Paper 018-2007

JobTrack: Transaction Computing 101

Blake R. Sanders, U.S. Census Bureau, Washington, D.C.

ABSTRACT

Transaction computer application handle data in a completely different way compared to data analysis applications. Data analysis application display information and manipulate it many different ways. Transaction applications also create data and manage it over the course of time. This management has many facets. We decided to focus on two areas: data history and ease of maintenance of the datasets and application.

INTRODUCTION

The U.S. Census Bureau's Foreign Trade Division collects and reports information on international trade (i.e. imports and exports). Over the years, we've collected a huge amount of information and the public calls us with requests for that information all of the time. Until now, we've had a patchwork of databases to track these requests.

Our assignment was to design a web-based application (using Base SAS® and SAS/IntrNet® software) to combine and manage the data from many applications. Of all of the requirements of this task, one was the most challenging: data history (a.k.a. an audit trail). There is no reason to enter a record into the system, change it and have no idea who changed it (or know what the record was before the change).

A requirement we put on ourselves: ease of maintenance. When there is a problem (and there always is), we want to be able to focus on the solution rather than on finding the needle in the haystack.

EASE OF MAINTENANCE AND HISTORY

DATA STRUCTURE

JobTrack was designed to track data requests from customers. These requests can range from customized requests to regularly produced data products. The basic data will contain:

- Customers: name, contact information
- Requests: product, description, price
- Notes: description

ALL IN ONE

One way to structure this data is to put it all in one dataset:

Table 1 An example data structure.

Record	Name	Address	Phone	Product	Price	Note
1	Blake	123 Main St.	555-1212	FT900	100	
2	John	685 1st St.	555-8634	FT895	150	
3	Stefanie	8 Ave.	555-1234	IM145	125	Specific codes are on file.
4	Blake	123 Main St.	555-1212	IM145	125	Specific codes are on file.
5	Bruce	61 1/2 Virginia Ave.	555-2234	FT920	100	

This structure makes data maintenance tricky. If the contact information for the customer "Blake" needs to change, we need to ensure that it changes for record number 1 and in record number 4.

Table 2 The shaded may need to be updated.

Record	Name	Address	Phone	Product	Price	Note
1	**Blake**	**123 Main St.**	**555-1212**	**FT900**	**100**	
2	John	685 1st St.	555-8634	FT895	150	
3	Stefanie	8 Ave.	555-1234	IM145	125	Specific codes are on file.
4	**Blake**	**123 Main St.**	**555-1212**	**IM145**	**125**	**Specific codes are on file.**
5	Bruce	61 1/2 Virginia Ave.	555-2234	FT920	100	

We'd like to be able to add notes to requests. This way, as a request changes over time, we can add justifications and explanations. The all-in-one structure allows only one really large note per request.

SEPARATE PACKAGES

A better design relies on the characteristics of the objects we're trying to track. We're tracking a combination of customers, requests, products and notes. If each record has a unique identifier (like a serial number), we can have access to all of the data we had before with less maintenance and more flexibility. This allows us to maintain the objects without having to scour the larger database for every occurrence of the item we want to change.

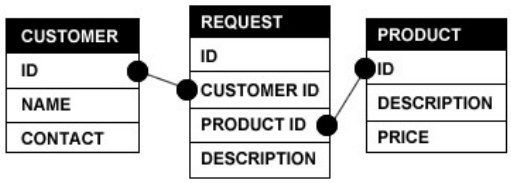


Figure 1 Better data structure.

Apply this concept to the "Notes" we had in the original data structure. Give each NOTE a serial number, but there still needs to be a reference from the NOTE back to the intended record.

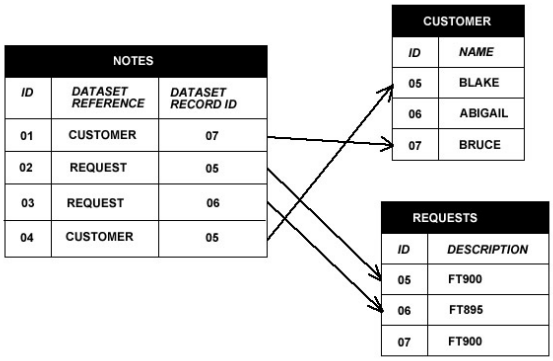


Figure 2 Linking NOTES to CUSTOMERS and REQUESTS

This design allows for multiple notes per customer and request.

