Paper 145-2011

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, 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, the increasing volume and complexity of data generated by business processes impacts the ability of analysts and report writers to extract meaningful information from operational systems. 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 a large proportion of the questions that could possibly be answered with those data), now source systems often involve hundreds or thousands of database tables that could be used to answer a seemingly endless number of business questions. Moreover, the increasing granularity of these data has 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), Heinsius (2001), Lupetin (1998), and Rausch (2006) for discussions specific to SAS and star-schemas], the present work focuses on one example of creating and maintaining a dimensional model using Base SAS techniques.

Specifically, the current work describes the creation of a dimensional model of billing activity associated with professional medical services. The main data source for this example is a charge transaction table that records each transaction (e.g., new charge, payment, charitable reduction, etc.) associated with billing line-items (e.g., medical procedures, tests, etc.). Each of these transactions contains a billing id that identifies the specific billing line-item, a transaction type (e.g., new charge, patient copay, etc.), a billing amount, and the names of the patient, physician, and clinic involved in the episode of care.

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

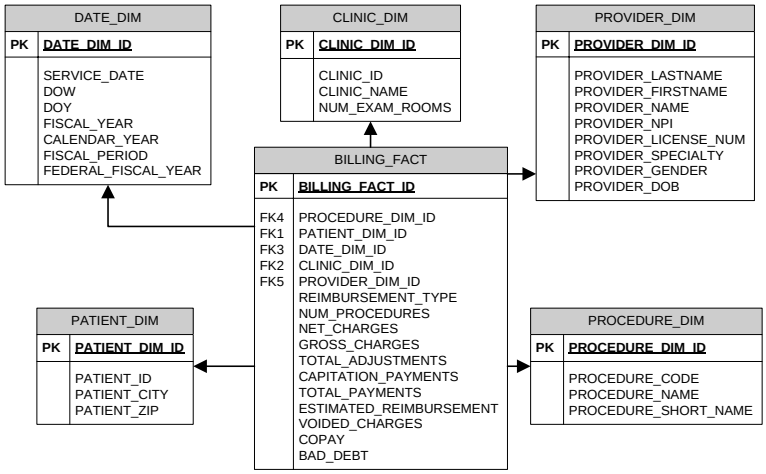
As one can see from this example, there is a lot of information stored redundantly (e.g., Provider Name, Clinic Name, etc.). This makes creating reports easier in some ways, because one does not need to join records from this table to other source system tables in order to bring (for example) the name of the physician or clinic onto the report. This is especially helpful for inexperienced report writers or those unfamiliar with the larger data model, but as the number of records grows into the millions, tens of millions, or hundreds of millions, the cost to query performance for storing these redundant data becomes significant. This problem is exacerbated by the fact that users may want their reports and analytic datasets to include not only a provider's name but also 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) to return text labels for inclusion on a report, but doing so could lead to inconsistencies across analysts as similar data elements (with subtle or not-so-subtle differences) are queried from different source tables to answer the same

question. In addition, use of current operational data could result in the loss of information about historical trends. For example, if the primary clinic on Provider A's provider profile is Clinic XYZ and we roll up reimbursements by primary clinic, 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. A separate rollup of the charge transactions table by clinic would fail to tie-out to the query performed using the primary clinic field from the provider profile. This state of affairs can be confusing and is often regarded with skepticism even by those who understand the reasons for such conflicting results.

The goal of a dimensional model is to eliminate the data model as a 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 comprised of records that contain (a) the numeric performance data of interest (e.g., gross charges, adjustments, and net charges associated with each billing line-item) and (b) foreign keys to tables (the "Dimensions") that describe the context within which the facts were generated. The latter part of this description is particularly important because it hints at the historical nature of the dimensions. Dimensions are not simply a description of the entity (e.g., provider) as it is currently described in the operational source system. Instead, dimensions should accurately describe the provider, clinic, patient, etc. at a point in time so that the facts can be understood in terms of their characteristics at the time the facts were generated.

In addition to building and maintaining dimensions in a way that enables valid analysis of the fact table, another design consideration for building a dimensional model is the granularity of the data in the fact table(s). Whereas there are certainly valid reasons for retaining our professional billing data at the finest level of analysis (i.e., the individual transaction), the present work will focus on a data model (depicted below) that rolls up the transactions to the level of the billing line-item. In addition to the aggregated and computed values (facts) assigned to each billing line-item, each record in the "billing_fact" table also contains foreign keys that link each fact table record to its corresponding record in each of the dimension tables.



