Blaise Editing Rules + Relational Data Storage = Best of Both Worlds?

Richard Frey and Mike Rhoads, Westat

1. Introduction

Data editing is a critical part of the system lifecycle for survey research projects. Blaise offers flexible and powerful data editing capabilities, and performing editing within the Blaise environment allows us to continue to enforce the rules that were in force during data collection. While the native Blaise data storage format (bdb files) provides an efficient mechanism to support case-by-case editing, it is not as well-suited for supporting aggregate queries that would typically be expressed using SQL.

Since the Blaise Datalink facility now allows Blaise, using OLE DB, to access data in relational databases such as Microsoft SQL Server, it is now possible to use a single data storage format (relational database tables) to support both Blaise-based data editing activities and relational data queries. Blaise 4.8.1 also adds built-in versioning to Datalink, thus providing an off-the-shelf mechanism for tracking all changes made by the data editors. This paper describes our incorporation of these capabilities into a corporate data editing system.

2. Background

2.1 Data Editing Systems

Social science survey research is a process involving a wide range of tasks, including initial design and development, the actual collection of the data, and data delivery and analysis. Some form of post-collection review of the data is essential to ensuring the quality of the data and the reliability of the conclusions that can be drawn from the information. Recognizing this truth, Westat has worked over the years to continually improve the quality of its data editing systems and their integration into the overall project lifecycle (Dulaney and Allan 2001, Gowen and Clark 2007).

Since so much of our data collection is implemented using Blaise software, Blaise has also played a central role in our data editing systems. Blaise offers a number of advantages when used in this manner, such as allowing editors to review and update data either using the same screens seen by the interviewers, or using an alternate layout if that would be more effective for a particular study. The most critical advantage of Blaise, however, is its rules enforcement mechanism. Blaise data collection instruments are designed with range checks, skip patterns, and other checks to enforce consistency of data as they are entered by the interviewers; clearly we do not want this consistency to be lost at the data editing stage. By using Blaise for data editing as well as for data collection, with the original data model used as a starting point, we are able to maintain data consistency without having to reprogram all of the instrument's consistency checks in some other software.

While we have used Blaise as the foundation of our data editing systems, we have typically augmented it with other software, utilizing Manipula or the Blaise API as an interface as necessary. For instance, we have used Microsoft Access for such tasks as managing the status of cases as they move through the editing system and maintaining a data decision log to document all editing decisions. A more fundamental issue for us has been with the native Blaise data storage mechanisms. BDB files work extremely well for storing both simple and relatively complex survey research data, allowing efficient access to cases and the blocks and fields within them for both interviewers and data editors. The underlying storage architecture is not relational, however, and thus is not particularly well-suited for aggregate reporting requirements. In earlier versions of our data editing system, we supported these needs by maintaining parallel data stores in Blaise and in a database, making changes to the Blaise data and then propagating the changes to the database version of the data, either on request or on a scheduled basis. While this proved to be reasonably satisfactory, we still hoped that at some point we could eliminate the need for this redundant data storage.

2.2 Relational Data Storage and Blaise Datalink

2.2.1 The Basics

Fortunately, the Blaise Team at Statistics Netherlands also recognized the value in some circumstances of being able to store data in formats other than native Blaise BDB files, and over the past several versions of Blaise they have significantly enhanced the capabilities of Blaise in this area (Rouschen 2007). One of the biggest steps forward came with Blaise 4.6, when it became possible to use an RDBMS to store not just external data, but the actual questionnaire data itself, using a BOI file (which stores linkage information) in place of a native BDB file.

Another major improvement came with the introduction of “generic” data storage in Blaise 4.8. This supports versioning of the data, which is important in a data editing environment.