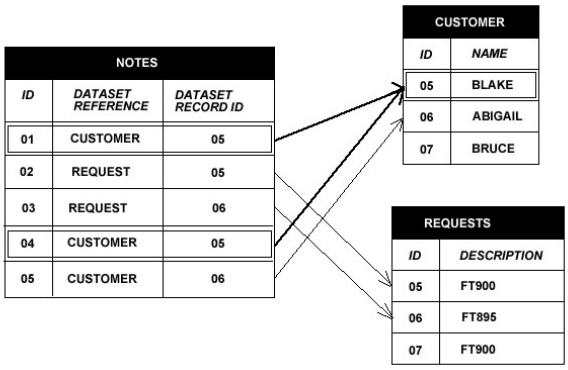


Figure 3 Multiple notes can point to the same request or customer.

HISTORY

Keeping a history of requests allows us to view them over time. If a customer has a history of frequently changing their requests, we can expect that the next time they place an order. If a customer disputes the details of a request, we can see when the request was changed and who changed it.

In the data structure, implementation is simple.

Table 3 Multiple versions per ID

ID	VERSION	NAME	PHONE
005	001	BLAKE	X1234
006	001	ABIGAIL	X3567
006	002	ABIGAIL	X4789
006	003	ABIGAIL	X2987
007	001	BRUCE	X6532

CODE: HISTORY

All of the records for customers, requests and products have ID numbers and version number. That way, we can track the changes over the course of time. Of course, when you want to only display the latest version (and not the whole history), you don't need every record with the correct ID number. DISPLAY pulls all of the ID records and then plucks out the most recent.

```
/* ---------------------------------------------------------- */
/* Get all of the records with the ID in question.            */

    data work.one;
           set tracking.&dbname;
           where id = "&rec_id";
    run;

/* ---------------------------------------------------------- */
/* Sort out all of the versions except the one that we want --  */
/* the most current version.                                  */

    proc sort data=work.one out=work.two;
           by id descending version;
    run;

    data work.three;
           set work.two (obs=1);
    run;
```

WHO DID WHAT?

The point of having a history is to be able to trace every action should the need arise.

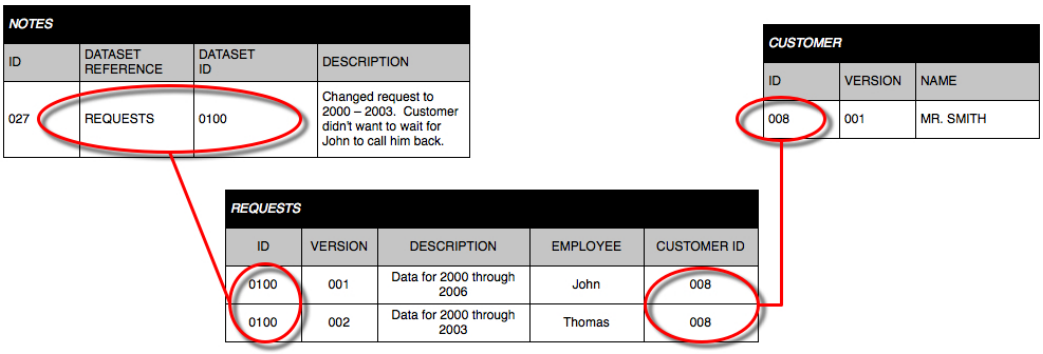
Example:

- John takes a request from Mr. Smith for data from 2000 through 2006.
- While John is at lunch, Mr. Smith calls and speaks with Thomas.
- Mr. Smith tells Thomas that he only wants data for 2000 through 2003.
- Mr. Smith tells Thomas that he doesn't want to wait for John to call him back.
- Thomas enters the change into the system and adds notes explaining the change.

Without the history, John wouldn't know who made the change and why.

JobTrack makes every user login. Once logged in, the username is saved to a session variable and is available to any and all programs running during that sessions.

Consequently, when data is added or changed, it is simple to take that name and stamp the new data with the name of the person that changed it.



NOTES

ID	DATASET REFERENCE	DATASET ID	DESCRIPTION
027	REQUESTS	0100	Changed request to 2000 – 2003. Customer didn't want to wait for John to call him back.

CUSTOMER

ID	VERSION	NAME
008	001	MR. SMITH

REQUESTS

ID	VERSION	DESCRIPTION	EMPLOYEE	CUSTOMER ID
0100	001	Data for 2000 through 2006	John	008
0100	002	Data for 2000 through 2003	Thomas	008

Figure 4 JobTrack lists who made the changes and allows notes.

CODE: WHO DID WHAT?

