Deploying an Automated Data Read In Process at National Agricultural Statistics Service

Roger Schou and Emily Caron, National Agricultural Statistics Service, USA

1. Introduction

The National Agricultural Statistics Service (NASS) uses multi-mode data collection including CATI, electronic data reporting (EDR, also known as CAWI), CAPI, and paper data collection. Blaise is best known as our primary data collection source for CATI, but it is also our one of the primary editing tools in NASS. The data from all other modes of collection need to make it into Blaise for editing. For many years, reading data into Blaise involved users manually kicking off a Maniplus program and then sitting there to watch it spin.

In 2010, we changed the way in which we stored our Blaise data. We moved from individual decentralized Blaise data sets in 42 separate locations to one centralized MySQL transactional database accessed by all states. All of our surveys' data are stored in the same set of tables in MySQL. We chose in-depth generic data storage so that the back end process which reads from the MySQL database and writes to a RedBrick analytical database could be one standard ETL (Extract-Transfer-Load) for all surveys.

As more and more surveys began using our new Centralized Blaise solution, the number of times the individual data read in processes ran also increased. With all of these users accessing the same resources to get data from the non-Blaise systems read into Blaise for editing, the performance quickly began to deteriorate. In May 2014 after realizing what was causing the problem, we redesigned the data read in process to run in a more automated fashion. This new process has greatly improved performance and user experience on many levels.

2. A Problem Arises

In the past when we were decentralized with 42 separate offices, reading data in from our non-Blaise sources was not an issue performance-wise. Each survey had its own proprietary Blaise database containing only records that belonged to that office. Everything was sitting on their Local Area Network, so the demand on resources was not very high. Users would initiate the read in process for that location and watch the process run from start to finish. Included in the read in process has always been an integral check (or batch edit) using the CHECKRULES method in Manipula. Even with this extra piece, the entire read in process was pretty fast since the data didn't have far to travel and not many users were competing for the same resources.

Once we centralized the data storage to a single MySQL database, we introduced the Wide Area Network (WAN) into our processes. We also began moving more and more surveys into the centralized world. The data read in functioned very similar to how it worked in the decentralized world: The user initiated the read in, sat there and waited while data was read into the centralized dataset and batch edited. Performance was not as good as it was with the decentralized environment, but it was still tolerable.

When some of our major surveys with very large sample sizes near 100,000 entered the picture, performance deteriorated to intolerable levels. Some locations were seeing read in times of nearly an hour for a batch of records. It even got to a point where we were assigning time slots for certain offices to run the read in process, since we feared concurrent read in jobs were causing problems.

The NASS infrastructure concerning Blaise changed as the number of surveys grew. In April 2014 we expanded from one to three Blaise Data Servers, splitting our surveys across the three servers. Any

given survey was housed entirely on one given server. This expansion eased the stress that we were once imposing on a single Blaise Data Server, but unfortunately we continued to experience unacceptable lags in the read in process. With some of the surveys having a short two-week turnaround from start to finish, something had to be done.

3. A Solution Is Formed

After doing some testing, we discovered that if we ran the read in process on the Blaise data server rather than from the client machine, it was significantly faster. This basically eliminated the WAN communication traffic. However, we were still thinking in a decentralized manner. We needed to step back and attack this from a centralized perspective.

Testing had also revealed that concurrent read in jobs run for the same survey were degrading performance. We decided it would be best to do a little footwork on the front end to combine all data needing to be read in from the various sources and numerous locations, and then WE would control the Manipula to read in the data and batch edit at a national level. This would eliminate concurrent read in jobs per survey, and we could also take advantage of overnight hours when no users were in the system.

4. Queueing the Data

CATI data is collected using the Blaise instrument, so it is available for editing the instant the interviewer completes the interview and exits the form. Our CAPI and EDR/CAWI data are collected in the same in-house system, so data is manually pulled for both of these at the same time and then read into Blaise. Paper data is keyed in a software called Viking before it's read into Blaise for editing.

We created a *Queue Files for Read In* process which users can run from a button on our VB.NET CASIC Menu when a batch of data is ready to be read into Blaise, whether it's EDR, CAPI, paper, or any combination of the three. This approach allows multiple users to stage their data into the queue, but the actual loading of the data into the database is done nationally. It runs at set times throughout the day, and runs completely on the server.

