Paper 274-27

Passing the buck

David Johnson FRSA, DKV-J Consultancies, Holmeswood, England

ABSTRACT

A recent customer project involved selecting records from a DB2 data mart. The data mart was unavailable for many months. This meant early development of the SAS® application was done on the Windows platform. SAS tables were modelled on the anticipated data mart structure.

Those tables satisfied the early system testing requirements for the AF application that formed the User Front End. The AF application was object based, and the data interface objects worked effectively in the LAN environment.

On the mainframe environment however, the different architecture meant some additional design flexibility was needed. The application libraries included macro libraries that were customised for each Operating System (O/S). The application code then transparently accessed those objects. This allowed the application to be migrated from the Windows development platform to the MVS production platform.

This paper explores the strategy used in designing and implementing the object libraries. The extensive tools of the Windows API in the O/S interface had to be matched with similar tools on MVS. The process wasn't simple, but the object approach made for easier delivery, and was founded on a successful and trusted methodology. The client was rewarded with an application that migrated efficiently across Operating Systems.

THE APPLICATION

The application would allow the user to set up specific conditions to define the records to make up a group. These conditions would be applied against a data mart on another host, which would also provide reporting and analysis functions. The unavailability of the data mart meant a model of the data was needed on the local client to allow development.

What we couldn't know was whether the data mart model would reflect all the problems we might encounter on the data mart, or whether the server would introduce other issues. The question was; could we 'pass the buck' to the host, or would it be less willing than Harry S Truman in accepting the workload?

We'll explore the application by looking at

- the Production host environment
- the Client environment
- the host model built on Windows
- the foundation (metadata)
- the platform specific objects
- where it worked
- where it failed
- how metadata facilitated the repair

THE PRODUCTION HOST ENVIRONMENT

At the core of the information delivery was a Persistent Data Repository (PDR), populated by data produced daily from the corporate transaction systems. A number of different host systems were involved, with a number of new and existing unload processes. The PDR comprised more than 100 tables in a normalised star schema.

The central fact table recorded client information, and related tables stored transaction, historical and dimension information. Transactions included such information as payments, historical tables described time-based information like balances, and dimension information included code tables that described regions, account types and other similar classifications.

The corporate standard required that a mainframe DB2 repository was used, but larger Unix platforms with Sybase, Oracle or other RDDB systems might also have been appropriate.

The type of structure for the data is modelled in the segment shown in figure 1. The rounded red border tables describe the relational keys between tables. For example, a client may associate with more than one address, and an address may associate with more than one client. Since these associations may change over time, the relational tables also include dates describing the period of association.

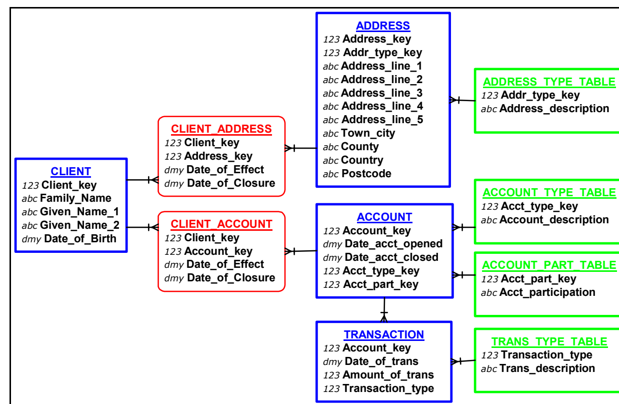

Figure 1

The green bordered tables on the right hand side all describe code table or derived values. Look at the following dummy of the address type code table. Consider whether the full text on the right is efficient to store, or whether substitution of the text with a numeric code might be an efficiency improvement. If the efficiency is unclear, then consider whether storing just the code value would make it easier in the future to create a new and very long address type description.

Address Type Table	
Code	**Description**
3	Primary residence
2	Normal Business
1	Postal address
0	Unknown

Figure 2

To deliver the required information, a data mart was assembled from the PDR to deal with a specific business need. Around 40 tables of de-normalised data were assembled in a structure that supported the record selection and reporting requirements of the application.