In order to Track which user records data, we need to know who is using the application. The best way to do that is to make the users log into the system

LOGIN

LOGIN handles three things: user logins, password resets and sending people to the main menu of the application. It assumes that, if you are logging in the first time, there will be a username and password. If you are resetting your password, it assumes that there will be a username and an indicator that the password needs to be reset. Finally, if you've already logged in, it assumes there will be a variable indicating there is already an authenticated user.

LOGGING IN

```
%macro one;
      %let resetPWIndicator = %superq(resetPassword);

/* If there is a password, assume they just want to login. */

      %if (&resetPWIndicator ne ) %then
      %do;
            %login(&username, &password, &resetpassword);
      %end;
            %else
```

```
     %do;

/* If there is no password, see if they are an authenticated user. */

            %if %sysfunc(exist(save.authenitcatedUser)) %then
            %do;
                  %frontPage;
            %end;
                  %else

/* If not an authenticated user, assume they want to reset the password. */

            %do;
                  %login(&username, &password, 0);
            %end;
     %end;
%mend one;
%one;
```

TRACK WHO LOGGED IN SUCCESSFULLY

If the user logs in successfully, the system will set a few session variables based on the login file: the username, the email of the user and the user's authorization level.

```
   %global SAVE_AUTHENTICUSER SAVE_AUTHENTICUSER_EMAIL
SAVE_AUTHENTICUSER_LEVEL;
   %let SAVE_AUTHENTICUSER = "&user";
   %let SAVE_AUTHENTICUSER_EMAIL = "&useremail";
   %let SAVE_AUTHENTICUSER_LEVEL = "&userlevel";
```

This way, the system knows the name of the user performing any action throughout the system. Also, it knows which options to show the user based on their authorization level.

OPERATIONAL FLOW OF THE APPLICATION

JobTrack has two main actions:

- Add
- Change

ADD

ADD requires that we display a specific form, allow users to enter the data and accept those values. At first blush, it seems as though we need to accommodate every possibility with its own file.

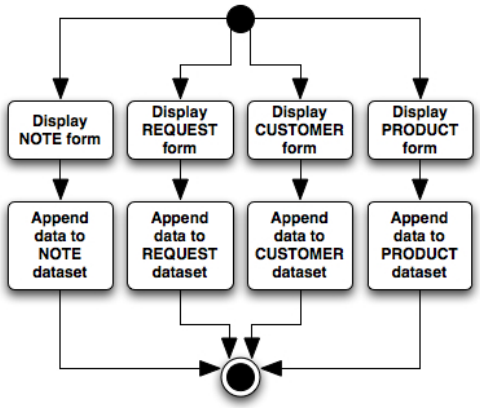


Figure 5 Redundant ADD functions

However, this produces a great deal of redundancy, as the logic to display forms for NOTES, REQUESTS, CUSTOMERS and PRODUCTS is nearly identical. The same is true for appending a new record to the database. Because of this, we can consolidate many of the files.

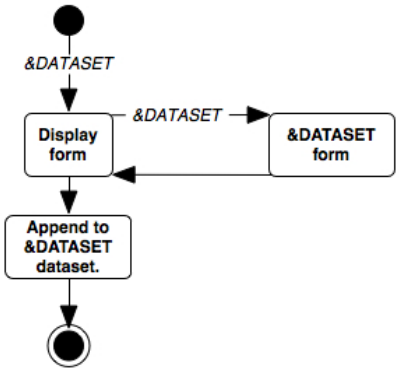


Figure 6 Non-redundant ADD functions

The program that displays the form includes a second file appropriate to the situation. The SAS code necessary to do this is listed under CODE: DISPLAY section later in this paper.

CHANGE

CHANGE requires that we find an existing record, display it in a form and accept any and all changes. Like ADD, we can write individual programs to handle all situations.

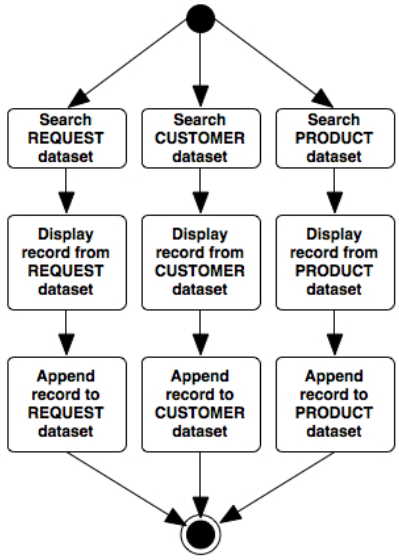


Figure 7 Redundant CHANGE functions

Again, like ADD, we can consolidate the redundant code in all of the options.