Users have two options for queueing data: daytime or overnight. The daytime read in process runs every fifteen minutes during the day from 6:00 am through 11:00 pm, seven days a week. The overnight read in runs at 12:15 am, Tuesday through Sunday. When a user clicks the Queue button, as long as the files are named correctly and sitting in a pre-determined location on a designated local drive, it will pick up and move the file(s) to the centralized Blaise Data Server. The file(s) will be renamed to include a date and time stamp and placed in either the QueueDay or QueueNight folder. These folders are located at the survey level on a given Blaise Data Server (i.e. \\KCBLZxx\casic\data\casic*surveyfolder*\hq\QueueDay\, where xx is the server number and *surveyfolder* is the folder name of a given survey). By adding the date and time stamp, the files are unique (to the second) which almost eliminates the possibility of someone's file getting overwritten by someone else's. It also allows for any number of files to be queued between program runs.

When a user queues data, the program will look for the existence of a "SurveyInfo.txt" file for that particular survey. This file contains pertinent metadata specific to each survey that the read in program will use, like instrument name, whether or not the survey allows new adds, etc. If the file doesn't exist or is older than today's date, the program will generate a new one.

A trigger file is also created during the queue up process. Its file name is made up of the name of the survey folder, the data source, and an extension of .TRG. It is placed in either the TriggerDay or TriggerNight folder, and the file name alerts the sweeping program as to which surveys to process and from what source they originated. The contents of this file contain the command line options to be

passed into Manipula. Multiple trigger files may be present, each representing a different survey and different data sources within surveys. The trigger folders are located at the server level (i.e. \\KCBLZxx\casic\data\casic\TriggerDay\).

5. Sweeping the Data

A VB.NET console application was written to accomplish the automatic data sweep. It is set up as a scheduled task on the Blaise Data Servers. By looking at the files present in the appropriate trigger folder, it will first move the trigger file into a TriggerStage folder and then automatically sweep the corresponding surveys' queue folder. To make this process even more efficient, we used threads to kick off multiple survey read in jobs at once. As long as data wasn't being read in concurrently for the same survey, performance wasn't degraded.

Since the application runs every fifteen minutes during the daytime and some large survey batches may take longer than fifteen minutes to process, it first checks for a file in the surveys' QueueStage folder called InUse.TXT. If it does already exist, that tells us the previous read in for that survey is still running so the queued data will be postponed until the next sweep. If the file does not exist, it creates it there for the surveys that have a trigger file present and the process continues to run.

The VB.NET application concatenates all instances of the EDR/CAPI data into one file and all instances of the paper data into another file for each survey and places them in the QueueStage folder. The TriggerStage and QueueStage folders are used to prevent interference of a user trying to queue data while the sweeping process is active. The program will still allow the user to queue data while the read in process is running, but that data will not be picked up until the next sweep.

Once the data has been concatenated, the console application executes the Manipula setup to read in the data and run integral check. If a record is already complete in the MySQL Blaise tables and the read in program attempts to read in the same record from a non-Blaise source, the system will detect the potential overlay and will write it to a proprietary Blaise dataset that sits off to the side. We then have a Maniplus program that will display these potential overlays next to the completed record in MySQL, showing a few important fields. The user then has the option to open either the overlay record or the original record to view the data. There are buttons that allow the user to keep the version they want, and the other one is deleted.

6. Log Files

The automated read in process appends to two separate log files so that we can ensure all is running smoothly behind the scenes, and if it's not we'll have a clue as to where the issue occurred and why.

One log is located at the server level. It creates log entries each time the auto read in scheduled task begins and ends with associated date and time stamps. If there are no trigger files present, then it states that fact in the log file. If there are trigger files, log entries are created showing each trigger file being moved to the staging area as well as the file size of the data file(s) to be processed. It also stamps the start time on the end of each line in parentheses, so it is easier to match up starts with endings in case any overlap each other. Any errors occurring at the high/server level of the program would be seen here. An example of some server level read in log entries can be seen in Image 1.