Around 1.2m records were stored in the central (fact) table of the data mart. Associated with the fact table were a series of dimension tables. A number of deeper tables held transactions and other records that clustered against individual client records, and a series of shallow tables held code table information.

The data mart was the first subset that was being created from the PDR. The use of a new PDR surfaced a large number of problems with the data extraction and load processes, and identified a significant number of data integrity issues. This impacted substantially on the delivery time for the project, but also meant the data mart was not available until quite late in the project.

THE CLIENT ENVIRONMENT

The client was built on Windows NT4, using SAS Version 6.12. (At the time, SAS Version 8 had not been accepted as a customer corporate standard.) The client services were made available in custom SAS/AF® frames, with SCL and submit blocks built for the application.

A common set of Windows macros were also deployed based on the Windows Application Programming Interface to provide common Windows functionality like file find, creation and deletion processes. Coupled with messaging based on the Windows Message Box API, this allowed core file management and messaging functions to be deployed early and made available with simple macro or method calls.

The primary function of the system involved selection of records from the central data mart table into groups. Other processes then allowed reporting and analysis of these groups for risk and qualitative assessment.

In the absence of the data mart, the selection function built code that was executed against model data sets stored on the LAN. With the availability of the data mart, the code could be moved to the mainframe and submitted on that host. At this time, it was intended to use SAS/Access® Views and descriptors to read the data mart.

THE HOST MODEL BUILT ON WINDOWS

The data extracted from the corporate systems was in a linear structure. This structure meant that multiple transactions were stored on a single record, and some client information was repeated across consecutive records of certain tables. This structure was normalised in the extraction and transformation processes for the PDR. This is the structure depicted in figure 1 above.

The data mart however, needed to support certain types of queries, so some denormalisation was performed in setting up the tables from the PDR. Compare the data mart structure depicted in figure 3 (below) with the PDR source depicted in figure 1.

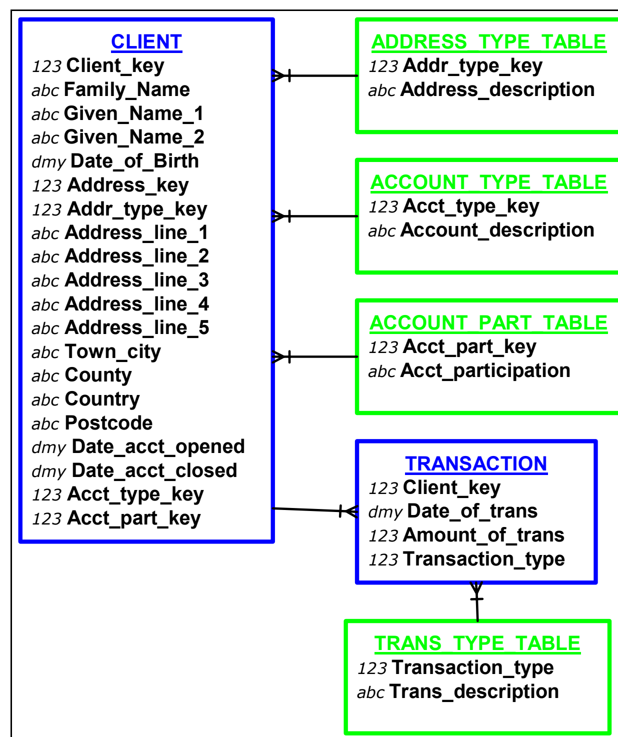

Figure 3

Note that we have now removed all the linking or foreign key tables. This is because this structure describes a current position. For instance, our concern is: "*John Smith lives at 123 Baker St Chester*". This allows for a smaller number of tables, which means fewer joins are needed to report information to the end user. The tables have, however, become wider and will take longer to process.

With copies of the source files available, SAS tables were built according to this final model, the model that would be seen in the data mart. These tables were then stored on the LAN environment and allowed development of the SAS/AF front end to proceed.

THE FOUNDATION (METADATA)

One of the first facilities made available from the systems analysis was a definition of the data sources and storage. This was transferred onto a spreadsheet and mapped the extracted data from end to end. The metadata could be represented in the manner shown in figure 4 below.