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. Query performance is improved significantly compared to the wider charge transaction table because: (a) redundant descriptive information has been removed from the organizational performance data of interest and placed in the dimensions, (b) the contents of the descriptive dimensions are limited to the relatively small number of unique historical dimension records, and (c) the relationships between the fact table and the dimensions are defined in terms of simple numeric keys. Further, because the dimension tables are smaller and easier to traverse, one can add many additional descriptors to each of these records without degrading performance—increasing the number of ways in which the facts can be sliced or rolled up.

In order to achieve the benefits of this powerful model, one must first transform the source system's relational data 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 fact table types, each of which serves 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 calculated 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 database connection. As described elsewhere (SAS Institute Inc., 1999, 2004; Schacherer, 2008), using native Oracle SQL*Net connections to the database, one can create SAS libraries that reference a database schema. The pre-requisites for such connections include installation of Oracle's SQL*Net client software on the server or workstation on which you are running SAS, access to the "tnsnames.ora" file containing the network references and connection parameters for the database(s), and database privileges appropriate for accessing the data.

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 invokes the SAS/ACCESS engine for Oracle and creates the SAS library "billing". The USER parameter specifies that the connection will be made using the credentials of Oracle database user "hca_etl", and the PATH parameter specifies the connection is being made to database instance "ehr". Finally, the SCHEMA parameter specifies the logical collection of database objects that will be referenced by the SAS library. Once this database connection is 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.).

INITIAL DATA EXTRACTION. As described previously, each record in the fact table ("billing_fact") summarizes the charges, payments, and adjustments applied to a professional billing line-item. For example, for a given patient visit there may be multiple procedures or tests performed, and each of these billable items generates a line-item on the professional billing statement. Each of these line-items, in turn, is associated with one or more records in the charge transactions table—for example, new charges (transaction_type = 0), payments (transaction_type = 1), voided charges (transaction_type = 2), etc. Therefore, the initial step in creating the fact table for this model is to create a dataset comprised of all new charges—those in which "transaction_type" equals zero.

The resulting dataset ("billing_fact1") contains a natural primary key ("billing_id"), natural foreign keys (e.g., "provider_id", "procedure_id", etc.) used to link dataset records to their associated records in the operational system's supporting tables, one aggregated fact ("gross charges"), placeholders for additional aggregated facts (e.g., "net_charges", "total_adjustments", etc.) and placeholders for surrogate foreign keys (e.g., "provider_dim_id", "clinic_dim_id", etc.) that will be used to link the fact table records to the dimensions.

This initial dataset is created in the SAS library "stage" that serves as the work area where the dimensional model will be built prior to being loaded to the business intelligence system. The SAS library "etldat" contains persistent dimension data (used later in the program) as well as the audit data associated with each execution of the extract, transform, and load (ETL) program.

```
LIBNAME stage '\\hca\etl\product\probill\data\';
LIBNAME etldat '\\hca\etl\persistent\';

PROC SQL;
   CREATE TABLE stage.billing_fact1 AS
   SELECT billing_id, service_date, service_fiscal_year, procedure_id,
          provider_id, patient_id, clinic_id, billing_amount AS gross_charges,
          . AS payments, . AS voided_charges, . AS copays,
          . AS net_charges, . AS balance, . AS total_adjustments
          . AS svc_date_dim_id, . AS provider_dim_id, . AS procedure_dim_id,
          . AS clinic_dim_id, . AS patient_dim_id
     FROM billing.charge_transactions
    WHERE detail_type = 0;
QUIT;
```

After the initial data extraction, additional PROC SQL statements are used to create summaries (e.g., payments, voided charges, and copayments) that roll up the other types of transactions associated with each billing id.

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

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

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

Once these summary variables are computed, each needs to be added to the "billing_fact1" record with the corresponding "billing_id". One approach to performing 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 ones associated with the professional billing model, however, each of these joins could take a significant amount of time to complete. To more efficiently perform these joins, SAS hash objects are used to speed the assignment of these summary variables to their associated billing line-item.

USING SAS HASH OBJECTS TO ASSIGN SUMMARY VALUES. As described by Dorfman (2000), Dorfman & Snell (2003), Dorfman & Vyverman (2006, 2009), Parman (2006), and Snell (2006), the SAS hash object provides an extraordinarily fast way to assign values from one dataset to corresponding records in another dataset in the context of the DATA step. Hash objects achieve their 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 object entries (key/data combinations) are held in memory, finding the data value that corresponds to a given key happens much faster than it would if the records were read from disk. In the following DATA step, hash objects are used to assign the summary values in the tables "payments", "voided_charges", and "copays" to the aggregated fact placeholders in "billing_fact1".

