Paper 57-2010

Building an Extract, Transform, and Load (ETL) Server Using Base SAS®, SAS/SHARE®, SAS/CONNECT®, and SAS/ACCESS®

Christopher W. Schacherer, Clinical Data Management Systems, LLC
Timothy J. Steines, Independent Consultant

ABSTRACT

There are a number of methods that can be used to efficiently build dimensional models for business intelligence systems using BASE SAS®, SAS/STAT®, and SAS/ACCESS®. Once developed, these programs must be run on a platform that provides the level of performance required to rebuild data models in ever-shortening build windows. The current paper describes how to build the infrastructure necessary to support resource-intensive processes such as extract, transform, and load (ETL) programs using SAS Server®, Base SAS®, SAS/SHARE®, SAS/CONNECT®, and SAS/ACCESS®. In addition to providing a robust infrastructure for quickly building dimensional models, the programming methods used to take advantage of this infrastructure can also increase the reliability of the ETL process and enhance the quality of the data in the data warehouse.

INTRODUCTION

The business intelligence department of a physician group practice recognized the desirability of creating a dimensional model of their professional billing system and developed a SQL-script approach to transform the relational data model in the operational system into a dimensional model loaded into their data warehouse. In this initial approach, sequentially executed SQL data manipulation language (DML) statements were executed in order to derive interim staging tables that incrementally approached the architecture of the desired dimensional model—for example, by summarizing different types of charges and credits applied to individual billing line-items and then joining those summary values to a master list of billing line-items. Although this approach was successful when it was first developed, it involved the creation of a large number of interim staging tables, required nearly 10,000 lines of code to transform the data, and as the amount of data being modeled increased, required 36+ hours to complete.

A dimensional modeling solution needed to be developed that would not only be scalable but could also be easily applied to other data models (e.g., appointment scheduling, clinic operations, etc.). The initial SAS programming solution provided a simple and readily replicable solution, but was being run on the developer's workstation. The next step in the evolution of this program was to enable it to run on a platform that would allow it to scale to meet the demands of ever-growing source data. The current work illustrates how a robust infrastructure was built to support the SAS programming solution developed for creating dimensional models using SAS Server, Base SAS, SAS/ACCESS, SAS/CONNECT, AND SAS/SHARE.

THE SAS INFRASTRUCTURE

The most obvious way to improve performance of the SAS program is to run it on an enterprise-class server. Performance gains for any SAS program would naturally be expected in the areas of I/O and processor speed, but especially important for creating the dimensional model was the increased available memory of the server. The dimensional modeling program (Schacherer, 2010) relied heavily on SAS HASH objects to achieve its performance gains, and the size of a HASH object is bound by the available physical memory on the machine where SAS is being run.

INITIAL SAS SERVER SETUP. Although a number of configuration changes were applied in developing the ETL server, development of the solution began with a simple installation of SAS Server 9.1.3 SP4 on a 64-bit Windows 2003 Server®. Although installation of the SAS Server is relatively simple, there was one unexpected complication. After installing our Oracle client, we were still unable to connect to the Oracle 10g® source database via SAS/ACCESS. After searching the SAS KnowledgeBase, we discovered that attempting to connect to Oracle server, using Oracle 10.2.0.1 or earlier, could result in an error if SAS is installed in a directory that contains parentheses in the directory name--for example, C:\Program Files (x86)\SAS (SAS Institute, Inc., 2006A). Once SAS was uninstalled and reinstalled in a different directory, we were able to connect to the source system via SAS/ACCESS.

SAS/CONNECT. With the server software installed, the next step to enable remote users to submit jobs to the server is to configure SAS/CONNECT to receive remote processing requests. As described elsewhere (SAS Institute, Inc., 2004A, 2004B, 2006B; Stokes, 2003, 2005; Sadof, 2005) SAS/CONNECT allows SAS to be run remotely across different platforms and to allow programs to utilize resources found on other machines on the network.

Once installed, SAS/CONNECT must be configured for use—in this case, by installing the SAS spawner as a service on the host computer. Installing the SAS spawner creates a Windows service called (by default) "SAS Job Spawner". The service, when started, listens for requests from remote (or local) SAS clients, and when these requests are properly authenticated, the SAS spawner starts a SAS session to perform the requested processing. To install the SAS Job Spawner in Windows®, open a DOS command window and execute the following command (SAS Institute, Inc, 2006B). In this example the "-i" switch tells spawner.exe to "install" the SAS Job Spawner and the "-c" switch specifies that the "communication" method to be used in making connections is tcp/ip. To see the full set of switches use the "-h" switch.

```
C:\Program Files\SAS\SAS 9.1\spawner.exe –i –c tcp
```

