables, however, is not as straight-forward as the calculation of "net_charges" and "total_adjustments". One example of a value that is somewhat more complex to derive is a "modified_charge" resulting from application of "charge modifiers". Charge modifiers are codes that describe to the payor that the procedure or test performed by the provider was modified in some way, but not to the extent (or in a manner) that warrants a different procedure code be used (e.g., surgically repairing both a broken left arm and broken right arm might appropriately have modifier "50" applied to the procedure billed). Some modifiers, as a result, may be associated with a change to the allowed charge amount—reflecting the special circumstances surrounding that particular procedure. For example, if the criteria are met, the correct submission of a procedure with a modifier "50" may result in 150% of charges being submitted to the payor. Associated with these modifiers, then, are proportions by which charges can be modified. Further, these proportions may change from year to year (or more frequently), but for the present purpose, we will assume that the modifiers change only once per fiscal year. In that case we might have a dataset in our staging area (or a table in our billing system) that contains the columns "fiscal_year", "modifier", and "modifier_proportion".

For modifier "50" the data might look something like the following for fiscal years 2004 - 2007:


ABLE: Etl.Modifiers

fiscal_year	modifier	modifier_proportion
2004	50	1.25
2005	50	1.25
2006	50	1.5
2007	50	1.5

There are a number of ways we could apply these modifier_proportions to the charges on our billing line-items. We could, for example, perform a PROC SQL left join from the "billing_fact2" dataset to the modifiers table on both the "fiscal_year" and the "modifier" or perform a PROC SQL update of the dataset based on the fiscal year and modifier. However, both of those approaches require another set of transactions against our very large dataset. Instead, what we will do is use PROC FORMAT to create a user-defined informat from these data so that the "modifier_proportion" can be applied to our billing line-items as part of the next DATA STEP that our billing fact data will undergo.

FORMATS are instruction sets that tell SAS how to represent data stored in a variable. For example, applying the SAS format MMDDYY10. changes the representation of a SAS date variable to the familiar format "mm/dd/yyyy"—for example, 17850 becomes "11/14/2008". INFORMATS, conversely, are instruction sets that determine how values should be interpreted as they are read into a variable. User-defined formats (and informats) are simply those for which you define your own instructions (e.g., 0 = 'No' / 1 = 'Yes')—or in the case of our modifiers, 50 = 1.25 (but only for fiscal years 2004 and 2005). To create a user-defined informat from our modifiers, we will use the data in the "modifiers" dataset to create a control file that PROC FORMAT will use to create the informat. PROC FORMAT requires that data being submitted for the creation of a format have a specific structure. In our example, we will need to supply a "start", "label", and "fmtname" for each record that will go into creating our "modifier_fmt". In order to create the value that will be used to look-up the appropriate modifier proportion based on the "fiscal_year" and "modifier" code, we define "start" as the concatenation of "fiscal_year", "-", and "modifier". The "label" returned as the interpretation of these concatenated data will be the "modifier_proportion". We also need to specify a name for our format ("modifier_fmt"), and, because we will be creating an informat , we must specify the format "type" as "i". Finally, because we want to assign null values the modifier proportion "1", we create one additional row in our control dataset that specifies nulls should be read-in as the value "1".

```
DATA work.modifier_format;
 SET etl.modifiers end=last;
  start=put(fiscal_year,4.)||'-'||modifier;
  label=modifier_proportion;
  type = 'I';
 fmtname='modifier_fmt'; OUTPUT;
   IF last THEN DO;
      hlo='o';
      start='' ;
      label= 1;
      OUTPUT;
   END;
KEEP start label fmtname type hlo;
RUN;
```

Next, we submit the control dataset "modifier_format" to PROC FORMAT to create the informat "modifier_fmt".

```
PROC FORMAT CNTLIN=work.modifier_format LIBRARY = work;
RUN;
```