Here we see:

- PROCESS: the place in the process that takes the corporate data from the corporate transaction system to final utilisation.
- ENTITY: the element describing the data position at each process point. This is the column name on the spreadsheet.
- DESCRIPTION: a description of the data ownership, purpose or intention.
- FORMAT; the SAS format name that will translate from a SAS variable to any other entity on the diagram (this process is described later in the text).

Process	Entity	Description	Format
Extraction	Extract filename	Spec by analysts writing the extract processes	FExtFil
	Extract column name		FExtCol
Transform	PDR Table	Spec by the PDR DBAs	FPDRTbl
	PDR Table Column		FPDRCol
Loading	Data Mart Table	Spec by the DM DBAs	FDmTbl
	Data Mart Table Column		FDmCol
Access	SAS Table/View	Spec by the SAS developers	FSASTbl
	SAS Variable		
	SAS Variable type	Code table / Text / Date / Numeric / Logical	FSASTyp
Support	Description	Purpose / function of the extracted data element	FColDsc
	Reference	Master reference in the system data documentation	FColRef

Figure 4

The spreadsheet was then read into a SAS table and formed the metadata layer on which the SAS application was built. It is now possible to read the metadata and create SAS formats. These formats would generate table and column names for any preceding step of the data mart load process from the SAS variable name (in blue in the above table). A code example using the formats follows.

```
Data _NULL_;
  Set METADATA( Obs = 1);
  EXTFILE = Put( SASCOL, $FextFil.);
  EXTCOL  = Put( SASCOL, $FextCol.);
  Put SASCOL= EXTFILE= EXTCOL=;
Run;
SASCOL=FAMNAME EXTFILE=EX0001 EXTCOL=EX1023
```

When we validate the 'FAMNAME' column in our data mart, we can report any empty cells with a simple data step. We will also provide the table and column name for the original extract file, to allow the data to be validated against the source.

```
Data _NULL_;
  Set PERM.VMAIN( Keep = ACCOUNT FAMNAME);
  File PRINT;
  Where Left( FAMNAME) Eq: ' ';
  EXTFILE = Put( "FAMNAME", $FExtFil.);
  EXTCOL  = Put( "FAMNAME", $FExtCol.);
  If _N_ = 1  Then Put
    "Empty data in FAMNAME read from file "
    EXTFILE " column " EXTCOL;
  Put _N_ ACCOUNT;
Run;
```

This also simplifies all our data validation, since we can place the above code in a macro, and pass the value 'FAMNAME' as a macro parameter. As we pass a series of column names to the macro, reports are produced on each of the text columns that have empty values.

With this quite simple technique, we can pass data validation issues to the systems analysts, in a format they can readily use against their own documentation. In essence, we are passing the buck to those best placed to resolve a source data issue.

The power lies not just in end-to-end validation of our data mart however. If we wish, we may read the PDR directly, and validate the data at the point where it has been loaded from the extraction process and transformed prior to loading to the data mart. All we need is SAS/Access descriptors and views on that layer of the process. To do this, we would change three lines of the above code as follows:

```
Set PERM.VMAIN( Keep = ACCOUNT FAMNAME);
EXTFILE = Put( "FAMNAME", $FExtFil.);
EXTCOL  = Put( "FAMNAME", $FExtCol.);

Set PDR.VMAIN( Keep = ACCOUNT FAMNAME);
PDRTABLE = Put( "FAMNAME", $FPdrTbl.);
PDRCOL   = Put( "FAMNAME", $FPdrCol.);
```

Our metadata also allows us to use any column in an 'object style'. Consider the process of selection. Where a column contains codetable type data, we may wish to exclude rows based on the presence of certain code values. If we could identify those columns, it would allow us to process record selections with an object designed to build selection code for code table fields.