Once the SAS Job Spawner service is installed and configured on the host, there is one additional system change that needs to be made before SAS/CONNECT can be used to initiate a SAS session from a remote host. Prior to this step, the spawner service is "listening", but no requests can get to it, because no port has been configured to receive these connections. In order to correct this situation, an entry similar to the following is made in the "services" file located (by default) in "**C:\Windows\system32\drivers\etc\services**":

```
SAS Job Spawner    23/tcp                          # SAS Spawner
```

This entry indicates that the SAS Job Spawner service will be using port 23 to receive requests via TCP/IP—the same port specified for the telnet service in this case. Once this entry is saved, SAS/CONNECT is ready to receive requests to launch a SAS session through the spawner service. [Note: Connections to SAS/CONNECT can also be made via telnet; for more information on connection methods see SAS Institute, Inc (2004B).]

To confirm that you are able to launch a SAS session on the host, you can execute the following code to make a remote request for processing on the server:

```
FILENAME rlink 'C:\Program Files\SAS\SAS 9.1\connect\saslink\tcpwin.scr';
%LET node=10.1.20.88 23;
OPTIONS COMAMID=tcp REMOTE=node ;
SIGNON rlink;

RSUBMIT;

DATA TEST;
 DO i = 1 TO 100;
    varx = RANNOR(12345);
    OUTPUT;
 END;
RUN;

ENDRSUBMIT;
SIGNOFF;
```

This connection example described extensively elsewhere (SAS Institute Inc. 2004A; Stokes, 2005) begins by identifying the location of the tcpwin.scr file as FILENAME "rlink". The tcpwin.scr file is a script that controls how the SAS client attempts to connect to the remote host and initiate a session. In response to a request to sign-on to a remote host, the default tcpwin.scr file presents the user with prompts for a userid and password (unless these values are hardcoded in the sign-on request or within a modified tcpwin.scr file). After providing valid credentials for the host system, the connection is completed.

The macro variable "node" is assigned the ip address (or DNS name) of the host machine and the port on which the spawner is listening. The OPTIONS specified in the example are the communication method (COMAID) and the network location of the spawner (REMOTE). The macro variable "node" is used to specify the location because the value of REMOTE must be an acceptable SAS name of eight characters or less. Even though "node" resolves to the full network address of the spawner, it passes the name-length test prior to being resolved and is accepted as a value for "REMOTE". With the connection parameters specified, the SIGNON command is issued to initiate the connection to the host. Once connected, the RSUBMIT command submits the subsequent SAS statements to the host system. When the execution of the submitted SAS statements is finished control returns to the client machine, and the SAS session is terminated with the SIGNOFF command.

SAS/SHARE. With SAS/CONNECT installed and configured, client machines can now start a SAS session on the server and submit code to be executed using server resources. However, in the ETL program, there are LIBNAME statements that connect SAS to two databases used to source the dimensional model. When running the program in a local SAS session, the user can provide the appropriate connection credentials either by specifying "dbprompt=yes" in the LIBNAME statement or by creating a library that is configured to connect at startup (Schacherer, 2008). However, these options are not viable in the SAS/CONNECT client-server infrastructure described here because (1) the remotely submitted specification of "dbprompt=yes" does not return a database prompt back to the client machine and (2) libraries configured to connect at startup on the host are generally unavailable to remote sessions being initiated through SAS/CONNECT.

Further, whereas it is likely that there are programmatic ways to prompt the user for their password and then send the response as part of the RSUBMIT code block, there are additional reasons to not pursue this method when building an ETL server in the manner described here. First, one of the end goals of this solution is to have these jobs run unattended as scheduled batch jobs. Second, if one could standardize the library names across all ETL jobs, doing so would reduce maintenance difficulties inherent to programmers developing their own naming conventions. The solution that helped achieve both of these goals was to utilize SAS/SHARE to define these commonly used libraries to all ETL developers initiating SAS sessions on the server.

As described in detail elsewhere (SAS Institute Inc., 2004C) SAS/SHARE allows remote users to utilize libraries defined on the SAS/SHARE server. Users connecting to the SAS Server, once their session has been launched, will have access to the libraries made available through SAS/SHARE.