Image 1. Server Level Read In Log Example

```
2/2/2015 10:00:00 PM....        START of Read In process. (220000)
2/2/2015 10:00:00 PM....        END of Read In process - no triggers found for DAY read in. (220000)
2/2/2015 11:15:00 PM....        START of Read In process. (231500)
2/2/2015 11:15:00 PM....        BEEPDI140000edr trigger moved to staging area. (231500)
2/2/2015 11:15:00 PM....        BEEPDI140000raw trigger moved to staging area. (231500)
                                BEEPDI140000 total data size = 45121 bytes
2/2/2015 11:15:00 PM....        FLMPC150100raw trigger moved to staging area. (231500)
                                FLMPC150100 total data size = 577 bytes
2/2/2015 11:15:21 PM....        END of Read In process. (231500)
```

The second type of log file tracks the process at the survey level, so these logs are stored and accessed under survey specific directories. There are only entries in this log file when there are data files present for that particular survey. Very similar information is displayed in this file when there is data to be processed: Start and end time of the read in process for each type of source file, along with data size. If an error occurs at the survey level of the read in program, it will appear in this log in significant detail with a related log entry also written to the server level log with much less detail.

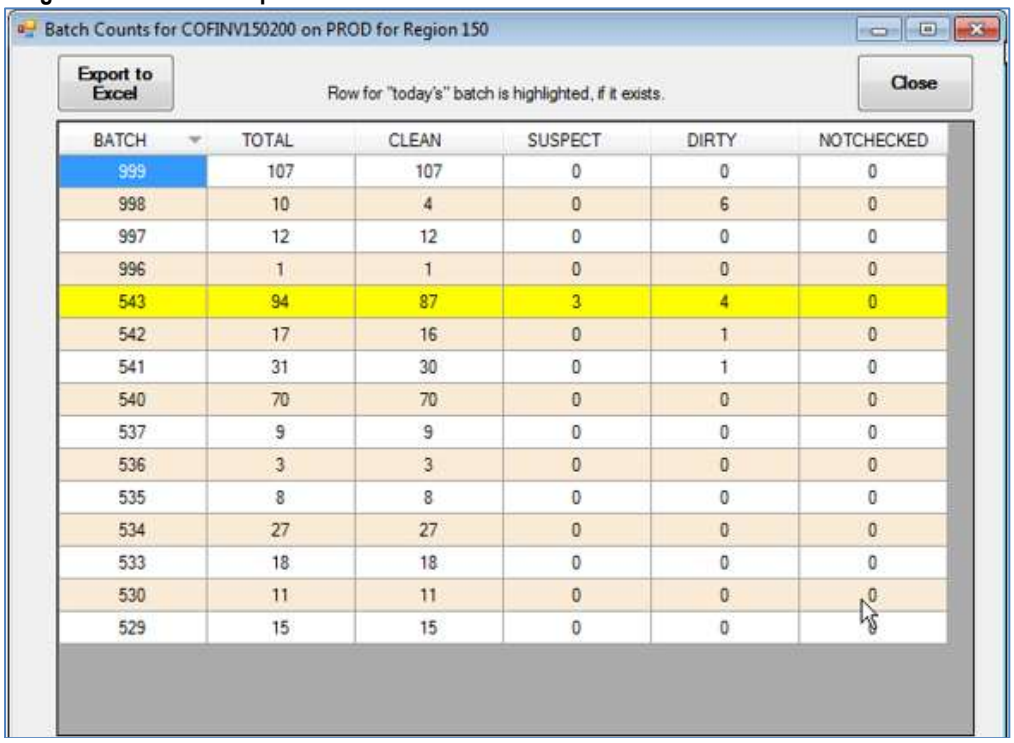
We have created buttons for staff in headquarters to review these log files. When the button is clicked, a copy of the log file is written to their personal temporary folder, so that the log files always remain available to the automated process. This was done to prevent the process from failing due to an inability to write to a file that is in use by someone reviewing it.

7. Improvement 1 - Batch Count Report

We use Blaise to interactively edit our data on many surveys, and the users prefer to edit them by batch. The CATI data is stored in batch numbers equal to the Julian date on which they were collected (1-366). For the auto read in, we decided to assign batch numbers by adding 500 to the Julian date for the day they were read in (501-866).