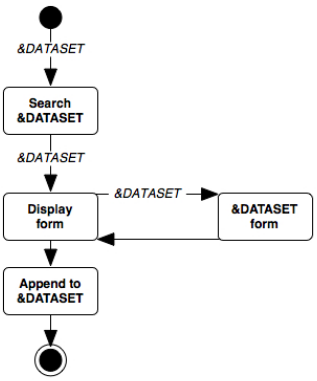


Figure 8 Non-redundant CHANGE functions

Given the similarities between the ADD and CHANGE structures, we can combine them into one.

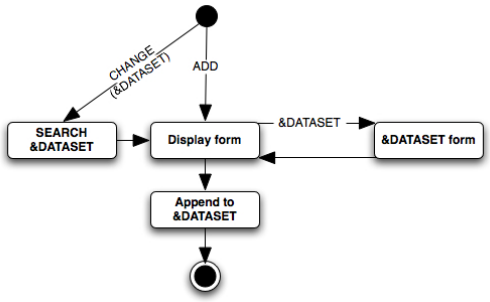


Figure 9 Combined ADD and CHANGE functions

CODE: DISPLAY

We save a great deal of administration by having one program in charge of displaying forms to the screen. This is done by assuming that the only things that will be different from form to form are the fields displayed. So, we can have the one program display the basic structure of the HTML page and include the details of the correct forms when the circumstances warrant.

```
/* ---------------------------------------------------------- */
/* Display the record in question.                            */

   %if &dbname = CUSTOMERS %then %do;
       %include ftrack(customers.html);
   %end;

   %if &dbname = REQUESTS %then %do;
      %include ftrack(requests.html);
   %end;

   %if &dbname = NOTES %then %do;
      %include ftrack(notes.html);
   %end;

   %if &dbname = PRODUCTS %then %do;
      %include ftrack(products.html);
   %end;
```

Based on the record that was pulled using DISPLAY, the included program (in this case CUSTOMER) displays the

correct HTML form.

```
options symbolgen mprint;
%include ftrack(html_pieces);
%macro one;
    %pageHead(Display Customer);
    data _null_;

           %if (&workingRecords ne ) %then
           %do;

           [ Print fields if there is an existing record. ]

           %end;
                 %else
           %do;

           [ Print fields if this will be a new  record. ]

           %end;

    %pageFoot;

%mend one;

%one;
```

ACTION

In all of our previous examples, we've appended data to the database in question. That's one ACTION of many.

JobTrack displays its content at the top of the page and an ACTION menu at the bottom of the screen.

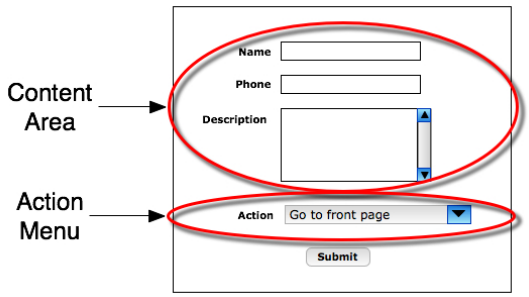


Figure 10 Display screens have a content area and an action menu.

This menu sets the "_program" variable used in SAS/IntrNet to initiate the next program. The "action" can run the gambit from "Add/Change Data" to "Display Requests From This Customer" to "Go To The Front Page".

The program "Add/Change Data" is a good example of how to use the data in the form to take action.

For all of the datasets, there are two types of records: new records (with no ID or VERSION) and existing records (with a specific ID and VERSION). For new records, DISPLAY will show empty text fields.

```
<!—- New record – XHTML -->

<html>
<body>
<form method="post" action="/scripts/broker.exe">
<p>
```

```
ID: New Record (Not Available)
<input type="hidden" name="id" value=""/>
</p>
<p>
Version: New Record (Not Available)
<input type="hidden" name="version" value=""/>
</p>
<p>
<label for="textfield">Name</label>:
<input type="text" name="name" value=”"/>
</p>
…
…
…
</form>
</body>
</html>
```

For existing records, display will show the values of the latest version.

```
<!—- Existing record – XHTML -->

<html>
<body>
<form method="post" action="/scripts/broker.exe">
<p>
ID: 0005
<input type="hidden" name="id" value="0005"/>
</p>
<p>
Version: 0003
<input type="hidden" name="version" value="0003"/>
</p>
<p>
<label for="textfield">Name</label>:
<input type="text" name="name" value=”Blake”/>
</p>
…
…
…
</form>
</body>
</html>
```

If the user selects ADD/CHANGE DATA from the ACTION menu, both of these forms will be submitted to the same program. How does that program, COPY_ID know what to do? It just looks to see what the form says.