Our SAS format 'FSASTyp.' will identify the field type when it is applied to the variable name. If we were to store all our record selection rules in a data set, we could identify the type of selection process with this SAS format, and subset the table accordingly. Consider the following set of exclusions:

FIELD	VALUE	TYPE
CNTY_CD	SUSSEX	CODETABLE
CNTY_CD	ESSEX	CODETABLE
CNTY_CD	DORSET	CODETABLE
START_DT	01Jan1998	DATE
ADDR_CD	POSTAL ADDRESS	CODETABLE

Figure 5

Using this data, we could process it as follows:

```
Data _NULL_;
  Set WORK.EXCLUDE( Where = (
     Put(FIELD, FSASTp.) Eq: 'CODETABLE') );
  Put FIELD = VALUE=;
Run;
CNTY_CD SUSSEX
CNTY_CD ESSEX
CNTY_CD DORSET
ADDR_CD POSTAL ADDRESS
```

The following data step will build code that applies the exclusions we have specified.

```
Data _NULL_;
  Set WORK.EXCLUDE End = LAST;
  If _N_ = 1  Then Put
    'Data SELECT; Set PERM.VMAIN;';
  Put 'If ' FIELD ' Ne "' VALUE '";';
  If LAST  Then Put 'Run;';
Run;
```

We can use similar processes for numeric values, although these are likely to be simpler to store. This is because a code table selection could include tens or hundreds of exclusions (consider postcodes for instance), whereas a numeric process is likely to only have at most 2. These would specify a high and low bound, and in many cases we may only need one.

The benefit of using our metadata approach is that we can define the way a particular type of data will be handled, and we can build an object to perform that type of process. If we wish to increase the number of selection objects of a particular type, then the task is simplified because we simply register the selection object and make it available to our users. The underlying selection process will recognise the object and handle it appropriately… without code changes.

THE PLATFORM SPECIFIC OBJECTS

The objects we have described all function entirely within SAS code, and are unaffected by the operating system. This means that the code build process we have described will work on any platform, with any type of source data, as long as SAS treats it as a SAS table. This will include any RDBMS for which we have the appropriate SAS/Access product licensed, and against which we have built the necessary Access and View descriptors.

The code build processes we described above however, depend on reading data sets and writing out SAS code. We demonstrated the code being written to the log, but our application will require the code to be written to a file. The file could then be read back into our SAS session and executed with the command '`%Include SELECT.SAS;`'.

To ensure we don't execute code remaining from any previous session, we need to test for the existence of files, and delete them. We can delete a file in Windows by 'shelling' to the operating system and executing the command '`del`'. We can also execute the command with a '`call system`' function.

My preference however, is to use the Windows Application Programming Interface (API). This is not as simple as writing a piece of code to perform a task, because we need to lay some groundwork. The following elements need to be in place:

- A macro to perform the function, such as file delete, needs to be created and made available to the SAS session. This might be done by one of the following methods:
  1. include the code at the beginning of the SAS session, thus compiling the macro to the WORK.SASMACR macro catalog
  2. write the macro code to a file and assign that file to the SAS session by including the path to the file in the SASAUTOS path definition
  3. Compile the macro to a Stored Compiled macro library and assign this to the SAS session.
- The macro will build the elements of a Windows command, so a 'template' or prototype for the command is needed. This takes the form of a text file and needs to be referenced to the SAS session using the SASCBTBL reserved file reference.

As you can see, there is a little work we need to do to implement the Windows API. Here is a sample of some of the code needed. The first part is the macro to delete a file.

```
%Macro DeleFile( MfilNam =) /
   Des = "Delete the file named";

  Data _NULL_;
    FILENAME = "&MFilNam";
    RC = ModuleN( DeleteFileA",  FILENAME);
    If RC = 1 Then Put  RC
      "Deletion of &MFilNam completed";
    Else If RC = 0 Then Put RC
      "The file &MFilNam was not found.";
    Else Put @5
      "Unknown condition. Notify Support";
    Run;

  %MEnd DeleFile;
```

So that the Windows Operating system understands the command we send it with the ModuleN call, we need to create the following entry in our SASCBTBL file.

```
routine DeleteFileA
  minarg=1
  maxarg=1
  stackpop=called
  returns=ulong
  module=kernel32;
arg  1 char input format=$cstr200.;
```

So why go to all this trouble when a file deletion can be performed so easily with a Call System command? Because this method is more robust, has better error trapping and reporting, and because the first OS command we need is usually followed quite quickly by a second and a third. With more macros to come of increasing complexity, and without alternatives such as '`Call System`', the time expense now is warranted.