Simply put, the Blaise Datalink facility allows the data used by a Blaise data model to actually be stored in a non-Blaise format, such as Microsoft SQL Server, Oracle, or other relational databases. Datalink implements this functionality using Microsoft’s OLE DB (Object Linking and Embedding, Database) technology. This means that it is possible to implement Datalink for any data source for which an OLE DB driver exists. As a practical matter, the type of Datalink system that we required is much easier to implement for a subset of the most common relational data sources, for which Blaise takes into account the characteristics of the particular data source, such as the maximum number of columns it can have in a single table. Blaise does provide such support for SQL Server, which is the database that we are using for our editing system.

Datalink supports four distinct types of BOI files. For our purposes, we use the type of BOI file that is created for an existing Blaise dictionary (data model). This type of BOI file supports not just basic field information, but also field attributes (don’t know, refused), remarks, form information, and block information. It thus can be used as the basis for a data model that supports data editing (or data collection, for that matter), exactly as a BDB file could be used.

2.2.2 Data Partition Types

Once you have decided to use this type of BOI file, you need to determine which of the five “data partition types” you want to implement. The data partition type determines exactly how tables are set up within the relational database to store the Blaise data. We basically used a process of elimination to determine which data partition type to use for the editing system.

The “stream” data partition type stores the data for each Blaise block as a single large column of data in the database—as a binary “blob,” if you will. This maximizes the efficiency of processing by Blaise, since the data for the block is already in the exact binary form that Blaise requires. However, these data blobs can only be decoded by Blaise itself, which did not meet our requirement for a data storage mechanism that could support SQL queries outside of Blaise in addition to within-Blaise access.

Datalink also offers “In Depth” and “In Depth Text” data partition types. Each of these types stores each Blaise field as a row in a database table; “In Depth Text” stores all values in a single text column, while “In Depth” has separate columns for numeric, string, and datetime data. The primary advantage of this storage mechanism is that it minimizes the number of tables and columns required to store the data, while storing each data value in a manner that is somewhat more accessible to non-Blaise software than is the stream type discussed above. Nevertheless, this transposed storage format is still not a good fit for the way most programmers and analysts design report queries. Therefore, we decided that these two data partition types did not meet our requirements particularly well either.

The “Flat No Blocks” data partition type is almost the complete opposite of the “In Depth” partition types. The latter results in a very tall, skinny data store: one table with only a few columns but a large number of rows, since each nonempty field is stored in a separate row of the table. The “Flat No Blocks” type, on the other hand, is an extremely short but wide storage mechanism: there is a distinct column for each iteration of each field (so a block with 5 fields that repeats 20 times would produce 100 columns), and a single row for each Blaise form. In theory, all of these columns would be in a single database table. With all but the smallest data models, however, they are spread out over multiple tables, since databases limit the number of columns that a single table can contain.

The disadvantage of this design for non-Blaise queries and reports is that it is not normalized. Imagine, for instance, that the 5-by-20 block mentioned above is a household enumeration. Locating females in the household would require a query that referenced each of the fields Sex1 through Sex20. Accordingly, we discarded “Flat No Blocks” from consideration as well.

This left us with the “Flat Blocks” data partition type. With this structure, a separate database table is created for every block type in the data model, containing a column for each Blaise field in the block. Each row in the table represents an actual occurrence of the block in the data, e.g. a person enumerated within a household. Each table also contains fields to identify the Blaise form and the instance number of the row within the block.

This data partition type is clearly the closest to the block structure of the Blaise data model, and thus provides the most suitable layout for “natural” SQL queries. For instance, each person enumerated within a household will be a separate row within the appropriate table in the relational database. If we want to obtain some information about all of the females who were enumerated, we can use the single field Sex for filtering, rather than having to use multiple fields as we would with the Flat No Blocks partition type.

For these reasons, we adopted the Flat Blocks data partition type as the basis for the data storage in the latest version of our corporate data editing system. There are some nuances of this data storage arrangement that need to be taken into account when designing queries and reports, and these are discussed later in this paper.

