Centralization and Regionalization at National Agricultural Statistics Service

Roger Schou, National Agricultural Statistics Service, USA

1. Introduction

The tides of change never seem to ebb in much of life. Changes at the National Agricultural Statistics Service (NASS) are no exception. NASS is the agency of the United States Department of Agriculture responsible for the nation's agriculture data. We use the typical modes of data collection including CATI, paper questionnaires for mail and field interviews, CAPI field interviews, and web self-administered instruments. Blaise is used at NASS for CATI data collection as well as one of two main editing systems for data collected in all modes. Our CASIC system utilizes Blaise, Manipula, ManiPlus, and Visual Basic .NET (VB.NET).

The office structure of NASS has undergone a very large change over the last year: regionalization. We are nearly complete in moving from forty-six field offices across the United States to twelve regional offices. The states that did not become a regional center will still have a presence person and a field interviewer coordinator, but the majority of the survey work will be completed in the regional field offices (RFOs). We still have our headquarters (HQ) in Washington, D.C., a research division in Virginia, and our National Operations Center (NOC) in St. Louis, MO. The NOC serves as a primary calling center for NASS with a capacity of up to 166 phone interviewers. As a result of the NOC becoming fully functional, we have closed two of our field office Data Collection Centers (DCCs). The majority of the CATI data collection is done at the NOC and our four remaining field office DCCs. The majority of the paper forms are processed at the NOC.

In 2010 at the International Blaise Users Conference (IBUC) XIII in Baltimore, MD, we reported on the successes and lessons learned from converting our first survey to a centralized environment. In 2012 at IBUC XIV in London, England, we had centralized another twenty-five surveys. We now have approximately fifty-five surveys in centralized Blaise.

2. Centralized Blaise Database and Tables at NASS

We continue to store our transactional data in a MySQL database. We are writing directly to the database from Blaise using BOI files. These files contain the connection information to the MySQL tables. We are using generic in-depth storage so all of our surveys are stored in the same format in the database. This allows us to have one Extract, Transfer, Load (ETL) program running to copy the necessary data from the transactional MySQL database to the analytical Redbrick database.

The generic in-depth storage in Blaise yields eight generic tables: BLAISE_DICTIONARY, BLAISE_ID, BLAISE_CASE, BLAISE_FORM, BLAISE_KEY, BLAISE_DATA, BLAISE_REMARK, and BLAISE_OPEN.

The BLAISE_DICTIONARY table keeps track of all of the surveys in the MySQL database. Anytime there is an instrument change (a change in the checksum), a new entry is created in this table. The change could be as minimal as data model name. This is how new instances are created of our surveys including our monthly and weekly surveys. We chose to use the survey folder name as the data model name, as it contains at a minimum a year, a month, if needed, and a day, if needed. This makes the data model names unique.

The paths to the externals also need to be update for a given survey. For a weekly or even monthly survey, manually changing the data model name and the paths of the externals for each survey period is too labor intensive.

Code to search and replace the name of the data model was put into place as the previous and current data model names are equivalent to the survey folder names and thus known.

The other issue with the weekly and monthly surveys is that the path to the externals changes with each survey period. To automate this, we put all of the external sections in their own INCLUDE files. The VB.NET code creates the INCLUDE files when a user clicks the button. These files are then included within the appropriate blocks of the instrument, so they are no longer hard-coded. Once the data model has been renamed and the external INCLUDE files have been created, the menu prepares the new instrument, and it is ready for the next survey period. Currently this is all in one button click of the menu. Once we are sure the process is bug-free, we are going to make it a CRON job that will run automatically.

Table 1. BLAISE_DICTIONARY

DMKEY: integer	Data model key (unique 1-up)
DATAMODELNAME: varchar(255)	Name of the data model
CHECKSUM: varchar(29)	Checksum of the data model
DPT: integer	Data partition type
BMI: varchar(255)	Path to the associated BMI file
BOI: varchar(255)	Path to the associated BOI file
SEARCHPATH: varchar(255)	Dictionary search path
ADDED: datetime	Date and time of adding
ADMINKEY: varchar(255)	Encrypted administrator password
COLLECTMODES: varchar(255)	*Not yet implemented*
CAB: blob(16777215)	Cabinet file

The BLAISE_ID table contains all of the block names and the field names within a given instrument. Blocks and fields are numbered independently: they each start at 1. We have replaced our Cameleon scripts with VB.NET code that makes use of the BLAISE_ID table. We create a mapping file for code data that is keyed from paper questionnaire linking the item code to a field in the Blaise dataset. For this, we use the field tag. We use the field description to store our variable names which are used by the ETL.