Even though the auto read in process runs every fifteen minutes, the users were concerned as to how they would know a batch had been read in. They were used to having total control of the read in process, watching it from start to finish and even assigning a batch number for each batch read in. In order to address this concern, we created a Batch Count Report as shown in Image 2.

Image 2. Batch Count Report



BATCH	TOTAL	CLEAN	SUSPECT	DIRTY	NOTCHECKED
999	107	107	0	0	0
998	10	4	0	6	0
997	12	12	0	0	0
996	1	1	0	0	0
543	94	87	3	4	0
542	17	16	0	1	0
541	31	30	0	1	0
540	70	70	0	0	0
537	9	9	0	0	0
536	3	3	0	0	0
535	8	8	0	0	0
534	27	27	0	0	0
533	18	18	0	0	0
530	11	11	0	0	0
529	15	15	0	0	0

This report is created in VB.NET by using a select statement against the MySQL tables. It lets the user know if a batch exists and the number of records within that batch, as well as the cleanliness of the records in the batch. Since most people do not know the Julian date for a given day, the current day's batch is highlighted if it exists, as shown in the image above. Also we added Julian date to our main menu screen for quick and easy user reference.

8. Improvement 2 – Record Level Rejects and Check Ins

Now that the read in program was taken out of the users' hands, we struggled with what to do with a report we used to generate out of the old read in program which alerted users to record level rejects. These rejects occur on surveys that do not allow new records to be added to the sample. For these types of surveys we have a set sample master and if a record that doesn't exist on the sample master attempts to be read in, the read in program will reject it. Often these errors are attributed to the primary key getting keyed incorrectly from paper forms, so the users would want to know about these errors as soon as possible to resolve the matter and get the correct record's data read in.

Another problem had us temporarily scratching our heads. When users were able to read in data at will the old way, this also accomplished records receiving a status of 'complete' in Blaise in a timely manner, which automatically removed the records from the CATI call scheduler. We do have other methods of checking non-Blaise records into Blaise to ensure they won't be called, but users worried about the rare cases when records had been missed being checked in via the other methods. If a user queued up data for the overnight run and it happened to get missed for check in, it could take hours for the record to be read into Blaise, therefore allowing the possibility for it to be called via CATI.

Both of these problems were solved by adding extra code to the queue data step. As users queue up their data files, a Manipula setup is run to check for non-matches between the data files and the sample master. If any are found, they are written to an output file which is presented to the user at the end of the queue up process. Another Manipula runs to ensure all records being queued up for read in are checked into Blaise, thus immediately removing them from the CATI Call Scheduler.

9. Improvement 3 – Item Level Rejects

In order to read in the data from the non-Blaise sources, we use a universal data layout with primary key information as well as an item code and the data. There is always a possibility that this item level data can be rejected.

In the past when users ran and watched the data read in, if item level rejects occurred the read in program would output a text file for them to view which contained this information. However, with the new automated process there was no one watching it run, and each run could contain any number of different users' batches. So how could we alert users about item level rejects?