We can now use this informat to create the multiplier necessary to calculate our "modified_charge" variable. We do this by using the INPUT and PUT functions with the "modifier_fmt" to convert each of the modifiers from their alphanumeric codes into the appropriate numeric proportion for the fiscal year in which charges were generated.

```
DATA etl.billing_fact2;
 SET etl.billing_fact2;
```

```
    multiplier1 = INPUT(PUT(fiscal_year,4.)||'-'||modifier1,modifier_fmt.);
    multiplier2 = INPUT(PUT(fiscal_year,4.)||'-'||modifier2,modifier_fmt.);
    multiplier3 = INPUT(PUT(fiscal_year,4.)||'-'||modifier3,modifier_fmt.);
    multiplier4 = INPUT(PUT(fiscal_year,4.)||'-'||modifier4,modifier_fmt.);
```

Because we created our informat to read in null values as "1", we can now use the product of all four multipliers as the multiplier that is applied to "net_charge" to produce the "modified_charge".

```
    multiplier = multiplier1* multiplier2 * multiplier3 * multiplier4

    modified_charge = net_charge * multiplier;

   RUN;
```

For example, if the "net_charge" for a given line-item was $100, and there were modifiers that converted to "1", "1.25", ".75", and "1", the "modified_charge" would be $93.75.

Of course, as was mentioned previously, one of our goals in using a user-defined format instead of a PROC SQL SELECT or UPDATE was to avoid traversing our large dataset again just to perform one calculation. In reality, one would either create this format prior to the first DATA STEP and perform the calculation there, or use it in conjunction with a subsequent DATA STEP. Assignment of these multipliers could also be done using a hash object, but in this case, the modifier format is a format that is used elsewhere in this program and in other analytic work performed by the development team, so we chose to reuse it for this purpose.

CREATING INFORMATIVE DIMENSIONS AND ASSIGNING FOREIGN KEYS

One very important DATA STEP for our billing fact data that has yet to be completed is the assignment of the surrogate foreign keys that will link our "billing_fact" table to the descriptive dimensions necessary for its analysis. As discussed previously, dimension tables provide the context for understanding the facts stored in a fact table. These dimensions might be ones that have a corollary in the source system (e.g., Providers) or they might be descriptive attributes of the fact records that are not represented as tables in the source system (e.g., Date). Regardless of the type of dimension, however, it is recommended that for each dimension table there be a single-column, integer primary key that is managed as part of the extract, transform, and load (ETL) process. The sole purpose of this key is to relate the fact table records to their appropriate dimension records. This "surrogate key" (which is the primary key of the dimension table and a foreign key in the fact table) should not be a natural key (i.e., a key that provides some inherently meaningful relationship between the dimension record and the fact record—for example, a foreign key in the operational system).

For example, consider the attributes that might go into a "Clinic" dimension. We might want to collect information about the location, schedule capacity, number of exam rooms and whether or not certain services (e.g., MRIs) are available at a given clinic. Perhaps this information is available in a data table in our source system such that, using the source system's "clinic_id" we can match specific billing line-items to the "clinic" table in the source system. We would be able to determine the characteristics as they exist right now. What we may not be able to know is what those characteristics were at the time of service. Therefore, if we based our dimension tables solely on the data that is currently available in the source system we might not be able to accurately assess, for example, the impact of expanding the number of exam rooms or the clinic schedule capacity on the number of new-patient visits. Once the change is made to the source system table "billing.clinics", that changed value becomes the new descriptor. If, for example, the number of exam rooms at Clinic XYZ was doubled from three to six, it would appear (when joining the "billing_details" to the "clinics" table) that Clinic XYZ always had six exam rooms. Just looking at the available data, we would have no explanation for why the number of clinic visits doubled in that clinic, because looking back across time in our query results, we would see a constant number of exam rooms at Clinic XYZ. In short, by always linking back to the source system descriptors we would lose an opportunity to see the impact of that change on performance at Clinic XYZ.