If it is simply a single field that needs the item code and variable name, assigning them is straightforward. However, if there is a block that is used more than once as a type and the fields within each instance of the block have different item codes and variable names, then the simplistic approach does not work. We have implemented what we call hash notation. If the field tag and field description are defined on the field using the block as its type, we use the hash mark (#) to separate the field tags and field descriptions. There is a one-to-one relationship between the field tags and field descriptions at this level and the fields within the block being used as the type. If the first field in the block does not have an item code or a variable name, we begin the field tag with a zero and the field description with a #. The VB.NET code then reads the BLAISE_ID table and for blocks that have field tags or field descriptions, the hash notation is deciphered and the appropriate field tag and field description are assigned to the corresponding fields within that block. The hash notation is then deleted from the block level.

In Example 1 the planted, harvested and production questions have item codes, and the amount and unit questions do not. The block is reused for corn and soybeans with the unique item codes and variable names defined outside of the actual block.

Example 1. Code Utilizing Hash Notation

```
BLOCK bCropBlk
    PARAMETERS
        piCropName : STRING
    FIELDS
        Planted “How many acres of ^piCrop were planted?” : INTEGER[9]
        Harvested “How many acres of ^piCrop were harvested?” : INTEGER[9]
        Amount “What was the total amount of ^piCrop produced?” : INTEGER[9]
        Unit “What was the unit produced?” : (bushels, tons, pounds, hundred "hundred weight")
        Produced : INTEGER[9]
    RULES
        Planted
        Harvested
        Amount
        Unit
        Produced.KEEP
        pCalcPrdctnProc(Amount, Unit, Produced)
ENDBLOCK

FIELDS
    Corn (530#531###370) “” / “CCRNXXPL#CCRNXXHV###CCRNXXPD” : bCropBlk
    Soybeans (600#763###227) “” / “CSOYXXPL#CSOYXXHV###CSOYXXPD” : bCropBlk
RULES
    …
```

After running the VB.NET code to reassign the field tags and field descriptions, the Corn.Planted field would have 530 as the field tag and CCRNXXPL as the field description. Soybeans.Harvested would have 763 as the field tag and CSOYXXHV as the field description. The Corn.Unit would not have a field tag or a field description. The one thing that is forfeited by using this structure is the ability to jump to a field tag within the instrument.

For fields that are defined by arrayed blocks with repeated item codes and similar variable names, the field tags and field descriptions are actually attached to the fields within the block being used as a type. When an array is encountered with field tags or field descriptions, then the mapping file puts a keyword into the array element number and adds an extra parameter which is a string with the word “ARRAY.” The Manipula that reads in the data detects the word and then substitutes the keyed table and row number on the raw data file into the array element keyword.

The variable names must also be unique for the ETL. So VB.NET code sees the array in the BLAISE_ID table and the field description within the block. It will then take the array element number from the Blaise field name using the first digits as a table number and the last two digits as the row number. The new resulting variable name becomes VarName_TableNum_RowNum. The arrays in the Blaise instrument must be assigned with this variable naming schema in mind.

Once the field tags and field descriptions have been updated in the BLAISE_ID table, our VB.NET code, that replaced our old Cameleon code, creates the files needed for reading code data into Blaise.

Table 2. BLAISE_ID

DMKEY: integer	Data model key (unique 1-up)
ID: integer	Id of B/F (unique 1-up for each)
TYPE: varchar(1)	Indicates type of ID (B=block, F=field)
NAME: varchar(255)	Fully qualified name of the B/F
TABLENAME: varchar(255)	MySQL table name containing B/F
DATATYPE: varchar(255)	B/F type as defined in the instrument
DECIMALS: integer	Number of decimals defined for the B/F
FIELDSIZE: integer	Length of the B/F
EMPTY: varchar(3)	YES/NO: Is EMPTY allowed for B/F?
DONTKNOW: varchar(3)	YES/NO: Is DK allowed for B/F?
REFUSAL: varchar(3)	YES/NO: Is RF allowed for B/F?
ARRAYINDEX: integer	Array index for B/F (or -1)
MININDEX: integer	Minimum value for array index
MAXINDEX: integer	Maximum value for array index
MINVALUE: integer	Minimum value for B/F (0 for string)
MAXVALUE: integer	Maximum value for B/F (0 for string)
ISSET: varchar(3)	YES/NO: Is B/F and enumerated set?
ISARRAY: varchar(3)	YES/NO: Is B/F an array?
ISTABLE: varchar(3)	YES/NO: Is B/F a table?
ISEMBEDDEDBLOCK: varchar(3)	YES/NO: Is B/F an embedded block?
FIELDTAG: varchar(255)	Field tag for B/F (null if not defined)
QUESTIONTEXT: varchar(255)	Question text for B/F
DESCRIPTIONTEXT: varchar(255)	Description text for B/F