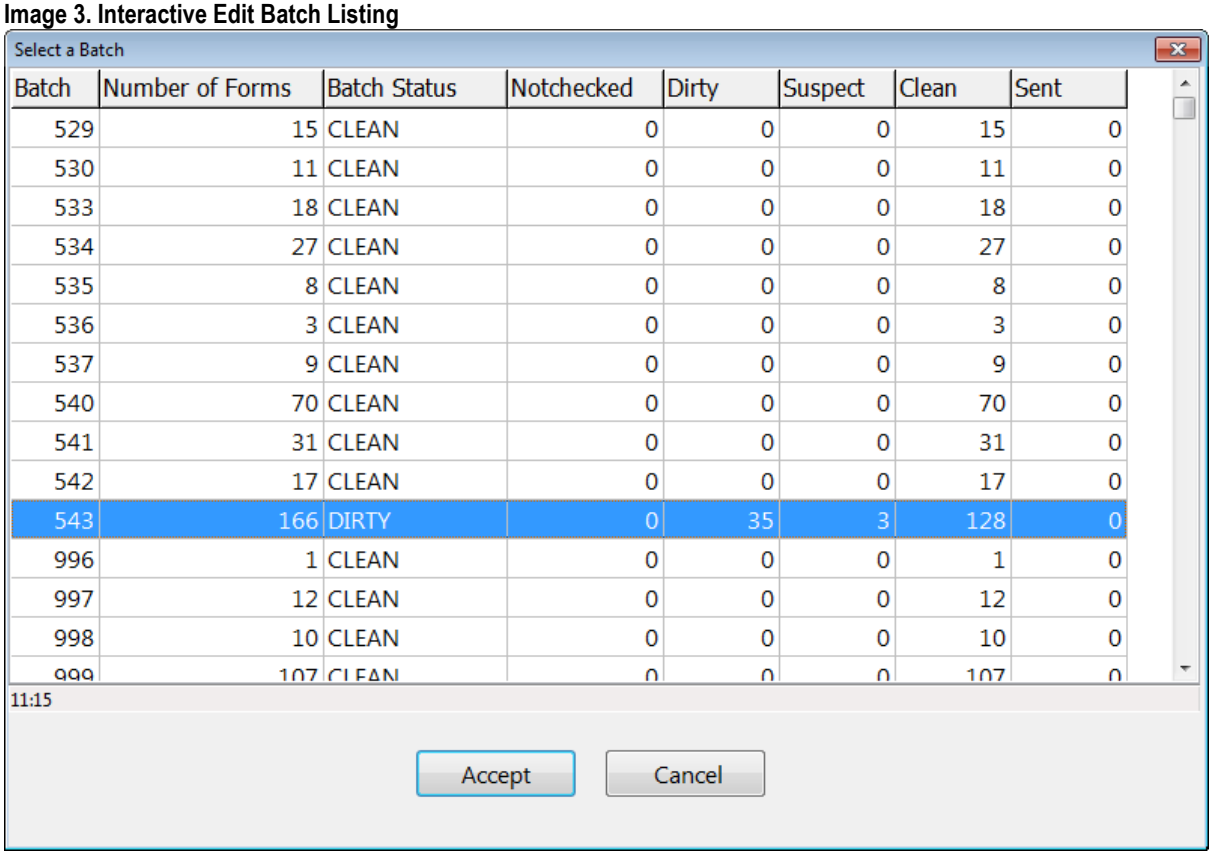
To address this in our new automated process, we created two fields in every Blaise instrument. One is an OPEN field that the auto read in process writes messages to when it encounters an invalid or blank item code, a duplicate item code, or an item code with a blank value. The second field is an enumerated field with one value for "Reviewed." There is a critical error check in every Blaise instrument that says if the enumerated field is not equal to "Reviewed", then the OPEN field must be empty. This forces the editors to review any bad item code data that failed to read in to the data set.

A text file is also created that lists all rejected data so that a user in headquarters may review them to potentially detect a systematic problem. The text file is a bit cryptic. We realize there is room for improvement on this file, as the headquarters staff are currently relying on the editors dealing with the critical errors to determine if there is a systematic problem.