Instead, we need to somehow capture those changes in our "clinic_dim" dimension table in a way that allows us to link the fact table records to dimension records that describe the clinic as it was at the time of the billed medical service. As we just discussed, we could simply pull the current records from the operational system's "clinics" table and use those data as our dimension—completely replacing the existing "clinic_dim" dimension data with each execution of our ETL process. When performed before the "Clinic XYZ" expansion from three to six exam rooms, this DATA STEP would result in a clinic dimension like Example A; when performed after the expansion, it results in the dimension shown in Example B—and, as discussed above, we would lose information about the changes to the clinic.

```
DATA stage.clinic_dim;
 SET billing.clinics;
RUN;
```

Example A

Clinic ID	Clinic Name	Number of Exam Rooms
0012	Clinic XYZ	**3**
0013	Clinic ABC	9
0014	123 Clinic	6

Example B

Clinic ID	Clinic Name	Number of Exam Rooms
0012	Clinic XYZ	**6**
0013	Clinic ABC	9
0014	123 Clinic	6

Instead of replacing the data in the clinics dimension each time we process the billing data, we use the following code to look for changes and add them to the clinics dimension maintained in our staging area. The first step in this process is to identify the most recent version of the clinics dimension that is in our staging table (Example C)—which we see already contains two records for Clinic XYZ:

Example C ("clinic_dim_stage")

Clinic ID	Clinic ID Surrogate	Clinic Name	Number of Exam Rooms	Effective Date
0012	**1**	**Clinic XYZ**	**6**	**01/01/2002**
0012	**4**	**Clinic XYZ**	**3**	**03/02/2003**
0013	2	Clinic ABC	9	01/01/2002
0014	3	123 Clinic	6	01/01/2002

```
PROC SQL;
 CREATE TABLE stage.max_clinic_records AS
  SELECT * FROM stage.clinic_dim_stage a
   WHERE effective_date = (SELECT MAX(effective_date)
                            FROM stage.clinic_dim_stage b
                           WHERE a.clinic_id = b.clinic_id);
QUIT;
```

The resulting dataset "max_clinic_records" contains the most recent information about each clinic that we have in our staging area.

"max_clinic_records"

Clinic ID	Clinic ID Surrogate	Clinic Name	Number of Exam Rooms	Effective Date
0012	4	Clinic XYZ	3	03/02/2003
0013	2	Clinic ABC	9	01/01/2002
0014	3	123 Clinic	6	01/01/2002

Using this dataset containing the most recent descriptions of the clinics, we then create a dataset "changed_clinics" that contains records from the source system that have a different "num_exam_rooms" value. Note that one can control the changes that are tracked by adding additional "OR" statements to the WHERE clause in the following query. This query will also identify new clinics that are in the operational system but have not yet been added to our staging table.

```
PROC SQL;
 CREATE TABLE stage.changed_clinics AS
 SELECT a.clinic_id,a.clinic_name,a.num_exam_rooms,TODAY() AS effective_date
   FROM billing.clinics a LEFT JOIN stage.max_clinic_records b
     ON a.clinic_id = b.clinic_id
  WHERE a.num_exam_rooms NE b.num_exam_rooms;
QUIT;
```

With all of the new and changed clinic records in the "changed_clinics" dataset, we find the highest value of the surrogate key in our staging table, assign that value to the macro variable "max_surrogate", assign surrogate key values to the new records and insert them into the staging table.

```
PROC SQL NOPRINT;
 SELECT MAX(clinic_surrogate) INTO :max_surrogate
   FROM stage.clinic_dim_stage;
QUIT;

DATA stage.changed_clinics;
 SET stage.changed_clinics;
 RETAIN clinic_surrogate;
 IF _n_ = 1 THEN clinic_surrogate = &max_surrogate + 1;
 ELSE clinic_surrogate = clinic_surrogate + 1;
RUN;

PROC SQL;
 INSERT INTO stage.clinic_dim_stage
(clinic_id,clinic_name,clinic_surrogate,num_exam_rooms,effective_date)
 SELECT clinic_id,clinic_name,clinic_surrogate,num_exam_rooms,effective_date
   FROM stage.changed_clinics;
QUIT;
```

Following the latest expansion of Clinic XYZ, our staging table now contains a third record for Clinic XYZ

Clinic ID	Clinic ID Surrogate	Clinic Name	Number of Exam Rooms	Effective Date
0012	1	Clinic XYZ	6	01/01/2002
0012	4	Clinic XYZ	3	03/02/2003
0013	2	Clinic ABC	9	01/01/2002
0014	3	123 Clinic	6	01/01/2002
0012	**5**	**Clinic XYZ**	**6**	**11/15/2009**

You can see one of the problems with using the source system's primary/foreign key "clinic_id" as the key that joins the billing fact records to the dimensions. Now that we have multiple entries for clinic_id "0012", it is no longer possible to join billing line-item records to their associated clinic record based solely on the foreign key "clinic_id". Instead, we will assign the appropriate surrogate key from the clinic dimension to the records in the billing_fact table and use that value as the foreign key linking the fact table to the clinics dimension.