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 (e.g., stage.payments) from which the values used to fill the hash object will be read. The key and data elements are then defined for the hash objects 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 object. At the time a look-up is performed against the hash object, the KEY values are searched for a match to the current record being processed, and if a match is found, the data element associated with that key will be returned as the result of the look-up operation. The declaration of the hash object is completed with the DEFINEDONE method. For the three hash objects in this example, "billing_id" is defined as the key value and the summary variables "payments", "voided_charges", and "copays", respectively, are defined as the data elements. It should be noted that although each of these hash objects has only one key variable and one data variable, hash objects can also be defined to use complex keys (i.e., those comprised of multiple variables) and to return values for multiple variables.

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

  DECLARE HASH payment(dataset:'stage.payments');
    payment.DEFINEKEY ('billing_id');
    payment.DEFINEDATA('payments');
    payment.DEFINEDONE();

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

  DECLARE HASH copay(dataset:'stage.copays');
    copay.DEFINEKEY ('billing_id');
    copay.DEFINEDATA('copays');
    copay.DEFINEDONE();
END;
```

With the hash objects defined, data from the first stage of the fact table build ("billing_fact1") are processed. As each record is read from "billing_fact1", the FIND method is used to perform a search of each hash object for entries having a key value that matches the value of "billing_id" in the current record. 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 "billing_id" in the "payment" hash object. If a match is found, the value corresponding to the data element of the hash record is assigned to the variable "payments" in the current record and the return code variable "rc1" 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 entries in the hash object that had a "billing_id" matching the "billing_id" of the current record. In this latter case, we set the value of "payments" to 0—indicating the total amount of the payments received for that billing line-item.

```
  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 = copay.FIND();
       IF rc3 ne 0 THEN copay = 0;
```

After using the hash objects to add the aggregate facts to the billing line-item records, additional facts are calculated in the same DATA step and the records are output to "billing_fact2".

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

In the actual production code there are a number of other calculated variables produced during this DATA step, but this example demonstrates that in a single pass through the dataset you can both use hash objects to look up values from multiple datasets and perform calculations and transformations involving the dataset variables.

Before leaving this discussion of the hash object, there are two additional points that deserve mention. First, when creating a hash object, you cannot assign it the same name as a variable that will be encountered in the source dataset—or anywhere in the DATA step, for that matter. This may not seem obvious at first; after all, the "payment" hash object is used to look up payments, so why not name it "payments" as well? Consider what would happen if we did this. The hash object "payments" would be declared just as "payment" was in the previous example.

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

Following this declaration the name "payments" refers to the "payments" hash object. Later in the program, when "billing_fact1" is SET as a source dataset, SAS attempts to define the dataset variable "payments" and an error is generated because "payments" is already defined as the name of the hash object.

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

Of course, this error can be easily avoided by not using the names of source dataset variables as hash object names, but the first time this error is encountered it can be a bit confusing. It should also be noted that even if there was not a variable named "payments" in "billing_fact1" a DATA step using this hash object would still generate errors because the hash object's data element, once defined, is also a DATA step variable.

A second, very important, consideration in the use of hash objects is the size of the hash object being created relative to the available physical memory. Because hash objects reside in memory, the size of the hash table is limited to the memory available to SAS. If the hash object exceeds the available memory, an error similar to the following will be generated:

```
ERROR: Hash object added 16777200 items when memory failure occurred.
FATAL: Insufficient memory to execute DATA step program.
       Aborted during the EXECUTION phase.
NOTE: The SAS System stopped processing this step because of insufficient memory.
NOTE: There were 1 observations read from the data set STAGE.BILLING_FACT1.
WARNING: The data set STAGE.BILLING_FACT2 may be incomplete.  When this step was
         stopped there were 0 observations and 23 variables.
```

As described in the SAS 9.2 Help and Documentation, the amount of memory (in bytes) used to hold the data associated with each entry in a hash object can be determined using the ITEM_SIZE attribute of the hash object. In the following example, each entry in the "payment" hash object requires 32 bytes of memory.