Similar to the configuration of SAS/CONNECT, once SAS/SHARE is installed, the host system must be configured to recognize the SAS/SHARE server as a service and specify the port through which remote clients can connect to the service. This is done, again, by adding the following entry to “C:\Windows\system32\drivers\etc\services”:

```
MEDSHARE      5010/tcp                # SAS/SHARE SERVER
```

In this case, "MEDSHARE" is the logical server name assigned in the SAS program "share server initiatiator.sas" that is used in this example to start the SAS/SHARE server and assign the shared LIBNAMES.

```
COMAMID=tcp SET=tcpsec _secure_;
```

This program begins by specifying the communication method used to make a connection to the SAS/SHARE server. In this example, tcp security (tcpsec) is specified as "_secure_" which requires users attempting to connect to the shared libraries to be authenticated by the local host in order to access the libraries. The following lines were added to create a datestamped log file every time the SAS/SHARE server is started; this log, in turn, can be used to track activity against the shared connections (e.g., connections to the SAS/ACCESS database connections.)

```
%LET share_log = sas share server_%SYSFUNC(PUTN(%SYSFUNC(DATE()),YYMMDD10.)
                 )_%SYSFUNC(TRANSLATE(%SYSFUNC( PUTN(%SYSFUNC(TIME()),TIMEAMPM12.)
                 ),.,:));
DATA _NULL_;
  CALL SYMPUT('log_file', "'d:\SAS Share\" || RESOLVE('&share_log') ||".txt'");
RUN;

PROC PRINTTO LOG=&log_file NEW;
RUN;

LIBNAME billing ORACLE
            USER = 'bi_etl'
        PASSWORD = '{sas001}<encrypted_password>'
            PATH = "billing"
          SCHEMA = billing;

%LET curr_dt = %sysfunc(datetime(),datetime.);
%PUT &curr_DT;

LIBNAME HCA OLEDB Provider=SQLOLEDB.1 required=Yes
        User=BI_ETL PASSWORD='{sas001}<encrypted_password>' datasource="BI-ETL"
        properties=('initial catalog'=HCA 'Persist Security Info'=True)
        bcp=Yes schema='DBO';
```

```
%LET curr_dt = %sysfunc(datetime(),datetime.);
%PUT &curr_DT;

PROC SERVER ID=medshare NOALLOC;
RUN;
```

Once these initial steps are executed, the libraries to be shared with SAS/CONNECT users are defined, a log entry is made to identify the time when each library definition is established, and the SAS/SHARE server name (medshare) is assigned with the NOALLOC option (which restricts users from creating their own shared libraries on the server). It should be noted that the SAS/SHARE server is only available as long as the SAS session from which it was launched remains open. When that SAS session closes, the SAS/SHARE server is no longer available.

It should also be noted that the program that launches the SAS/SHARE server utilizes hard-coded, encrypted values for the passwords associated with a restricted user IDs. Although the practice of hard-coding passwords should be avoided when sharing programs on a network or across a development team, it was used here (we believe, safely) because the user IDs used to connect to the databases are constrained to provide read-only access from a single IP Address—that associated with the ETL server. Further, only the server administrator has access to modify (or even view) the file, and only the encrypted version of the password is stored anywhere on the server. For an excellent treatment of the issues involved in using and storing passwords, see Sherman & Carpenter (2009).

In order to ensure that the SAS/SHARE server starts every time the ETL server is started, the SAS Service Configuration Utility (SSCU) was used to create the service "SAS SHARE Server", and the following command was entered as the Service Path for the service:

```
"C:\Program Files\SAS\SAS 9.1\sas.exe"
-sysin "D:\SAS SHARE\share server initiatiator.sas"
```

In addition to providing remote users access to the libraries defined in "share server initiatiator.sas ", this approach has the added benefit of enforcing library naming standards across the development team. In the production environment, the only LIBNAMES that will give client programs access to the data are those defined on the SAS/SHARE server. Therefore, developers will use those LIBNAMES consistently and the maintenance difficulty inherent to developers each developing their own naming standards is minimized. During development, individual programmers will use their personal credentials to connect to these libraries, but when the code is migrated to production, the LIBNAME statements will reference the SAS/SHARE server to identify the library to which they are connecting.

For example, a call to the billing database in the development version of the program might look something like:

```
LIBNAME billing ORACLE
            USER = 'CSchacherer'
        PASSWORD = 'MNVikings_4'
            PATH = "billing"
          SCHEMA = billing;
```

But once the code is moved to the test and production environments (in which the SAS/SHARE server is available, the calls to that same database reference the SAS/SHARE server connection.