This process can be done by either (a) performing a PROC SQL update of "billing_fact2" to assign the appropriate surrogate key or (b) creating a separate dataset containing the "billing_fact2" primary key and the appropriate surrogate key and using that dataset to source a hash object that will assign the surrogate key during a subsequent DATA STEP.

In the PROC SQL approach, the "clinic_id" (foreign key from the operational system) and "service_date" in "billing_fact2" are used to define the clinic and date of service for the current billing record. Using these values, a subquery of "clinic_dim_stage" is performed to identify the clinic dimension record with the most recent effective date prior (or equal) to the "service_date". Once identified, that record's surrogate key value is assigned as "clinic_dim_id" on the "billing_fact2" record.

```
PROC SQL;
 UPDATE etl.billing_fact2 A
 SET clinic_dim_id = (SELECT clinic_surrogate
                       FROM stage.clinic_dim_stage b
                      WHERE A.CLINIC_ID = b.CLINIC_ID and
                            b.effective_date =
                           (SELECT MAX(effective_date)
                              FROM stage.clinic_dim_stage c
                             WHERE a.CLINIC_ID = c.CLINIC_ID and
                                   c.effective_date <= a.service_date));
QUIT;
```

In the hash object approach, a similar PROC SQL query is performed to create a table that contains the primary key values from "billing_fact2" and their associated surrogate key—based on the same logic used in the update approach, above. It should be noted here that performing this query as a LEFT JOIN is not necessary because we are going to later use these records to assign the identifed surrogate keys back to the "billing_fact2" table only where an appropriate surrogate key value exists. Those "billing_fact2" records for which an appropriate clinic dimension record is not found, will having surrogate key value "9999999" assigned to them. This surrogate key is associated with a dimension record that defines the clinic as "Unknown", "Missing", or "Not Specified". The important point is that we are not "losing" any records by not performing a LEFT JOIN. Also we are not performing the subquery with an "IN" qualifier on the subquery because we know (based on the unique key constraint on "clinic_dim_stage") that the subquery will result in (at most) a single record.

```
PROC SQL;
 CREATE TABLE etl.CLINIC_DIM_HASH AS
 SELECT a.billing_id ,b.clinic_surrogate as clinic_dim_id
   FROM etl.billing_fact2 a , stage.clinic_dim_stage b
   WHERE a.CLINIC_ID = b.CLINIC_ID AND
         b.effective_date = (SELECT MAX(effective_date)
                              FROM stage.clinic_dim_stage c
                             WHERE a.CLINIC_ID = c.CLINIC_ID AND
                                   c.effective_date <= a.service_date);
QUIT;
```

Once we have this dataset created, we can use it to source a hash object in our last DATA STEP where we will assign the surrogate key value and perform our remaining calculations on the fact records.

```
DATA etl.billing_fact3;
IF _N_ = 1 THEN DO;
  DECLARE HASH clinic_surrogate(dataset:'work.clinic_dim_hash');
    clinic_surrogate.DEFINEKEY ('billing_id');
    clinic_surrogate.DEFINEDATA('clinic_dim_id');
    clinic_surrogate.DEFINEDONE();
END;

DO UNTIL (eof_keys);
 SET etl.clinic_dim_hash (KEEP=billing_id clinic_dim_id) END=eof_keys;
 rc1 = clinic_surrogate.ADD();
END;

DO UNTIL (eof_fact2);
 SET etl.billing_fact2 END=eof_fact2;
 rc1 = clinic_surrogate.FIND();
 IF rc1 NE 0 THEN clinic_surrogate = 999999999;

OUTPUT;
END;
RUN;
```

One should note that both of these approaches should be approached with caution when dealing with dimension tables of significant size (e.g., patients). Because the method described relies on a subquery to assign the appropriate surrogate key, the size of the dimension tables can quickly cause performance issues for this approach. However, for Departments, Clinics, and even Providers, this approach may suite your needs. Regarding the "automaticity" of the identification of changes to the source system tables that source dimensions, it should be noted that the approach described here is not a replacement for business intelligence architects and analysts staying closely in tune with both the operations side of the business and their analytic power-users. Managing dimensions is one of the most important aspects of maintaining a useful dimensional model, and communicating regularly with both the providers and consumers of data is essential to the accuracy of your dimensions.