```
    DECLARE HASH payment(dataset:'stage.payments');
      payment.DEFINEKEY ('billing_id');
      payment.DEFINEDATA('payments');
      payment.DEFINEDONE();
...
    hash_size = payment.ITEM_SIZE;
    PUT 'hash size:' hash_size;
END;
```

```
NOTE: There were 20000000 observations read from the data set STAGE.PAYMENTS.
hash size:32
```

Multiplying the hash size by the number of rows in the hash object's source dataset will give a rough estimate of the memory required for the hash object (Secowsky & Bloom, 2007), but this method will underestimate the hash object size because ITEM_SIZE accounts only for the memory needed to store the data and does not account for the overhead required by the hash object. However, SAS Institute, Inc. (2008) provides a macro (%hash_test) that more accurately predicts the memory usage of a hash object with given characteristics. Another quick way to estimate the size of a hash object is to create a smaller subset of the dataset that sources the hash object and execute the DATA step in which the hash object is created with the FULLSTIMER system option enabled (OPTIONS FULLSTIMER;). This option will provide statistics on the amount of memory used in creating the smaller hash object, and one can use these data to estimate the memory required for the full hash object.

In the previous example, there was sufficient memory for all three hash objects to be loaded, and the DATA step that produced "billing_fact2" successfully added the aggregate facts and calculated fields to the fact table records. Following assignment of the facts to each record, variables that serve as the foreign keys linking the fact table to its descriptive dimensions must be assigned the appropriate value for each record. Before that assignment can occur, however, maintenance of the dimension tables must be performed.

CREATING INFORMATIVE DIMENSIONS AND ASSIGNING SURROGATE FOREIGN KEYS

Dimension tables provide the context for understanding the facts stored in a fact table. Some dimensions have a corollary in the source system (e.g., the "provider_dim" dimension and the "providers" table), whereas others are not represented as tables in the source system (e.g., "date_dim"). Regardless of the type of dimension, however, it is recommended that each dimension table have—as its primary key—a single-column, integer variable that is

managed as part of the ETL process and has no relation to a record identifier or operational code in the source data system. Not only does the use of these surrogate keys help avoid the potentially disastrous impacts of unexpected changes to the operational system keys, but their use also facilitates the management of dimensions that change over time.

As an example of the latter benefit, consider the clinics dimension ("clinic_dim"); it contains information about the location, schedule capacity, and number of exam rooms associated with each clinic. This information is available in a table in the source system, and using the source system's primary key ("clinic_id"), the billing line-item records could be linked to the current description of each clinic in the source system. Therefore, one could use this operational table as the clinics dimension and be able to answer questions about the performance of clinics in terms of their current characteristics. However, this approach presents a challenge to analyzing the performance impact of changes to the clinics. Once a change is made to the operational system's "clinics" table (e.g., an increase in the number of exam rooms at Clinic XYZ), that changed value would become the new descriptor for all billing records associated with that clinic. The number of exam rooms associated with medical services provided at Clinic XYZ prior to the change would be lost, and the impact of this change on performance could not be readily analyzed. Instead, we need to somehow capture such changes in our "clinic_dim" dimension in a way that allows us to link the "billing_fact" records to records that describe the clinic as it was at the time the billed service was performed.

Instead of using the operational system's "clinics" table as the dimension table, the clinics dimension ("clinic_dim") is built and maintained in a way that allows us to capture certain changes to the attributes describing each clinic.

clinic_dim

Clinic ID	Clinic Dim ID	Clinic Name	Num Exam Rooms	Effective Date
0012	**1**	**Clinic XYZ**	**2**	**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

Starting from a baseline copy of the operational system's "clinics" table, the following code executes as part of the ETL process to look for changes in the clinic descriptions and adds records reflective of those changes to the clinics dimension ("clinic_dim"). The first step in this process is to identify the most recent description of each clinic in the current clinics dimension.

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

The resulting dataset "max_clinic_records" contains the most recent information about each clinic from the clinics dimension.

"max_clinic_records"

Clinic ID	Clinic Dim ID	Clinic Name	Num 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, the next step in updating the clinics dimension is to create a dataset ("changed_clinics") that contains: (a) those records from the source system that have a different value for "num_exam_rooms" than the most recent record for that clinic in the dimension table and (b) new records (associated with new clinics). Note that one can control the changes that are tracked in the dimension by adding additional OR statements to the WHERE clause in the following query.

```
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, the macro variable "max_surrogate" is assigned the value corresponding to the highest value of the surrogate key in the clinics dimension. Surrogate key values are then assigned to the "changed_clinics" records, and these new records are inserted into the clinics dimension.