10. Improvement 4 - Interactive Edit Sort Options

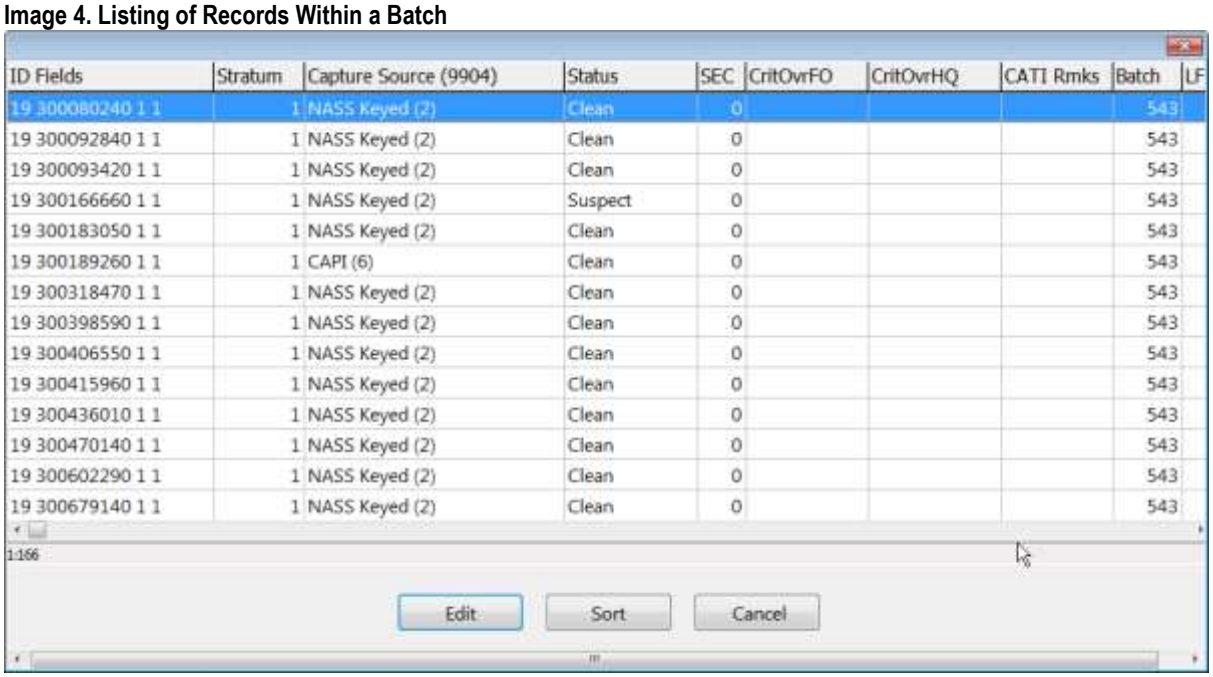
The interactive edit interface at NASS is written in Maniplus. It allows the user to search the database on batch number. It presents a dialog box as shown in Image 3, where it shows the number of forms within each batch as well as the status totals for the batches. The Batch Status takes into account the "dirtiest" status of any form within each batch. From this dialog box, the user can select the batch they would like to edit.

Image 3. Interactive Edit Batch Listing



Once a batch is selected, another dialog box is displayed showing the individual records in that batch as shown in Image 4.

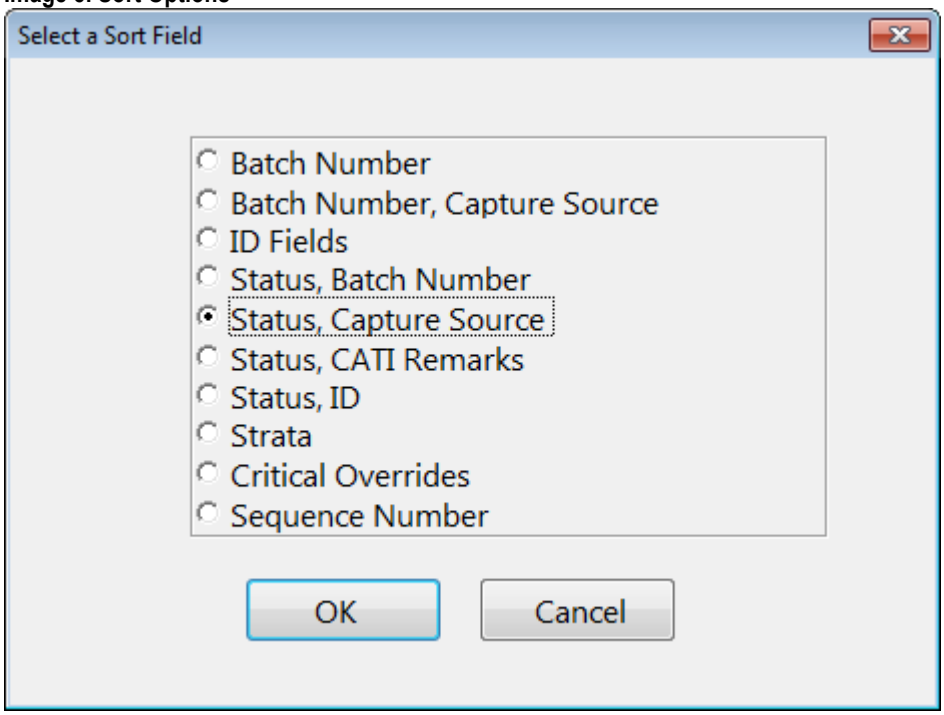
Image 4. Listing of Records Within a Batch