On that note, it should also be explained that in our staging area, we also include a record in each dimension table with the surrogate key "9999999". The labels assigned to this record denote that "billing_fact" records assigned this value for a given dimension are have "UNKOWN" attributes along that dimension. Therefore, your ETL program should also contain, a report generation stage that identifies the billing records with these unknown attributes and an effort should be made to determine and understand the origins of these missing values. Assigning this missing value code, however, allows analysts and report writers to be able to use inner joins without the fear that they may have accidentally dropped "billing_fact" records.

LOADING DATA INTO THE DATA WAREHOUSE

Once the fact and dimension datasets are created, the data need to be loaded into the data warehouse where analysts and report-writers will use the data to do their work. The data load is accomplished using pass-through SQL to first truncate the existing data tables associated with the professional billing model and then load the newly created datasets into those tables.

When performing multiple pass-through SQL statements to the same database, repeating database connect strings for each step of a process can become confusing and add unnecessary clutter to your program. To avoid this situation, the macro variable "bi_etl" is assigned the value of the connect string for the target system.

```
%LET bi_etl = OLEDB (PROVIDER=SQLOLEDB.1 REQUIRED=Yes
                     USER=ETL_ADMIN DATASOURCE="BI-PROD"
                     PROPERTIES=('initial catalog'=HCA
                                 'Integrated Security'=SSPI
                                 'Persist Security Info'=True)
                     BCP=Yes SCHEMA='BI_OWNER');
```

Next, the program connects to the data warehouse and drops the existing foreign key constraints for this data mart.

```
PROC SQL;
CONNECT TO &bi_etl;
   EXECUTE (
alter table bi_owner.billing_fact drop constraint fk_billing_fact_providers
alter table bi_owner.billing_fact drop constraint fk_billing_fact_dates
alter table bi_owner.billing_fact drop constraint fk_billing_fact_procedures
alter table bi_owner.billing_fact drop constraint fk_billing_fact_patients
alter table bi_owner.billing_fact drop constraint fk_billing_fact_clinics
) BY OLEDB;
```

The existing indexes are dropped to speed the loading of the data. These indexes will be rebuilt after the tables are loaded with the newly modeled data.

```
EXECUTE (
drop index bi_owner.billing_fact.billing_fact_id
drop index bi_owner.billing_fact.providers
drop index bi_owner.billing_fact.dates
drop index bi_owner.billing_fact.procedures
drop index bi_owner.billing_fact.patients
drop index bi_owner.billing_fact.clinics
) BY OLEDB;
QUIT;
```