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

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

PROC SQL;
 INSERT INTO etldat.clinic_dim
(clinic_id,clinic_name,clinic_dim_id,num_exam_rooms,effective_date)
 SELECT clinic_id,clinic_name,clinic_dim_id,num_exam_rooms,effective_date
   FROM stage.changed_clinics;
QUIT;
```

Following the latest expansion of Clinic XYZ from three exam rooms to six, the clinics dimension now contains a third record for Clinic XYZ.

Clinic ID	Clinic Dim ID	Clinic Name	Num Exam Rooms	Effective Date
0012	1	Clinic XYZ	2	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**

This view of the clinics dimension highlights the problem inherent in using the source system's primary keys as the keys that link the billing fact records to the dimensions. Because there are multiple entries for clinic_id "0012", it is not possible to unequivocally match billing line-item records to the correct clinics dimension record based solely on the natural foreign key "clinic_id". Instead, the surrogate key value ("clinic_dim_id") from the clinics dimension needs to be assigned to each record in the billing fact table based on the date of service and the effective date of the clinics dimension records.

In the following example, "clinic_id" and "service_date" in "billing_fact2" are used to identify the record in the clinics dimension that accurately reflects the clinic characteristics at the time the billing line-item was generated. A correlated subquery of the clinics dimension ("clinic_dim") 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 the value of "clinic_dim_id" on the "billing_fact2" record.

```
PROC SQL;
 UPDATE etl.billing_fact2 A
 SET clinic_dim_id = (SELECT clinic_dim_id
                        FROM etldat.clinic_dim b
                       WHERE a.clinic_id = b.clinic_id and
                             b.effective_date =
                            (SELECT MAX(effective_date)
                               FROM etldat.clinic_dim c
                              WHERE a.clinic_id = c.clinic_id and
                                    c.effective_date <= a.service_date));
QUIT;
```

With the surrogate foreign key values assigned to the billing fact table records, the dimensional model is complete. It should be noted, however, that in each dimension table there is also a record with the surrogate key value "9999999". The values assigned to variables in these records denote that "billing_fact" records assigned this surrogate foreign

key have "UNKOWN" attributes along that particular dimension. If you use this approach to identify missing data in your fact table, 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 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 fear that they may accidentally drop "billing_fact" records.

LOADING DATA INTO THE DATA WAREHOUSE

Once the fact and dimension datasets are created, the data are loaded into a Microsoft SQL Server 2005® database where the analysts and report-writers access the data model.

```
LIBNAME hca OLEDB PROVIDER=SQLOLEDB.1 DATASOURCE="BI-PROD"
        PROPERTIES = ('initial catalog'=HCA 'Integrated Security'=SSPI
                      'Persist Security Info'=True)
        BCP = Yes SCHEMA = 'BI_OWNER';
```

Several steps of the load process are accomplished using pass-through SQL. 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 the 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  DATASOURCE="BI-PROD"
              PROPERTIES = ('Initial Catalog'=HCA
                            'Integrated Security'=SSPI
                            'Persist Security Info'=True) );
```

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 new data.

```
EXECUTE (
drop index bi_owner.billing_fact.billing_facts
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.

```
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;
```

At this point the tables in the professional billing data mart are empty. The following PROC SQL insert statements are used to load data from the staging area to the analytic data mart.

```
PROC SQL;
INSERT INTO hca.billing_fact
       (billing_fact_id,procedure_dim_id,patient_dim_id,date_dim_id,
        clinic_dim_id,provider_dim_id, gross_charges, payments, voided_charges,
        copays, net_charges, balance, total_adjustments)
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  stage.billing_fact2;
QUIT;

PROC SQL;
INSERT INTO hca.procedure_dim
       (procedure_dim_id,procedure_code,procedure_name,procedure_short_name)
SELECT  procedure_dim_id,procedure_code,procedure_name,procedure_short_name
  FROM  etldat.procedure_dim;
QUIT;

PROC SQL;
INSERT INTO hca.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  etldat.patient_dim;
QUIT;

PROC SQL;
INSERT INTO hca.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  etldat.date_dim;
QUIT;

PROC SQL;
INSERT INTO hca.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  etldat.clinic_dim;
QUIT;

PROC SQL;
INSERT INTO hca.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 etldat.provider_dim;
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 truncate your published analytic tables, and "publish" your empty datasets. For more information 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

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.

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.. (2009). The SAS Hash Object in Action. Proceedings of the SAS Global Forum 2009. 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.

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.

Grasse, D. & Nelson, G. (2006). Base SAS vs. SAS Data Integration Studio: Understanding ETL and the SAS Tools Used to Support it. Proceedings of the 31st 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. (2006). Sleepless in Wherever - Resolving Issues in Scheduled Jobs. Proceedings of the 31st Annual SAS Users Group International Meeting. Cary, NC: SAS Institute, 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.

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.

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. (2008). Sample 34193: How to determine how much memory my hash table will require. Downloaded February 13, 2011 from: http://support.sas.com/kb/34/193.html

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

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

Schacherer, C.W. (2008). Utilizing SAS as an Integrated Component of the Clinical Research Information System. Proceedings of the SAS Global Forum 2008. 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. Cary, NC: SAS Institute, Inc.

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.

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