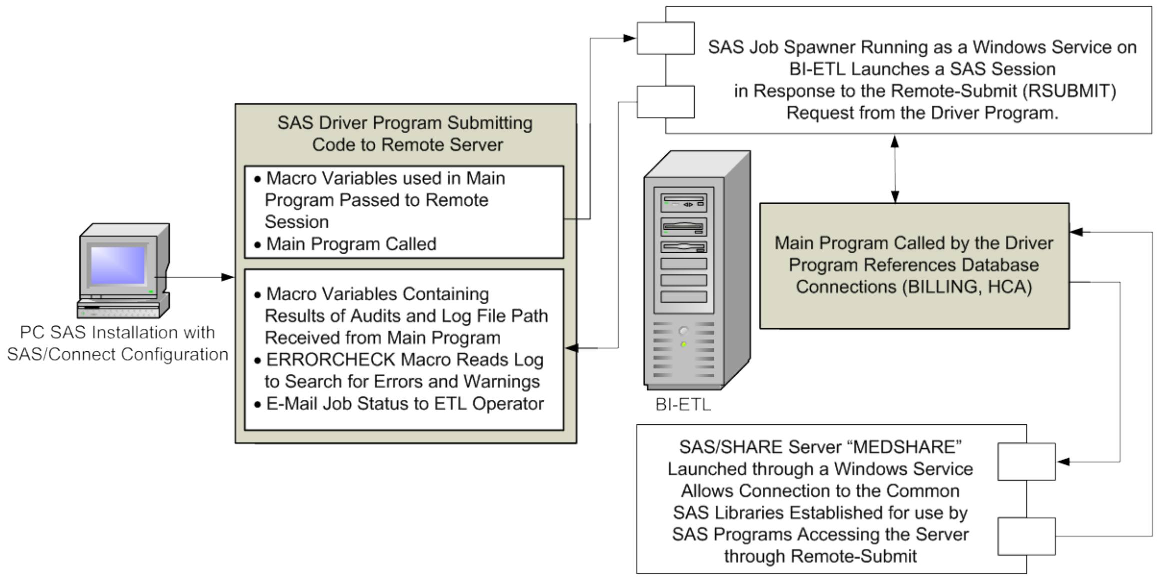
```
LIBNAME billing SERVER=medshare;

LIBNAME practce SERVER=medshare;
```

With this new infrastructure in place, all of the ETL developers on the team can now utilize the higher performing server resources to process their SAS jobs and they utilize a common set of LIBNAMES across all of their programs. In addition to providing access to these higher performing resources, however, this infrastructure affords the opportunity to have better control of processes—inserting process-checks and audits that allow more sophisticated data integrity checks prior to loading data to the production decision support system.

PROGRAMMING FOR THE NEW INFRASTRUCTURE

One of the challenges in scheduling production jobs is making sure that in the event of an error, processing is controlled effectively so as to not yield unintended production data (see Flavin & Carpenter, 2001 for an overview of techniques). For example, in the case of loading a fact table to a data mart, if the dataset to be loaded is empty for some reason (e.g., database connectivity failure, change to a source system table name, etc), you would not want processing to continue to the point where the previously loaded data are truncated and the new data are not available to refresh the data model. The ETL program must be able to recognize failures, act accordingly, and notify the operator that a failure occurred. The SAS/CONNECT infrastructure is ideal for supporting this functionality.

In order to develop such controls over the ETL process, we split the SAS program into two parts: (1) a driver program (Fecht, 2008) and (2) the main program that performs the data manipulation necessary to produce the dimensional model. As depicted in the following figure, the driver program calls the main program—passing to the main program parameters that will be used to define the ETL job being performed. The main program, using the libraries defined on the SAS/SHARE server, extracts data from the defined library connections and performs the transformations and data loads for which it was developed. When the Main program has finished running, control is passed back to the Driver program. It is important to note, that control is returned to the Driver program regardless of whether the Main program executed successfully. The methodology described here takes advantage of that fact in order to reliably determine the success or failure of the ETL job. Because these two programs are running in two separate SAS Sessions, the Driver program can evaluate the LOG of the Main program to determine if errors were encountered during the processing of the Main program. In addition, the Main program can send audit information about the dimensional data model being generated back to the Driver program and this information can be delivered via e-mail to the ETL operator.


SAS Job Spawner Running as a Windows Service on BI-ETL Launches a SAS Session in Response to the Remote-Submit (RSUBMIT) Request from the Driver Program.
SAS Driver Program Submitting Code to Remote Server
• Macro Variables used in Main Program Passed to Remote Session
• Main Program Called
• Macro Variables Containing Results of Audits and Log File Path Received from Main Program
• ERRORCHECK Macro Reads Log to Search for Errors and Warnings
• E-Mail Job Status to ETL Operator
PC SAS Installation with SAS/Connect Configuration
Main Program Called by the Driver Program References Database Connections (BILLING, HCA)
BI-ETL
SAS/SHARE Server "MEDSHARE" Launched through a Windows Service Allows Connection to the Common SAS Libraries Established for use by SAS Programs Accessing the Server through Remote-Submit