The data in the existing model are truncated and reloaded from the dimensional model created by the SAS program.

```
EXECUTE (truncate table bi_owner.billing_fact
         truncate table bi_owner.provider_dim
         truncate table bi_owner.procedure_dim
         truncate table bi_owner.patient_dim
         truncate table bi_owner.date_dim
         truncate table bi_owner.clinic_dim
...)BY OLEDB;
QUIT;

PROC SQL;
INSERT INTO bi_owner.billing_fact
       (procedure_dim_id,patient_dim_id,svc_date_dim_id,clinic_dim_id,
        provider_dim_id,reimbursement_type,num_procedures,net_charges,
        gross_charges,total_adjustments,modified_charges,
        voided_charges,copay,bad_debt)
SELECT  procedure_dim_id,patient_dim_id,svc_date_dim_id,clinic_dim_id,
        provider_dim_id,reimbursement_type,num_procedures,net_charges,
        gross_charges,total_adjustments,modified_charges,
        voided_charges,copay,bad_debt
  FROM  etl.billing_fact3;
QUIT;

PROC SQL;
INSERT INTO bi_owner.procedure_dim
       (procedure_dim_id,procedure_code,procedure_name,procedure_short_name)
SELECT  procedure_dim_id,procedure_code,procedure_name,procedure_short_name
  FROM  stage.procedure_dim_stage;
QUIT;

PROC SQL;
INSERT INTO bi_owner.patient_dim
       (patient_dim_id, patient_name, patient_city, patient_zip patient_gender)
SELECT  patient_dim_id, patient_name, patient_city, patient_zip patient_gender
  FROM  stage.patient_dim_stage;
QUIT;

PROC SQL;
INSERT INTO bi_owner.date_dim
       (date_dim_id,service_date,dow,doy,fiscal_year,calendar_year,
        fiscal_period,federal_fiscal_year)
SELECT  date_dim_id,service_date,dow,doy,fiscal_year,calendar_year,
        fiscal_period,federal_fiscal_year
  FROM  stage.date_dim_stage;
QUIT;

PROC SQL;
INSERT INTO bi_owner.clinic_dim
       (clinic_dim_id, clinic_id, clinic_name, clinic_state, clinic_city,
        num_exam_rooms, clinic_director )
SELECT  clinic_dim_id, clinic_id, clinic_name, clinic_state, clinic_city,
        num_exam_rooms, clinic_director
  FROM  stage.clinic_dim_stage;
QUIT;


PROC SQL;
INSERT INTO bi_owner.provider_dim
      (provider_dim_id, provider_id, provider_name, provider_specialty, dea_number,
       license_number)
SELECT provider_dim_id, provider_id, provider_name, provider_specialty, dea_number,
       license_number
  FROM stage.provider_dim_stage;
QUIT;
```

After the new data are loaded, the foreign key constraints are recreated and the data are re-indexed.

```
PROC SQL;
CONNECT TO &bi_etl;

EXECUTE (
alter table bi_owner.billing_fact add constraint fk_billing_fact_providers
 foreign key (provider_dim_id)
```

```
    references bi_owner.provider_dim (provider_dim_id)
   alter table bi_owner.billing_fact add constraint fk_billing_fact_dates
    foreign key (svc_date_dim_id)
    references bi_owner.date_dim (date_dim_id)
   alter table bi_owner.billing_fact add constraint fk_billing_fact_procedures
    foreign key (procedure_dim_id)
    references bi_owner.procedure_dim (procedure_dim_id)
   alter table bi_owner.billing_fact add constraint fk_billing_fact_patients
    foreign key (patient_dim_id)
    references bi_owner.diagnosis_dim (patient_dim_id)
   alter table bi_owner.billing_fact add constraint fk_billing_fact_clinics
    foreign key (clinic_dim_id)
    references bi_owner.clinic_dim (clinic_dim_id)
   ) BY OLEDB;

   EXECUTE (
   create index idx_billing_fact_id on bi_owner.billing_fact(billing_fact_id)
   create index idx_billing_fact_provider on bi_owner.billing_fact(provider_dim_id)
   create index idx_billing_fact_svc_date on bi_owner.billing_fact(svc_date_dim_id)
   create index idx_billing_fact_procedure on bi_owner.billing_fact(procedure_dim_id)
   create index idx_billing_fact_patient on bi_owner.billing_fact(patient_dim_id)
   create index idx_billing_fact_clinic on bi_owner.billing_fact(clinic_dim_id)
   ) BY OLEDB;

   QUIT;
```

With the data re-indexed, the SAS program is finished, and the refreshed professional billing data mart is ready for use.

CONCLUSION

The information provided here has hopefully stimulated your thinking about how to build your own dimensional models using Base SAS, PROC SQL, and SAS hash objects. There are number of related topics that should be considered as the reader begins contemplating his or her own dimensional modeling/ETL project. For example, the ETL program could be scheduled to run daily as an unattended batch job (Kincheloe, 2002; 2006), but this will entail conditionally controlling the execution of your SAS code (see Flavin & Carpenter, 2001). You would hate to have empty SAS data sets at the end of your transformation process and then continue on to truncating your published analytic tables, and "publishing" your empty datasets. For more on automating your ETL program and controlling your processes with audit checks, see Schacherer and Steines (2010). Whether your ETL program is simple or complex, I think you will agree that PROC SQL and SAS DATA STEP programming with hash objects can be used to create fast, efficient ETL programs for transforming relational source system data into dimensional models.