The BLAISE_CASE table contains the JOINKEY values for each record in the survey.

Table 3. BLAISE_CASE

JOINKEY: integer	Integer identifying a record (unique 1-up)
DMKEY: integer	Data model key (unique 1-up)
KEYVALUE: varchar(255)	Internal culture independent value of the primary key

The BLAISE_KEY table contains all of the primary and secondary keys as defined in an instrument for a survey. The BEGINSTAMP value has to be unique in combination with JOINKEY and DMKEY.

Table 4. BLAISE_KEY

JOINKEY: integer	Integer identifying a record (unique 1-up)
DMKEY: integer	Data model key (unique 1-up)
BEGINSTAMP: datetime	Begin time of the period of validity of a particular record. This column is used for versioning.
KEYNAME: varchar(255)	Name of the key as defined in instrument
KEYVALUE: varchar(255)	Internal culture independent value of the primary key

The BLAISE_FORM table contains the status information about each record. The table also contains information about collection mode and the data entry behavior being used when the record was stored.

Table 5. BLAISE_FORM

JOINKEY: integer	Integer identifying a record (unique 1-up)
DMKEY: integer	Data model key (unique 1-up)
BEGINSTAMP: datetime	Begin time of the period of validity of a particular record. This column is used for versioning.
ENDSTAMP: datetime	End time of the period of validity of a particular record. The ENDSTAMP of newly inserted record will have the predefined value of '99991231

	00:00:00'.
STATUS: tinyint	Form status of the record (1=Clean, 2=Suspect, 4=Dirty, 8=NotChecked)
COLLECTIONMODE: integer	*Not yet implemented*
DATAENTRYBEHAVIOUR: integer	Data entry behavior during the time the record was written (0=bldbUncheckedEditing, 1=bldbCheckedEditing, 2=bldbFreeInterviewing, 3=bldbStrictInterviewing).
ERRORCOUNT: integer	Number of errors in current record
REMARKCOUNT: integer	Number of remarks in current record
DONTKNOWCOUNT: integer	Number of DK answers in current record
REFUSALCOUNT: integer	Number of RF answers in current record
STREAMSTATUS: varchar(1)	Status of the stream that has been stored in STREAMDATA. (Uppercase O=out of sync).
STREAMDATA: blob(16777215)	Binary stream of current record

The BLAISE_DATA table contains the data for the records in the survey.

Table 6. BLAISE_DATA

JOINKEY: integer	Integer identifying a record (unique 1-up)
DMKEY: integer	Data model key (unique 1-up)
BEGINSTAMP: datetime	Begin time of the period of validity of a particular record. This column is used for versioning.
FIELDID: integer	ID of the field to which the data belongs
STATUS: tinyint	Status of the field (1=Unprocessed, 2=Response, 4=DK, 8=RF)
STRINGDATA: varchar(255)	Answers to fields with string data type
INTEGERDATA: integer	Answers to fields with integer or enumerated type
FLOATDATA: double	Answers to fields with real data type
DATETIMEDATA: datetime	Answers to fields with date and time data type

The BLAISE_REMARK table contains the remarks left on fields for a survey.

Table 7. BLAISE_REMARK

JOINKEY: integer	Integer identifying a record (unique 1-up)
DMKEY: integer	Data model key (unique 1-up)
BEGINSTAMP: datetime	Begin time of the period of validity of a particular record. This column is used for versioning.
FIELDID: integer	ID of the field to which the remark belongs
REMARKTEXT: text(16777215)	Text of the remark

The BLAISE_OPEN table contains the answers to any OPEN fields with a response.