DRIVER PROGRAM –SERVER CONNECTION & MAIN PROGRAM CALL. After assigning options such as the Universal Naming Convention (UNC) name of the location of the macros used by the Driver program, the Driver program begins with the assignment of macro variables that either (a) drive processing (e.g., begin and end dates for the period for which the Main program extracts data) or (b) are associated with the management of the ETL operation (e.g., the e-mail address of the ETL operator overseeing the process).

```
LIBNAME macstor '\\bi-etl\Macros';
OPTIONS MSTORED SASMSTORE=macstor;

%LET recipients = "etloperator@cdms-llc.com"
```

Next, SYMPUTX is used to assign a number of date variables that control processing and capture information used to measure the performance of the ETL process.

```
DATA _NULL_;
  CALL SYMPUTX('extract_period_begin',"'01JUN2001'D");
  CALL SYMPUTX('extract_period_end',''''||put(intnx('month',DATE(),0)-
1,date9.)||''''||'D');
  CALL SYMPUTX('process_start_time',DATETIME());
RUN;
```

Then, an email macro is used to let the ETL operators know that this particular job has begun.

```
%email(process_name = 'Begin Processing: ProBill',
            message = 'The Professional Billing Model build has been initiated.',
             tolist =(&recipients));
```

With the parameters for our job established, the Driver program connects to the SAS Server...

```
FILENAME rlink 'C:\Program Files\SAS\SAS 9.1\connect\saslink\tcpwin.scr';
%LET node=10.1.20.88 23;
OPTIONS COMAMID=tcp REMOTE=node;
SIGNON;
```

Because the connection being made to the remote host is launching a new SAS session on that remote host, at this point there are no macro variables defined within that session. The Main program will have no way of knowing the values defining the beginning and ending dates of the data extract period. Using %SYSLPUT, the values of the previously defined macro variables are passed to the new session on the remote host so they can be used by the Main program. The "d_" notation is used simply to designate that these are macro variables being assigned in the Driver program.

```
%SYSLPUT d_extract_period_begin = &extract_period_begin;
%SYSLPUT d_extract_period_end = &extract_period_end;
%SYSLPUT d_process_start_time = &extract_date;
```

Next, the request is made to the remote host to begin processing the main program that contains the ETL logic.

```
RSUBMIT;

%INCLUDE '\\bi-etl\SAS ETL Programs\MEDSHARE Professional Billing – Main.sas';

ENDRSUBMIT;
SIGNOFF rlink;
```

When the Main Program has finished running (either following a successful completion or because of errors), control returns to the Driver program which disconnects from the host.

MAIN PROGRAM. The Main program needed to be altered in only a few small ways in order to run in this new infrastructure. First, as a result of using SAS/SHARE to assign and manage library assignments, the reference to SAS/SHARE server "medshare" replaced the connect strings that defined these database connections.

```
LIBNAME billing SERVER=medshare;

LIBNAME practce SERVER=medshare;
```

A second change necessitated by this client-server implementation was the need to use UNC names for identifying local libraries (local, that is, to the remote host where the Main program is run). This change was made for two reasons. First, as the server team may remap drive arrays from time to time in the data center, using a UNC name helps us avoid accidentally losing access to our library because a volume gets mounted as "Drive D" instead of "Drive E". Second, and more importantly, the directory permissions assigned to allow remote client access are available to the client only as a UNC name; the remote client has no access to the drive letter assigned by the HOST system.

```
LIBNAME etl '\\bi-etl\SAS ETL Programs\Professional Billing\SAS Datasets'
```

A third change that was introduced by the new methodology involved the passing of macro variables from the Driver program to the Main program. At the beginning of the Main program these variables are written out to the log as a record of the input parameters used in this particular execution of the ETL job.

```
%PUT d_extract_period_begin;
%PUT d_extract_period_end;
%PUT d_process_start_time;
```

The remaining changes to the Main program were implemented to automate audit-checks and failure notifications performed by the Driver program. When the stand-alone version of the Main program was first implemented, the part of the program that did the extraction and transformation of the source system data was run separately from the part of the program that performed the truncation of the data warehouse tables and the loading of the new data. In between these steps, a series of (somewhat manual) audit checks were performed to confirm that the dollars associated with billed services did not grow or shrink unexpectedly in a given month, that the dollars associated with actual vs. expected reimbursement did not fluctuate unexpectedly, and that the number of billing line-items neither shrank nor grew in a manner that would lead us to suspect that changes to the source system might be threatening the veracity of our analyses and reports. Similarly, there was a search of the log performed to ensure that no errors or warnings were generated by the ETL process. The goal of automating the ETL programs clearly could not be attained if there was not a means of automating these quality controls.