2.2.3 Generic Data Storage and Versioning

With Blaise 4.8, Statistics Netherlands further enhanced its Datalink capabilities by adding support for "generic" data storage. This technique is intended to reduce the burden of RDBMS administration in an environment where there are many surveys (such as a call center) by minimizing the number of tables that need to be created when a new survey is added. This is accomplished by using a set of common database tables that can store data for more than one survey (Blaise data model). Arno Rouschen's 2007 IBUC paper provides a fuller explanation of generic data storage in Blaise 4.8.

It is important to note that generic data storage is not a "data partition type." Rather, it is a separate choice that you make when setting up a BOI file to use Datalink. Generic data storage can be utilized with any of the five data partition types discussed above. However, if your goal in choosing generic data storage is to minimize the number of tables in your database (and thus administration burden), some data partition types are much more compatible with this goal than others. In particular, if you choose the In Depth or In Depth Text data partition type along with generic data storage, you can implement new data models under Datalink without requiring any changes whatsoever to the structure of the database that is hosting the surveys.

The Flat Blocks data partition type that we are using for the Westat data editing system does not take full advantage of this capability; new tables must still be created in the database corresponding to each block in every data model. This was not a problem for us, since reducing database administration burden was not a particular concern. In fact, given the large number of different clients for which we carry out data collection, combining data from different surveys within the same database or data tables would create more difficulties for us than it would solve.

The reason that we did decide to utilize generic data storage for our editing system is that it is required to support the versioning capabilities of Datalink, which were also added to the system in Blaise 4.8. When versioning is enabled, Blaise maintains both the old and new versions of records in the database whenever a record is changed, using a special timestamp variable (BEGINSTAMP) to track the history of the record. This process is completely transparent to those who are using DEP or Manipula to access the instrument; they only "see" the latest version of each record. Since all versions are maintained in the database, however, they are accessible to SQL queries, as well as to the new Blaise Data Center tool. In a data editing environment, having the capability to track all changes to the data is obviously attractive.

3. Implementing Datalink for Our Data Editing System

The remainder of this paper discusses how we implemented the Blaise Datalink facility for our corporate editing system. We begin by summarizing the capabilities and characteristics of the system. We then discuss the data storage architecture of previous system versions and show how Datalink greatly simplified our data storage strategy. We describe how Datalink stores the data within the relational database and present some issues to keep in mind when designing SQL queries for the data. We conclude by briefly discussing some tools we found useful when deploying and using the system.

3.1 Editing System Capabilities and Characteristics

As described in more detail in the next section, we incorporated the relational database storage capabilities offered by Blaise Datalink into the latest version of Westat's Blaise Editing System (BES). This system provides a variety of data editing, reporting and processing options to projects. It includes a set of core capabilities and can be customized to meet the needs of specific project applications. Figure 1 shows the main menu for one implementation of the system.