Table 8. BLAISE_OPEN

JOINKEY: integer	Integer identifying a record (unique 1-up)
DMKEY: integer	Data model key (unique 1-up)
BEGINSTAMP: datetime	Begin time of the period of validity of a particular record. This column is used for versioning.
FIELDID: integer	ID of the field to which the remark belongs
STATUS: tinyint	Field status of the open field (1=Unprocessed, 2=Processed, 4=DK, 8=RF)
OPENTEXT: text(16777215)	Answer text of the open field

In addition to these standard eight generic tables that Blaise creates, NASS has added a few flat data tables to help manage the processes as well as increase performance.

The CASIC_SURVEYINFO table contains survey-level information such as the instrument name, folder name, BOI file name, a number of indicators, and some starting and ending dates. Our menu system makes extensive use of this table.

Table 9. CASIC_SURVEYINFO

SAMPLE_ID: integer	Integer identifying a survey
YEAR_: integer	Four-digit year of the survey
MONTH_: integer	Month of the survey
DAY_: integer	Day or Week of the survey
INST_NAME: varchar(15)	Name of the .bmi file
FOLDER_NAME: varchar(15)	Folder name for the survey
BOI_NAME: varchar(15)	Name of the .boi file
PRODUCTION_IND: integer	Indicates if a survey is in production (0=beta, 1=production, 9=in production but deactivated)
SHELL_IND: varchar(4)	Indicates which NASS shell code was used (L7=list, MF7=multiple frame, C7=Census)
NONOPDOM_IND: integer	Loosens some of the restrictions on records
CAMELEON_IND: varchar(1)	Indicates which Cameleon scripts are to be used (I=code data input, O=code data output, B=both code data input and output, N=neither)
REGIONAL_IND: integer	Indicates if survey is regionally processed (0=not regional, 1=regional survey, 2=regionally-processed survey)
IDAS_FOLDER: varchar(15)	Folder name for data fed to analysis
DM_KEY: integer	Corresponding DMKEY in the BLAISE tables. Used by ETL to know which surveys to transfer to analytical database
SMETA_KEY: varchar(8)	Number identifying a survey in Questionnaire Repository System
ALERT_IND: integer	Indicates that alerts will be used to check in forms keyed at the National Processing Center (NPC)
NPC_IND: integer	Indicates NPC will be involved in survey
SURVEY_TYPE: varchar(2)	Two-character survey type for Census surveys
SURVEY_ABBREV: varchar(3)	Three-character survey abbreviation for Census surveys
SURVEY_DESC: varchar(50)	Survey name from Metadata Repository System
BETA_START_DATE: datetime	Date survey becomes available on Beta
COL_START_DATE: datetime	Data collection start date on Production
COL_END_DATE: datetime	Data collection end date on Production
EDIT_END_DATE: datetime	Editing end date on Production
SDATE: date	Date associated with survey
SURVEY_ID: tinyint	ELMO survey id
PERIOD_ID: tinyint	ELMO classify period
NEWADDCATEGORY: tinyint	Newly added record category to indicate if survey allows newly added records
SAMPLE_TYPE: tinyint	Sample type (e.g. aggregated, independent, area, census, etc.)
FREQUENCY: tinyint	Survey frequency (e.g. annual, quarterly, monthly, weekly, etc.)
PARTNER_INCL: tinyint	Indicator to include or exclude partners

The CASIC_FAT table is used to control access to certain groups of records to an authorized state, data collection center, estimation center, or regional field office.

Table 10. CASIC_FAT

SAMPLE_ID: integer	Integer identifying a survey
YEAR_: integer	Four-digit year of the survey
MONTH_: integer	Month of the survey
DAY_: integer	Day or Week of the survey
FO_FIPS: integer	State in the survey
SURVEY_DESC: varchar(50)	Survey name from Metadata Repository System
DCC_FIPS: integer	State responsible for collecting the FO_FIPS state data
EC_FIPS: integer	State responsible for editing the FO_FIPS state data
ADD_FUNC: integer	Used for multi-state DCC or EC functionality
BETA_INIT_PREP: integer	Indicates the input sample files for this state have passed the preparation phase on Beta
BETA_INITIALIZED: integer	Indicates this state has been initialized on Beta
INIT_PREP: integer	Indicates the input sample files for this state have passed the preparation phase on Prod
INITIALIZED: integer	Indicates this state has been initialized on Prod
NAME_STAMP: varchar(10)	Login name of the person who updated this line in the table
DATE_STAMP: datetime	Date of the update of this line in the table
RFO: integer	Region code for the FO_FIPS
DCC_RFO: integer	Region responsible for collecting the FO_FIPS state data
EC_RFO: integer	Region responsible for editing the FO_FIPS state data

The CASIC_MANAGEMENT table has several key fields that also appear in all of our instruments. This table has been indexed on these fields, which increases the performance of the Blaise instruments. It allows the record filters in Manipula and the .BOI files to be much more efficient because the database being accessed is indexed.