At the most basic level, maintaining control over a process requires knowing the outcome of that process (i.e., did the process succeed or fail?). When control is returned to the Driver program, we need to know whether it was because the Main program finishing successfully or because failures in the execution of the Main program resulted in no additional processing being performed. One way to assess this fundamental outcome is to analyze the log for "error" and "warning" messages. When the program was run as a stand alone program, this was not possible within the same SAS session because the log could not be programmatically read while the session running the program was still open. However, with the Driver program running in another session the Main program's session log can be written out to a text file and an error-processing macro called from the Driver program can read the log and determine if errors were encountered in the execution.

```
%LET logpath = \\bi-etl\ETL Programs Logs;
%LET etl_program = professional_billing;
```

In order to enable this functionality, the location and name of the log file are assigned to the macro variables &logpath and &etl_program, respectively. Next, a datestamped name for the current execution of the Main program is assigned to the macro variable datestamped_log.

```
%LET datestamped_log = &etl_program._%SYSFUNC(PUTN(%SYSFUNC(DATE()),YYMMDD10.)
                        )_%SYSFUNC(TRANSLATE(%SYSFUNC(PUTN(%SYSFUNC(TIME()
                        ),TIMEAMPM12.)),.,:));
```

Then the full path to the log file is assembled and assigned as the value of macro variable log_file_path.

```
DATA _NULL_;
CALL SYMPUT('log_file_path', "'" || RESOLVE('&logpath') || "\SASLOG_" ||
RESOLVE('&SAS_LogDateTimeStamp') ||".txt'");
RUN;

PROC PRINTTO LOG=&log_file_path NEW;
RUN;
```

The macro variable &log_file_path, in turn, is used in the subsequent PROC PRINTTO statement to begin generation of the SAS session's log to a datestamped file similar to the following

\\bi-etl\ETL Program Logs\SAS_Log_professional_billing_2009-09-09_10.45.00 PM.txt

Once the name and location of this session's log file is assigned to a macro variable, %SYSRPUT can be used to send this macro variable back to the Driver program's session in the same manner used by %SYSLPUT to send macro variables from the Driver session to the Main program's session.

```
%SYSRPUT m_log_file_path = &log_file_path;
```

The Driver program's session now has the location and name of the .LOG file to be analyzed. When control is returned to that session an %errorchecking macro will be used to check the log for errors and report them to the ETL operator (see Haworth, 1997; Smoak, 2002 for examples of building an error-checking macro).

Retrospectively identifying that an error occurred, however, will be of little use if the Main program continues past those errors to the section of the program in which the target database is truncated and reloaded. Imagine what would happen if the source system for our model was unavailable for some reason. The very first extract in the Main program would fail, there would be datasets created without any records in them and then, left unchecked, the program would proceed to the step where the target database tables are truncated and loaded with...no records. The next morning analysts determined to get a report out before noon would be completely paralyzed. Yes, the Driver program would alert the ETL operator that there was a failure in the process, but there would still be a significant delay in delivery of the data for the dimensional model.

For this reason, we chose to configure the Main program to stop SAS immediately upon encountering an error. This approach was implemented by adding the option to use STRICT error checking with an ERRORABEND response should the ETL process encounter an error.

```
OPTIONS ERRORCHECK=STRICT ERRORABEND;
```

With this option specified, the error that causes the process to abend will be captured in the log—which will be analyzed and communicated to the ETL operator when control returns to the Driver program.

Even in the absence of true system errors there are reasons one might want to halt the execution of the main program. If, for example, the source system had experienced a mishap whereby billing detail records were lost, miscoded, or duplicated, we would not only want to identify this situation and suspend our own processing, but we would also like to be able to alert the stewards of that source system that we suspect there is a problem in their system or processes. Therefore, after the dimensional modeling process is complete (but before we truncate and load the target system), several audits are done to confirm the integrity of the data.

One very simple example of such an audit would be the comparison of the number of records in a given fact table or dimension table produced by the previous and current executions of the Main program. The first step in this process is to determine, from the previous audit data, the number of records loaded to the specified table during the last run of the program.

```
PROC SQL;
 SELECT record_count INTO :billing_record_count_prev
   FROM audits.record_counts a
  WHERE table_name = 'billing_detail_fact' and
        load_date IN (SELECT MAX(load_date)
                        FROM audits.record_counts
                       WHERE table_name = a.table_name);
QUIT;
```

After this record count is assigned to the macro variable :&billing_record_count_prev, the corresponding record count is derived from the dataset that has been prepared for loading into the data warehouse.

```
PROC SQL;
 SELECT COUNT(*) INTO :billing_record_count_curr
   FROM profbill.billing_detail_fact;
QUIT;
```