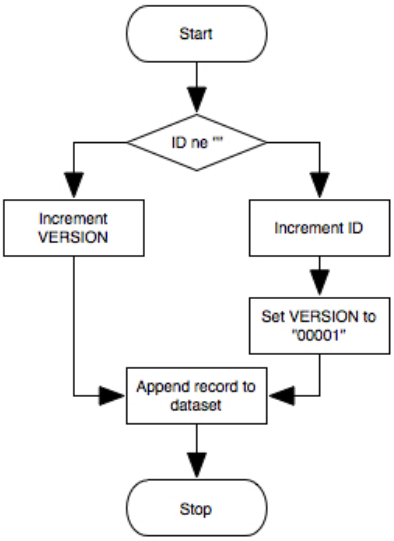


Figure 11 Flowchart for new vs. existing records.

If ID does not exist, it is assumed that the data is for a new customer, request or note. In that case, we need to find the highest existing ID and increment it, set the version to “00001” and append it to the dataset.

If ID does exist, increment the version and append it to the database.

CODE

The following code does not facilitate the history or ease of maintenance of the application, however, they are integral to the operation.

RESET THE PASSWORD

To reset the password, a new password is randomly generated and e-mailed to the user.

```
%macro pass(x);
    %let txtstg = 0123456789abcdefghijklmnopqrstuvwxyz;
    %let passwd = ;
%let char = ;
    %do i = 1 %to &x;

/* Assume that each new character of the password will not be  */
/* unique.                                                     */

          %let uniqueChar = 0;

/* Keep pulling new characters until you get one that is not   */
/* already in the system.                                      */

          %do %until (&uniqueChar>0);

/* Pull a character from the list at random.                */

                data _null_;
                   txtstg = "&txtstg";
                   c_index = round(uniform(0)*length(txtstg)) + 1;
                   char = substr(txtstg,c_index,1);
                   if char = ' ' then char = '_';
```

```
                    call symput('c_index', c_index);
                         call symput('char', char);
                  run;

/* If the character is not already in the password, it is a    */
/* unique character.                                           */

                    %if %index(&passwd,&char) = 0 %then
                    %do;
                       %let uniqueChar = 1;
                    %end;

            %end;

/* If the character is unique, add it to the password.         */

            %let passwd = %trim(&passwd)&char;

   %end;

%mend;
```

HTML_PIECES

DISPLAY outputs the main structure of the form. This is done through many included macros that handle the details of the HTML. Using this method, DISPLAY focuses on the message. The HTML_PIECES focus on how to get that message across. When methods change (like changing from HTML to XHTML), the HTML pieces are where most of the changes will take place.

```
%macro pageHead(title);
    data _null_;
           file _webout;
           put "<html>";
           put "  <head>";
           put "  <title>&title</title>";
           put "  </head>";
           put "  <body>";
    run;
%mend pageHead;

%macro pageFoot;
    data _null_;
           file _webout;
           put "  </body>";
           put "</html>";
    run;
%mend pageFoot;

%macro displayMessage(theMessage);
    data _null_;
           file _webout;
           put "&theMessage";
    run;
%mend displayMessage;
```

CONCLUSION

By designing the system to focus on tracking requests and their associated version while keeping the structure simple, we were able to learn some specific lessons.

- Having a similar structure for all of our datasets allows us to reuse a great deal of code.
  - DISPLAY compresses the functions of six processes into one process.

- A transaction application can easily handle a history, but it's not always necessary to see it. Adding just enough administration to reveal the latest version makes it palatable.
  - Pull the required ID and do a descending sort based on VERSION. Display only the top record (the latest version).

- With judicious use of session variables, we're able to keep track of who makes changes in the system and thus make the history (a.k.a. audit trail) more effective.
  - All transaction that add or change data are stamped with the value of those variables.

We've been able to build a system to take advantage of our data structure to produce a system that is lean, easy to maintain and produces an effective history of our data requests.

REFERENCES

SAS OnlineDoc 9.1.3: http://support.sas.com/onlinedoc/913/docMainpage.jsp

ACKNOWLEDGMENTS

The following people helped in the development of JobTrack: Lori Dickerson, Monique Eleby, Theresa Gordon and Bruce Walker.

CONTACT INFORMATION

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

Blake R. Sanders
U.S. Census Bureau / Foreign Trade Division
4600 Silver Hill Rd.
Rm 6K505
Washington, DC 20233

Work Phone: 301-763-2234
E-mail: blake.r.sanders@census.gov
Web: http://www.census.gov/trade