In the case of this application, many more were needed as messaging and other utilities were added. For a description of a number of Windows APIs, including those needed to manage files, the authors paper 'Coding across the boundaries' is available from the SUGI 26 proceedings.

For the mainframe application, a similar outcome and process was needed. The macros on OS/390 were very different in function, but by using a naming standard and set of parameters in common with the Windows macros, the code could be run equally effectively on both platforms.

```
%Macro DeleFile( MfilNam =) /
   Des = "Delete the file named";

  /* Test for the existence of the specified
file.  If the file does not exist, report
this to the log and branch to the MACREND
label.*/

  /* Read the files allocated to the SAS
session from the Dictionary.Extfiles SQL
View. If this file is already allocated to
the SAS session, issue a warning to the SAS
session and branch to the MACREND label to
avoid removing an open file reference. */

FileName ZZTEMPFL "&MFilNam"
   Disp = (Mod, Delete, Delete);

FileName ZZTEMPFL Clear;

%MACREND:

%MEnd DeleFile;
```

This particular macro demonstrates some of the robustness that is available to the application when a macro is used. The tests described in the macro would not be routinely performed each time a deletion was done within the code. Nonetheless, they provide some very important and robust processes for the system. A number of OS/390 macros were described in the users reprise of 'Coding across the boundaries' at the Views 2001 conference. The paper presented is available from http://www.dkvj.co.uk/presentations.html#VIEWS2001

Sample macros for both platforms are also available from the author's website at http://www.dkvj.co.uk/sasmacro/index.html.

The Windows macros were deployed by the following method.

- Allocate all necessary file references in the autoexec.
- Read and compile macros from a SOURCE entry in an AF catalog at the time the application was started.

Saving each macro as an individual member in a PDS (Partitioned Data Set) deployed the OS/390 macros. The PDS was then concatenated onto the SASAUTOS option path.

WHERE IT WORKED

The development of the Users application screens moved ahead well with data available in the model data sets. The code builder processes described above were refined and delivered before the testing data mart was ready, and the application consolidated effectively. Reporting processes, which also included object based report code builders, were developed and proven satisfactorily.

With the small testing data mart available, a handful of file references were changed, and the objects to establish, test and close connections to the remote host were added. The code was then running on the remote host, against the RDBMS, with acceptable results. As expected, the performance times also improved.

While more than half of the application was available, a lot of development was still occurring, and this was progressed against the LAN data. As each element of the application was readied, it was deployed to the System testing environment and run against the RDBMS and the remote host. Generally this was a trouble free process. The mainframe provided a faster processing environment for the queries performed.

A team of SAS programmers developed the application, with each working on a particular area. All used the common macros and methods and shared code pieces as development proceeded. The use of metadata at the core of all processing meant that the application was readily extensible, understood reasonably well by all the team and development shared effectively.

WHERE IT FAILED

The first problem became apparent quite quickly during integration testing of the selection processes. The following query, when run on a SAS table, would give us 700 matched records.

```
/* Test the distribution of the data */
Proc Freq Data = VMAIN.CLIENT;
  Tables ADDRTYPE / NoPrint Out = TEMP;
Run;

Data _NULL_;
  Set TEMP;
  Put ADDRTYPE= COUNT=;
Run;
ADDRTYPE=. COUNT=400
ADDRTYPE=1 COUNT=100
ADDRTYPE=2 COUNT=200
ADDRTYPE=3 COUNT=800

/* Select the desired rows from the data */
Data TEST;
  Set VMAIN.CLIENT( Keep = ADDRTYPE);
  Where ADDRTYPE Ne 3;
Run;
The Data Set TEST has 1 Variable and 700
Observations
```

However, when using the Views on the DB2 data mart, the TEST data set would have 300 observations. The reason for this was that DB2 handled missing values differently to SAS, and a missing value was excluded. A similar issue occurred with dates. In SAS, records selected below a given date include missing date values, but in DB2 the missing date values will be excluded. The reason is that DB2 does not store missing values as '-∞', but as a special value called a 'NULL'.