The current and previous record counts are then compared to see if the current month's data falls within the tolerance levels determined by previous experience with these data. If the number of records in the current extraction of the data are below 95% or above 105% of the number of records from the previous month, the macro variable &audit_027 is assigned the value of '1'; if the number of records falls within the tolerance range, &audit_027 is assigned the value '0'.

```
DATA _NULL_;
 IF &billing_record_count_curr < .95*(&billing_record_count_prev) OR
    &billing_record_count_curr > 1.05*(&billing_record_count_curr) THEN
    CALL SYMPUTX('audit_027',1);
  ELSE CALL SYMPUTX('audit_027',0);
RUN;
```

After all such audit values are calculated, their values are summed and assigned to the macro variable &audit_total.

```
DATA _NULL_;
CALL SYMPUTX('audit_total',&audit_001 + &audit_002 + ... &audit_027);
RUN;
```

The &audit_total macro variable is then passed back to the Driver program as &m_audit_total.

```
   %SYSRPUT m_audit_total = &audit_total;
```

If there are no audit failures encountered (i.e., if the value of &audit_total equals zero) then the macro variable &terminator is assigned the value of null ('') and SAS continues on to the next section of the program where the truncation and loading of the target system tables occurs. If, instead, the Main program encounters audit failures, then &terminator is assigned the value "ENDSAS;" and when this macro variable is resolved, it will end the SAS session associated with the Main program. Importantly, however, information about the audit failures has already been passed back to the Driver program before this evaluation.

```
   DATA _NULL_;
    IF &audit_total = 0 THEN
    CALL SYMPUT('terminator','');
    ELSE
    CALL SYMPUT('terminator','ENDSAS;');
   RUN;

   &terminator
```

DRIVER PROGRAM–ERROR ASSESSMENT & NOTIFICATION. When the Main program has finished executing due to (a) successful completion, (b) syntax errors causing the program to abend, or (c) audit values that exceed tolerances, control passes back to the Driver program—with the Main program having passed to the Driver session the macro variables identifying the name and location of the Main program's session log (&m_log_file_path) and the number of audit failures experienced in the current run of the Main program (&m_audit_total).

The Driver program then calls the %errorcheck macro to process the log file specified by &log_path, write the lines containing errors to an error report in the same directory as the log, and assign a value indicating the number of errors found in the log (&errorcount).

```
   %errorcheck(&m_log_file_path);
```

With both the number of errors and the number of audit failures collected, the following _null_ datastep assigns a value to the macro variable "emailvar" depending on the number of errors and audit failures associated with the Main program's execution. The macro variable, when resolved, will execute the macro &email with the appropriate message to the ETL operators (and other interested parties) identified in the "%recipients" list.

For example, if the Main program fails with both syntax errors and audit violations, an e-mail describing this situation is sent.

```
   DATA _NULL_;

   IF &errorcount > 0 and &auditmetric > 0 THEN
      CALL SYMPUT('emailvar',
                 "%" || "email(loadname= 'Error: Professional Billing ETL Process',
                 message= 'The ProBill Model ETL process encountered both Syntax
                           and Audit failures.The ETL process failed and the
                           decision support production data were not replaced.

                           Please investigate and re-execute the ETL process.',
                  tolist=(" || RESOLVE('&recipients') ||  "),
                  attach=" || "(" || RESOLVE('&ErrorLocation_ForEmail') ||
                         RESOLVE('&AuditLocation_ForEmail') || ")" || ")");
```

If both of those conditions are not met, the Driver program checks to see if the Main program ended with errors, and sends the appropriate message.

```
   ELSE IF &errorcount > 0 THEN
      call symput ('emailvar',
                 "%" || "email(loadname= 'Professional Billing ETL Process',
                  message= ' The ProBill Model ETL process encountered processing
```

```
                              ERRORS', The ETL process failed and the
                              decision support production data were not replaced.
                              Please investigate and re-execute the ETL process.'
                    tolist=(" || RESOLVE('&recipients') ||  "),
                    attach=" || RESOLVE('&ErrorLocation_ForEmail') || ")");
```

There are additional checks made for other possible combinations of values, but the message that the ETL Operator prefers to see is that there were neither audit violations nor errors.

```
ELSE IF &auditmetric = 0 and &errorcount = 0 THEN
   CALL SYMPUT ('emailvar',
               "%" || "email(loadname= 'Success: ProBill ETL Process',
                message= 'ProBill ETL process finished successfully',
                tolist=(" || RESOLVE('&recipients') ||  "),
                attach= )" );
```

With the "emailvar" macro variable set up to execute the %email macro and send one of the assigned messages, the appropriate call to the %email macro is executed by resolving the &emailvar macro variable.

```
&emailvar;
```

AUTOMATING AS A SCHEDULED BATCH JOB

With the infrastructure in place and the Driver and Main program methodology developed, the final step of implementing this solution is to schedule the job to run unattended on the server. Although the preceding sections of the paper have discussed running the Driver program as a "client/server" program, there are no changes required in either program in order to schedule the solution to be run completely on the server. The Driver and Main programs will still be running in separate sessions and the Driver program will still submit the Main program for processing via SAS/CONNECT. The Main program will still use the libraries defined on the SAS/SHARE server, and the two programs will still pass macro variables back and forth using SYSLPUT and SYSRPUT. The only difference is that now the Driver and Main programs will both be running on the same computer—the ETL server.