REFERENCES

Borowiak, K. W. (2006). A Hash Alternative to the PROC SQL Left Join. Proceedings of the NorthEast SAS User's Group.

Dorfman, P. M. & Snell, G. P. (2003). Hashing: Generations. Proceedings of the 28th Annual SAS Users Group International Meeting. Cary, NC: SAS Institute, Inc.

Dorfman, P. M. & Vyverman, K.. (2006). Data Step Hash Objects as Programming Tools. Proceedings of the 31st Annual SAS Users Group International Meeting. Cary, NC: SAS Institute, Inc.

Dorfman, P. M. & Vyverman, K.. (2009). The SAS Hash Object in Action. Proceedings of the SAS Global Forum 2009.

Dorfman, P. M. (2000). Private Detectives in a Data Warehouse: Key-Indexing, Bitmapping, and Hashing. Proceedings of the 25th Annual SAS Users Group International Meeting. Cary, NC: SAS Institute, Inc.

Flavin, J. M. & Carpenter, A. L. (2001). Taking Control and Keeping It: Creating and using conditionally executable SAS® Code. Proceedings of the 26th Annual SAS Users Group International Meeting. Cary, NC: SAS Institute, Inc.

Heinsius, B. (2001). Querying Star and Snowflake Schemas in SAS. Proceedings of the 26th Annual SAS Users Group International Meeting. Cary, NC: SAS Institute, Inc.

Inmon,W.H. (1993). Building the Data Warehouse, John Wiley & Sons,Inc.

Kimball, R. (1996). The Data Warehouse Toolkit. John Wiley & Sons, Inc.

Kimball, R. & Ross, M. (2002). The Data Warehouse Toolkit, Second Edition. John Wiley & Sons, Inc.

Kincheloe, F. (2002). While You Were Sleeping - Scheduling SAS Jobs to Run Automatically. Proceedings of the 27th Annual SAS Users Group International Meeting. Cary, NC: SAS Institute, Inc.

Kincheloe, F. (2006). Sleepless in Wherever - Resolving Issues in Scheduled Jobs. Proceedings of the 31st Annual SAS Users Group International Meeting. Cary, NC: SAS Institute, Inc.

Lupetin,Maria,(1998). A Data Warehouse Implementation Using the Star Schema. Proceedings of the 23rd Annual SAS Users Group International Meeting. Cary, NC: SAS Institute, Inc.

Muriel, E. (2007). Hashing Performance Time with Hash Tables. Proceedings of the SAS Global Forum 2007.

Parman, B (2006). How to implement the SAS® DATA Step Hash Object. Proceedings of the SouthEast SAS User's Group.

Rausch, N. (2006). Stars and Models: How to Build and Maintain Star Schemas Using SAS® Data Integration Server in SAS® 9. Proceedings of the 31st Annual SAS Users Group International Meeting. Cary, NC: SAS Institute, Inc.

SAS Institute Inc. (1999). SAS/ACCESS Software for Relational Databases: Reference, Version 8. Cary, NC: SAS

SAS Institute Inc. (2004). SAS 9.1 SQL Procedure User's Guide. Cary, NC: SAS Institute Inc.

Schacherer, C.W. & Steines, T.J. (2010). Building an Extract, Transform, and Load (ETL) Server Using Base SAS, SAS/SHARE, SAS/CONNECT, and SAS/ACCESS. Proceedings of the Midwest SAS Users Group.

Secosky, J. & Bloom, J. (2007). Getting Started with the DATA Step Hash Object. Proceedings of the SAS Global Forum 2007.

Snell, G. P. (2006) Think FAST! Use Memory Tables (Hashing) for Faster Merging. Proceedings of the 31st Annual SAS Users Group International Meeting. Cary, NC: SAS Institute, Inc.

ACKNOWLEDGMENTS

The author wishes to thank Dr. Michelle Detry for reviewing previous drafts of this manuscript.

CONTACT INFORMATION

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

Christopher W. Schacherer, Ph.D.
Clinical Data Management Systems, LLC
6666 Odana Road #505
Madison, WI 53719
Phone: 608.478.0029
E-mail: CSchacherer@cdms-llc.com
Web: www.cdms-llc.com