The solution involved remodelling the selection objects that dealt with numeric fields so that they were dealt with as text values. This was not as difficult as it seems, since all numeric selection code was built either by a numeric selection object, or a data selection object. To implement the new method, only two changes were required.

```
Data TEST;
  Set VMAIN.CLIENT( Keep = ADDRTYPE);
  Where Put( ADDRTYPE, 8.) Ne '       3';
Run;
The Data Set TEST has 1 Variable and 700
Observations
```

The second issue was one of performance. When the data set build processes were run against the full data mart, the system was too slow to be effective. The problem was traced to the load on the system, and the priority of the queries being run.

When tests were run against the testing data mart, the queries were being run against a small set of data. When we compare this against the production data mart, we see the following consequences:

- Reading a small data set requires a relatively short amount of time
- Writing the results from subsetting a small amount of data requires a relatively short amount of time
- That short amount of time might be provided in one or a few I/O requests, which may occur with little contention from other I/O requests.
- Writing a small amount of data can be done in a small space. This may often be a contiguous piece of disk space, which reduces the amount of disk movement, and consequently the time on a busy system.
- The first part of the task started on the mainframe system will run at a high priority, but the task priority will be lowered over time. If the task takes a long time, the reduction in task priority may disproportionately increase the amount of time for the task.

The mainframe system was busy, as was the DB2 address space administrator. This meant that other jobs on the mainframe DB2 system would slow down our longer running queries.

Analysis of the queries generated by the application showed that the design efficiencies implemented with the SAS table model were ineffective on the DB2 data mart. The key efficiencies were:

- Use of Keep and Drop options on Set statements to reduce the width of data retrieved
- Use of Where clauses to reduce the depth of data retrieved

- Indexing on the DB2 tables matched the where clauses to improve record retrieval times
- The multi-step selection processes were designed so that the first step was the most effective; i.e. the variables with the highest cardinality were used in the first process. (See Figure 3 below.)

CARDINALITY OF QUERIES

Looking back at our data model in figure 2, selection of records by address type and account type can be performed on the main table. With a handful of key values distributed fairly evenly across the data, exclusion of some of these would reduce the data significantly. Since these could be easily built into Where clause, these were our first selections.

If exclusions were then applied on County, the higher number of possible values meant that any substantial exclusion could result in a Where clause that could not be processed. Such exclusions were better performed by match merging, or use of formats.

The third type of selection might be on type of transaction: excluding accounts where certain transactions take place. This process would involve analysis of a different table, the Transaction table, and again may involve a match merge or use of formats.

Figure 6

The cause of the problem is that the SAS/Access engine performs a 'table unload' when a View is used against an Access descriptor. The consequences were:

- The whole data mart was being read, causing a high 'unload' demand on the DB2 administrator
- The whole data mart was being written to a temporary table causing high I/O activity, and high space demands.
- This 'unload' process also makes it clear that none of the DB2 indexes were being employed in query optimisation.
- The Where clauses, and the Keep and Drop options were only being performed on the temporary table that was being written from the View.

It may not be surprising to realise that such high demands for workspace were at times impossible for the system to meet. It became probable that the query would progress some considerable way to completion (taking a lot of time) before failing for resource problems.

The solution to the problem lay in rewriting some of the batch processes to use SQL Passthrough rather than the Access/View descriptors for the larger tables. With some very wide tables involved, and a system that was built using 'objects', tying the process down to hard coded SQL Passthrough would cripple some of the extensibility and flexibility of the system.

HOW METADATA FACILITATED THE REPAIR

The solution lay in the metadata, since the metadata knew the name of the DB2 table, the names of all the columns in that table, and the required names for the SAS variables that the metadata was formatting. It was possible to build an object that would take a set of SAS variable names and build a DB2 query that would retrieve only certain columns. It could also apply a where clause based on certain values, and match to an existing table using an inner join.