Since records being read in with the automated program are assigned a batch relative to the date, there suddenly came the possibility of a batch being comprised of records that were collected by any combination of paper, EDR/CAWI, and CAPI. In the past when the users had control of assigning batch numbers, they always kept the different modes separated because the way in which editors edit

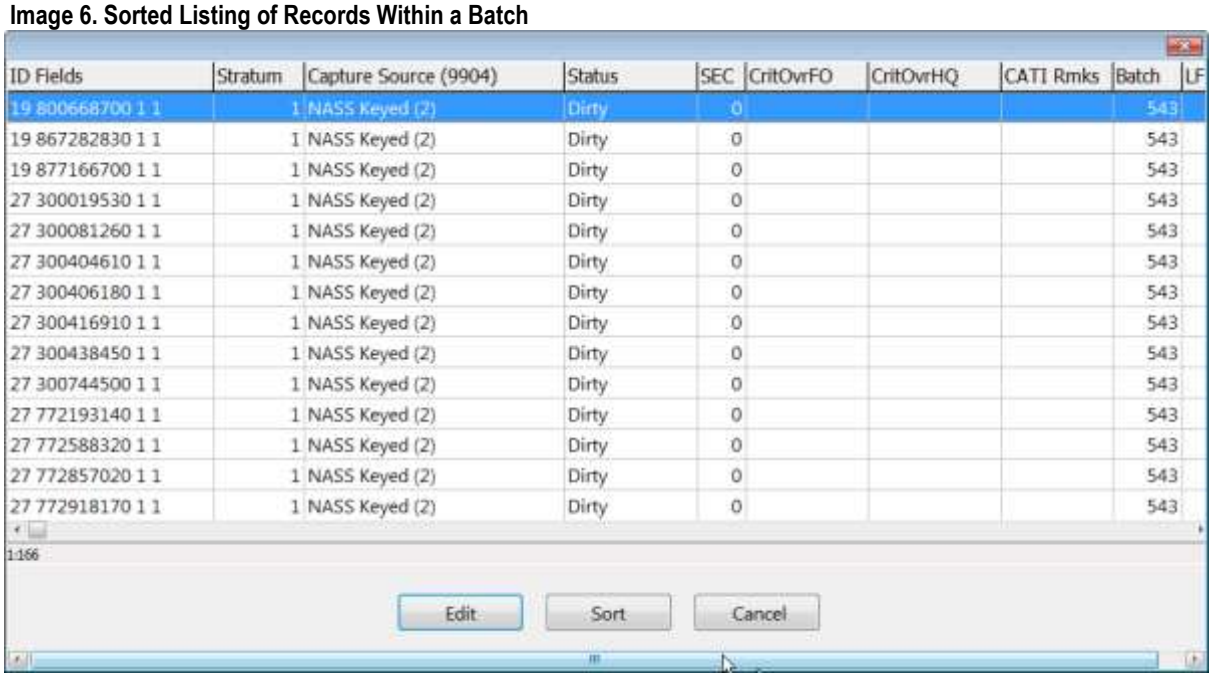
the records varies slightly between collection mode. We ended up adding a couple of new sort options that included Capture Source so that these records would sort together as shown in Image 5.

Image 5. Sort Options



Selecting the Status, Capture Source sort option groups the records by status and then capture source. This will bring the dirty records to the top of each group in the table to be edited as shown in Image 6.

Image 6. Sorted Listing of Records Within a Batch



11. Improvement 5 - Email Alerts

When we first started using the new auto read in program, we kept a close eye on the log files being generated to be sure everything was running smoothly. At first this worked out well, as there were

numerous bugs to work out of the system. However as time went on and the bugs were getting fewer and fewer, no one wanted to keep tabs on the log files anymore. But as with any automated program, there must be some type of notification system in case things go awry.