Scheduling the Driver program to run as a batch job on the server was accomplished using the Windows Scheduler®. When scheduling the Driver program event in Windows Scheduler, the following is entered in the "Run" field. SAS is identified as the executable, and the –CONFIG AND –SYSIN switches identify the SAS configuration file to use and the Driver program to execute.

```
C:\PROGRA~1\SAS\SAS9~1.1\sas.exe -CONFIG "C:\Program Files\SAS\SAS
9.1\nls\en\SASV9.CFG" -sysin "d:\SAS Programs\ProBill Driver.SAS"
```

The scheduled event is run under the credentials of a system user (ETL_ADMIN) that has been granted the privileges necessary to read from all directories associated with the scheduled ETL jobs and write to the directories where the SAS datasets and logs are written. This user's credentials have also been hard-coded into a modified TCPWIN.SCR file with an encrypted password in order to allow auto-sign-on via SAS/CONNECT.

ETL Developers are also assigned read/write privileges to these same directories in case they still need to manually remote-submit their code to re-execute jobs following audit- or error-related failures. However, individual ETL Developers are not allowed the ability to write to the repository of production SAS programs. Instead, developers must follow a solution development lifecycle that results in change management procedures performed by the ETL server administrator. This administrator is responsible for assuring that all standard operating procedures related to promoting code to the production environment have been followed. Following their review the new production Driver and/or Main program is promoted to the production environment.

CONCLUSION

Whereas tools built solely for ETL programming clearly have advantages in terms of their ability to catalog meta-data, perform drag-and-drop transformations, and provide a number of "out-of-the-box" alert and error-processing functions, the current work demonstrates how organizations already licensing SAS Server, Base SAS, SAS/ACCESS, and SAS/CONNECT can build a robust ETL server with existing resources. Moreover, the solution described in the current work can easily be adapted to a wide variety of resource intensive programming tasks—from simulating econometric or biologic models to monitoring and analyzing network traffic.

REFERENCES

Flavin, J. M. & Carpenter, A. L. (2001). Taking Control and Keeping It: Creating and using conditionally executable SAS® Code. Proceedings of the Twenty-Sixth SAS User's Group International.

Haworth, L. (1997). Reports Based on SAS Output: Taking Advantage of PROC PRINTTO, DATA STEPs and PROC GPRINT. Proceedings of the Twenty-Second SAS User's Group International.

Heinsius, B. (2001). Querying Star and Snowflake Schemas in SAS. Proceedings of the Twenty-Sixth SAS User's Group International.

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

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

Lupetin,Maria,(1998). A Data Warehouse Implementation Using the Star Schema. Proceedings of the Twenty-Third Annual SAS User's Group Internantional.

Rausch, N. (2006). Stars and Models: How to Build and Maintain Star Schemas Using SAS® Data Integration Server in SAS® 9. Proceedings of the Thirty-First SAS User's Group International.

SAS Institute Inc. (2004A). SAS/CONNECT 9.1: User’s Guide. Cary, NC: SAS Institute Inc.

SAS Institute Inc. (2004B). Communications Access Methods for SAS/Connect 9.1 and SAS/SHARE 9.1. Cary, NC: SAS Institute Inc.

SAS Institute Inc. (2004C). SAS/CONNECT 9.1: User’s Guide. Cary, NC: SAS Institute Inc.

SAS Institute Inc. (2006A). Problem Note 17669: ORA-12154: TNS could not resolve service name.

SAS Institute Inc. (2006B). TS-DOC: TS-638 - Installing and Starting a PC Spawner on Windows Operating Systems.

Schacherer, C. W. (2010). Base SAS Methods for Building Dimensional Data Models. Proceedings of the Midwest SAS Users Group.

Sherman, P. D. & Carpenter, A. L. (2009). Secret Sequel: Keeping your password away from the LOG. Proceedings of the SAS Global Forum 2009.

Smoak, C. G. (2002). A Utility Program for Checking SAS ☐Log Files. Proceedings of the Twenty-Seventh SAS User's Group International.

Stokes, J. C. (2003). SAS/CONNECT®: The Ultimate in Distributed Processing. Proceedings of the Twenty-Eighth SAS User's Group International.

Stokes, J. C. (2005). Remote Compute Services Simply Stated. Proceedings of the Thirtieth SAS User's Group International.

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