Here are some examples of the code to be built…

```
/* Select certain client records */
Proc Sql _Method STimer;

    Connect To RDBMS ( SSID = MYDB2);

    Create Table SELECT As
      Select *  From
      Connection  To RDBMS(
        Select CLIENT_KEY As CLIENT,
/* More column selection statements */
               ACCT_TYPE_KEY As ACTP_KEY
        From MYTABLES.CLIENT
        Where ADDR_TYPE_KEY Ne 3   And
              ADDR_TYPE_KEY Ne 4   And
              ACCT_TYPE_KEY Ne 15  And
              ACCT_TYPE_KEY Ne 23)
        Order By CLIENT;
    %Put PNB: RDBMS reports &SQLXRC &SQLXMSG;

    Disconnect From RDBMS;

Quit;
```

```
/* Join tables based on selections from a
second table. The second table can be used as
a CNTLIN data set for formats to exclude
certain clients. */
Proc Sql _Method STimer;

    Connect To RDBMS ( SSID = MYDB2);

    Create Table EXCLUDE As
      Select *  From
      ( Select CLIENT,
               ACCNT_NO
        From SELECT)
        Inner Join
      Connection To RDBMS(
        Select Distinct ACCOUNT_KEY As ACCNT,
          TRANSACTION_TYPE As TRANS_TP
        From MYTABLES.TRANSACTION)
        On ACCNT_NO = ACCNT
        Order By CLIENT;

    %Put PNB: RDBMS reports &SQLXRC &SQLXMSG;

    Disconnect From RDBMS;

  Quit;
```

Notice that what has been performed here is a rewriting of the query so that it is running within a block of code passed to the RDBMS. In this way, we can select our columns to increase the number of records in a block of data, we can select rows to reduce the total amount of data to be retrieved, and we can utilise indexes to match merge data together.

While the code here looks more complicated, it is no more difficult to build than the code we looked at in the code table selection process. Here are the key elements:

- We can write a query preamble to initialise the SQL procedure and connect to the RDBMS by testing for '`_N_ = 1`' in the data step.
- We can write the query postscript to test the RDBMS connection, report the SQL return code and disconnect from the RDBMS by using the '`END=`' option on our set statement.

- Our list of columns to select can come from a standard list held in a data set or if we are building in SCL submit blocks, in an SCL list. Our where statements are simply a translation of the selections specified in the selection data set.

SUMMARY

A large part of the system was built using SAS data tables to reflect a data mart that was not yet available. This introduced some problems, but without those tables, none of the code could have been adequately tested.

The system was built using a series of objects to perform functions.

- The object oriented parts of the application, i.e. the SAS/AF parts of the application worked without any real problems.
- The object based pieces that built code for execution against the data mart also worked effectively on test data, and when there were problems with the full data mart, changes were easy to make and implement.
- The object based pieces that supported Operating System functions worked without any problems at all.

Building the system in this manner also allowed a number of SAS people to work and develop on the system, and use common methods to deliver functionality.

The use of metadata to define functionality and build code was a powerful tool that made development, modification and implementation much easier.

There are currently a number of related organisations discussing the application with the client. If the client sells the system on to these other organisations, the modifications necessary to deliver the required functionality will be minimal. These modifications would almost certainly include table and column name changes to reflect their own data, but through updates to the metadata we could readily deal with most of these.

Using any approach that did not bring together metadata structure definition and object based development would substantially reduce the perceived value of the system.

We intended to 'pass the buck' from processing of LAN data sets to a mainframe environment and succeeded. We have also produced a system that can be easily supported and extended by support staff. They didn't seem to mind us 'passing the buck' for future work to them!

TRADEMARK CITATION

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

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

ACKNOWLEDGEMENTS

Paul Dorfman for providing the sounding board for ideas, and some insights into DB2 processing.

My family for their support, patience and late night coffee pots.

CONTACT INFORMATION

Your comments, suggestions and questions are valued and encouraged. Please contact the author:

David Johnson FRSA
DKV-J Consultancies
C/- 'Bonds Cottage'
Holmeswood Rd
Holmeswood nr Rufford
Lancashire England L40 1UA
Work Phone: +44 (0)7092 25 9556
Fax: +44 (0)7092 25 9556
Email: sugi274@dkvj.biz
Web: http://www.dkvj.biz