Table 11. CASIC_MANAGEMENT

DMKEY: integer	Data model key (unique 1-up)
LFINFO_STATE: integer	State field (part of primary key)
LFINFO_ID: integer	ID field (part of primary key)
LFINFO_TRACT: integer	Tract field (part of primary key)
LFINFO_SUBTRACT: integer	Subtract field (part of primary key)
BEGINSTAMP: datetime	Begin time of the period of validity of a particular record. This column is used for versioning.
MANAGEMENT_DCC_FIPS: integer	State responsible for data collection
MANAGEMENT_EC_FIPS: integer	State responsible for editing
MANAGEMENT_FO_FIPS: integer	State to whom the record belongs
MANAGEMENT_BATCH: integer	Batch number to which the record belongs
MANAGEMENT_PROCESSSWITCH: integer	NASS process indicator (1=CATI Complete, 2=Default, 3=Edited, 4=Sent)
LFINFO_PID: integer	Another ID-type field
LFINFO_COUNTY: integer	County field
MANAGEMENT_RFO: integer	Region to whom the record belongs
MANAGEMENT_DCC_RFO: integer	Region responsible for data collection
MANAGEMENT_EC_RFO: integer	Region responsible for editing
LFINFO_XSTATELINK: integer	Field used to link records

The CASIC_EVENT_LOG table tracks the activity on the CASIC Menu. It has an entry consisting of the person's login name, location, the date and time a button was clicked, and the survey involved. There is another entry logging the end of the process. This was created as a debug tool to indicate what procedures were running when we saw any undesirable issues arise.

Table 12. CASIC_EVENT_LOG

DMKEY: integer	Data model key (unique 1-up)
SURVEY: varchar(50)	Survey name from Metadata Repository System
YEAR_: smallint	Four-digit year of the survey
MONTH_: tinyint	Month of the survey
DAY_: tinyint	Day of the survey
WEEK_: tinyint	Week number of the survey
STATE: tinyint	State where the button was clicked
LOGIN_NAME: varchar(10)	Login name of person who clicked the button
PROCESS_NAME: varchar(50)	Name of the process running
EVENT: varchar(10)	The event of the process: BEGIN/END
TIMESTAMP: timestamp	Date process began or ended

3. Centralized Blaise Concept

The concept of Centralized Blaise is a simple one. There is one dataset, centrally located, with controlled access for all who need it. Implementing this concept was a bit more involved. Blaise generic in-depth data storage takes care of storing the Blaise data. The actual controls had to be developed by NASS. We only wanted to provide access to those who needed access. Identifying these needs was fairly straightforward at first, but then the lines began to get fuzzy.

The two obvious functions are data collection and interactive editing. Based on the entry in the CASIC_FAT table, each record is assigned a DCC_FIPS to give the data collection responsibility to a state, and an EC_FIPS to give the data editing responsibility to a state. Using record filters, the data can be logically separated. Now that the data is centrally located, HQ can play a much more interactive role. Special reports were created specifically for people in HQ, so an up-to-date picture of how a particular survey is progressing is available. Another benefit is the ability to look at the exact record of a state with which they may be having issues. It also allows someone in HQ to edit records for states.

Typically, the EC_FIPS state would be responsible for reading out the data (until WIP2, our analytical database, is fully functional), but since the data is centrally located, it made more sense for one readout at a national level. So the menus were created to do just that. Every rule seems to have exceptions, and the readout process was no different. For some surveys, the states wanted the readout control back, so they didn't have to wait on the whole country to get the data in which they were interested. This was accommodated by the menu script. An indicator was introduced to identify the party responsible for the readout. This is just one example of a number of tweaks made along our way.

The infrastructure is not yet ideal at NASS. We are 95% virtual, using Citrix. The country is split into two halves, east and west. The Blaise data servers, the MySQL database server, and the eastern Citrix servers are currently in HQ. The western Citrix servers are in Kansas City. This makes the distance for eastern Citrix users quite short; whereas, the distance for the western Citrix users is quite long. There is a noticeable difference in performance between the eastern and western users. Plans are to move all the servers to Kansas City in the middle of the country. This will make everyone equidistant and we are anticipating a significant performance improvement especially for the western users.