Modifications were made to the auto read in program to automatically send emails to our group whenever any of the following conditions are detected:

1) An InUse.txt file is detected, so the assumption is that the Manipula program is still running and the data will wait until next sweep.
2) A Manipula failed to compile or some other problem was encountered when trying to complete the auto read in process.
3) No data files were found associated with the trigger file(s).
4) A file already exists to which the data files were to be concatenated.
5) An error occurred when trying to concatenate the files.

Once an email is received, steps are taken to resolve the problem and get the files set back up for the next sweep.

12. Challenges

Ever since we first deployed the auto read in process, we received random COMexception errors which would cause the read in to fail. Data files would remain in limbo in the TriggerStage and QueueStage folders, halting the read in process for that particular survey until someone in our group manually re-queued the data and deleted the InUse.TXT file. Putting the same exact data files back in queue would do the trick and we might not see another COMexception error for days or even weeks. There didn't seem to be any rhyme or reason to when they would occur. They'd show up across different surveys, different times and days of the week, and across the various servers we use. We were never able to recreate the error on demand, because as mentioned even queuing up the same exact files again wasn't able to recreate the error.

Documentation was created to instruct members of our group how to properly re-queue the data files. This worked; however, manually re-queuing the data files was a general annoyance for everyone to have to deal with. Also if someone didn't follow the re-queue instructions to the letter, it would cause further headaches with the process.

We eventually added extra code to the auto read in program to detect the random COMexception error and programmatically handle all of the re-queuing steps for us. Image 7 below shows an example of the log entries created after receiving this error at 7:00:02. It took a mere 1 second for the program to re-queue the data files, which were then successfully read in the next time the process kicked off 15 minutes later.

Image 7. Sort Options

```
12/29/2014 7:00:01 AM....       START of EDR ReadIn Process for BEEPDI140000.
                                EDR data = 18737 bytes, Remarks = 689 bytes
12/29/2014 7:00:02 AM....       ERROR in ReadIn Function while survey = BEEPDI140000.
       System.Runtime.InteropServices.COMException (0x8000FFFF): Catastrophic failure (Exception from HRESULT: 0x8000FFFF (E_UNEXPECTED))
   at StatNetn.Blaise.Interop.BlPars38.IBlaiseParser.Parse()
   at AutoReadIn.ReadIn.ReadIn(String survey) in C:\VisualStudioProjects\CASICMenu\src\AutoReadIn\ReadIn.vb:line 834
12/29/2014 7:00:03 AM....       COMException error occurred while survey = BEEPDI140000.
12/29/2014 7:00:03 AM....       Only EDR data was detected for requeue process (no RAW).
12/29/2014 7:00:03 AM....       EDR data was successfully requeued for survey = BEEPDI140000.
12/29/2014 7:15:01 AM....       START of EDR ReadIn Process for BEEPDI140000.
                                EDR data = 18737 bytes, Remarks = 689 bytes
12/29/2014 7:15:07 AM....       END of EDR ReadIn Process for BEEPDI140000.
```

We would love to get rid of this error altogether, but in the meantime at least we have a workaround in place that allows us to live with it without too much hassle.

13. Future Plans

Seeing how successful the auto read in became made us stop to think about whether there are other manually run processes we could consider automating. Some possibilities include our check in and CATI/Non-CATI transaction programs.

Something else we are investigating is incorporating an automatic pull of the EDR/CAPI data into the beginning of the overnight auto read in process. This would allow our users to see the most current data coming in from these two modes first thing in the morning without needing to manually pull the data and queue it up for the auto read in process.

14. Conclusion

The day we converted from the old method of reading data into Blaise to the new automated process marked a significant day in recent NASS data collection and editing history. General Blaise performance magically improved; service freeze occurrences all of a sudden dropped drastically in number; and user satisfaction with the system greatly increased because they suddenly had extra minutes, and in some cases hours freed up from their days without having to watch the data read process tick away on their screens or deal with other sluggish processes.

The new auto read in program also freed up some of our group members' time. Where we were previously spending significant amounts of time tracking and troubleshooting service freezes and overall poor system performance, we could now get our minds focused on developing other system improvements that previously had been put on hold. This system enhancement also gave us some ideas and coding experience to start thinking about other related improvements we could make in the future.