Figure 1 illustrates the current scenario for an eastern Citrix user. Response times are good because all of the servers, including the Citrix server, are very close physically.

Figure 1. Current Eastern Citrix Scenario

```
Workstation ← Local Area Network (LAN) → Blaise Data Server ← Local Area Network (LAN) → MySQL
                  (1 ms latency)                                 (1 ms latency)
```

Figure 2 illustrates the reason we have users on western Citrix that are suffering from the poor performance. The 30-millisecond latency is thirty times slower than the eastern Citrix performance when sending something from the workstation to the Blaise data server and thirty times slower when receiving something back from the Blaise data server.

Figure 2. Current Western Citrix Scenario

```
Workstation ← Wide Area Network (WAN) → Blaise Data Server ← Local Area Network (LAN) → MySQL
                  (30 ms latency)                                (1 ms latency)
```

Figure 3 illustrates our optimal server solution when they are all located in Kansas City. We don't anticipate much of an increase in performance, if any, for the eastern Citrix users, but the western Citrix users should see dramatic increases.

Figure 3. Future Eastern & Western Citrix Scenario

```
Workstation ← Local Area Network (LAN) → Blaise Data Server ← Local Area Network (LAN) → MySQL
                  (1 ms latency)                                 (1 ms latency)
```

The computer facility in Kansas City will also provide us with a more stable environment for the hardware. There is also staff dedicated to maintaining that environment.

4. Hybrid Surveys

We have a small number of surveys that have not made it through the entire conversion from decentralized to centralized. These tend to be our high-profile surveys with a very short data collection window. One night lost for calling, could derail the estimate due date. The hybrid survey approach was developed to handle these few surveys until the comfort level of moving them to fully centralized is acceptable.

The hybrid survey approach consists of the CATI data collection portion of the survey cycle to be decentralized in a typical Blaise dataset. Once data are collected, the completed forms are copied to the centralized dataset in MySQL. When the data arrive into the central dataset, the remaining survey processes are completed in the centralized environment.

The management of the hybrid surveys is much more disjointed than the centralized surveys. As our confidence has risen, plans to fully centralized these few surveys is underway.

5. Regionalization at NASS

Recognizing that resources are fewer and fewer, and the need to do more with less, NASS began plans to reorganize our field office structure. We have moved from forty-six field offices to twelve regional field offices. This is yet another change that has had a large impact on our CASIC system. We had already made the leap from decentralized surveys with up to forty-six physically separated datasets to one logically separated centralized dataset. All of the filters that we had built were dealing with data collection centers, estimation centers, and individual field offices. Now we have introduced regional field office filters. Three additional fields have been added to the shell code of our instruments to identify the region where the record belongs, the region responsible for data collection, and the region responsible for editing the record.

Since not all of our field office staff have relocated, the CASIC system has to detect where the user is sitting, and to which region they belong. We pick up the user's location from their Windows AD group membership. We then have a procedure to determine their region. The filters are built on the fly to filter the centralized dataset down to just their region. It is much more efficient to use the record filters rather than using an if-statement in the manipulate section. When a record filter is used, only the records that qualify for the filter are actually read by Manipula. If the control logic is done via an if-statement in the manipulate section, then all of the records will be read and evaluated. In some cases, this could mean a difference of reading 10,000 to 20,000 records and reading 200,000 records.

As stated earlier, there are always exceptions to the rule. For example, not all of our surveys have made the transition to being processed regionally. So our system has an indicator in the CASIC_SURVEYINFO table to make this determination. Since the record filters are built within the menu code, which is written in VB.NET, the menu can react to this indicator and build the appropriate record filter. The filters are read by the Manipula programs using an INCLUDE file. This allows us to have one Manipula program that is independent of how the survey is being processed.

6. Conclusion

NASS still has systems slated to move into a more centralized environment that will communicate with our CASIC system. As these systems are developed, we will need to continue to adapt and modify our system. Some of the bridges that we have built will be replaced with more efficient methods and direct links to other systems at NASS.

The waves of change will continue to reshape our current system. We can only hope that through open communication and collaboration with our fellow NASS developers, the emerging systems connect seamlessly with the new system designs developed up to this point. We also hope to be able to sunset some of our legacy systems, as our new systems mature and prove themselves. Blaise continues to play an important role at NASS. It is our only CATI software and will make up one of our two edit systems, as we are phasing out one of our old legacy edit systems.