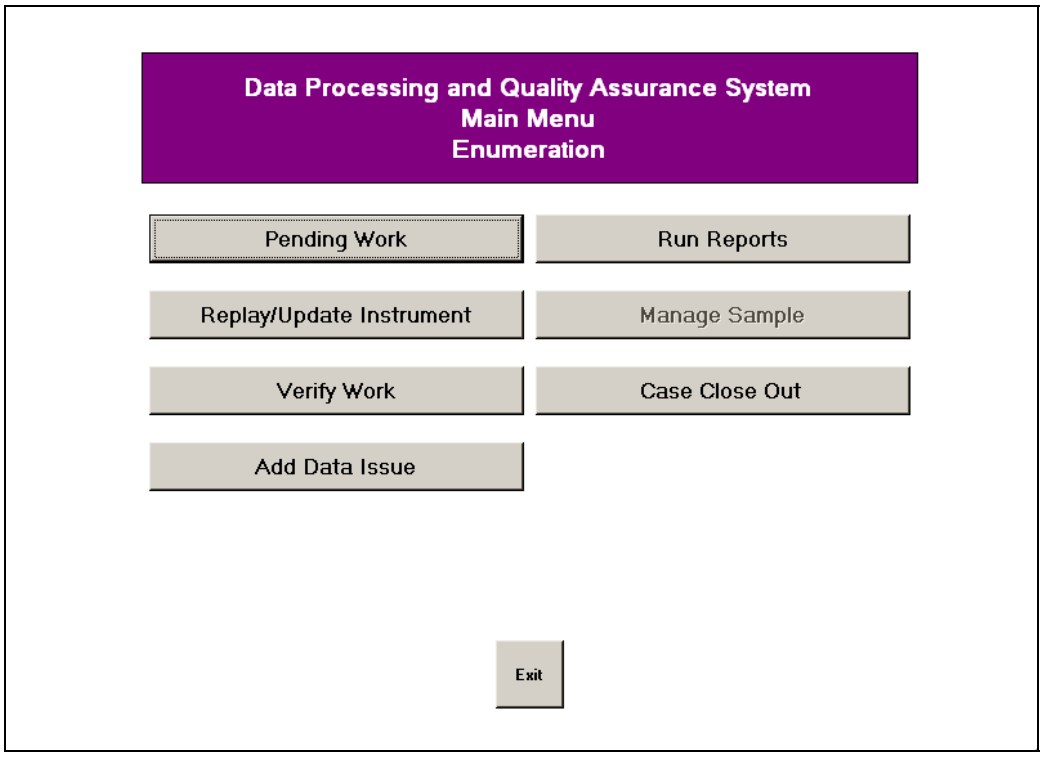


Figure 1. Blaise Editing System main menu

While most of the data processed by the system has been collected using Blaise, this does not have to be the case. For instance, we have used the system to review and edit data that originate in Cardiff's TeleForm software, which we use for paper document capture and processing. Since multi-mode studies may include CAI and hard copy data collection components, this capability allows projects to apply consistent quality control and edit processing regardless of the data collection mode.

In the latest version of the system, all data coming into the system are loaded into SQL Server so that we can take advantage of the Blaise Datalink functionality. The data model used to collect the data (CAPI, CATI or web) is replicated in the home office editing system with the SQL database. Editors review interviewer comments and other specify responses and make updates by replaying the Blaise instrument. Using Blaise for all data changes simplifies the data update process and improves data quality by preventing the unintentional violation of skip patterns during the data cleaning process.

Updates made through interview or form replay are stored in SQL Server, and data decisions are recorded in another database table. The BES decision log module permits projects with requirements to record the reason for data updates to maintain this information in a centralized location while editing is in progress. BES maintains all versions of the data, although only the latest version can be viewed by editors. Audit trails can be activated to recreate prior data history.

BES also provides a variety of reports, covering a wide spectrum of functionality as requested by the project. Reports can present details on workflow monitoring, case edit results, list cases, frequencies and cross-tabs, coding reports, and reports of field comments and data issues. The system integrates with other stages of the project lifecycle by permitting users to view PDFs of codebooks and data dictionaries, and annotated questionnaires or other helpful documentation.

Metadata are maintained using a data dictionary interface. With each new instrument release, the data dictionary is updated with the latest information from the Blaise or Teleform files.

The system normally operates on client computers connected locally to Westat's corporate network. It can also support remote access, with a security portal managing role-based access as appropriate to the user. VB.NET is the primary development tool we used for the system, in addition to Blaise and SQL Server.

3.2 Moving From Many to One – A Simplified Data Storage Architecture

As discussed above, one of the requirements of our editing system was to be able to record and track all changes that were made to data values. This is not an easy thing to do in native Blaise; when a change is made, the previous value is gone. Previously, in order to implement the change tracking capabilities, we wound up using multiple data stores, in BDB and Microsoft Access formats, for each instrument. The data flow for this earlier version of the BES is illustrated in Figure 2.

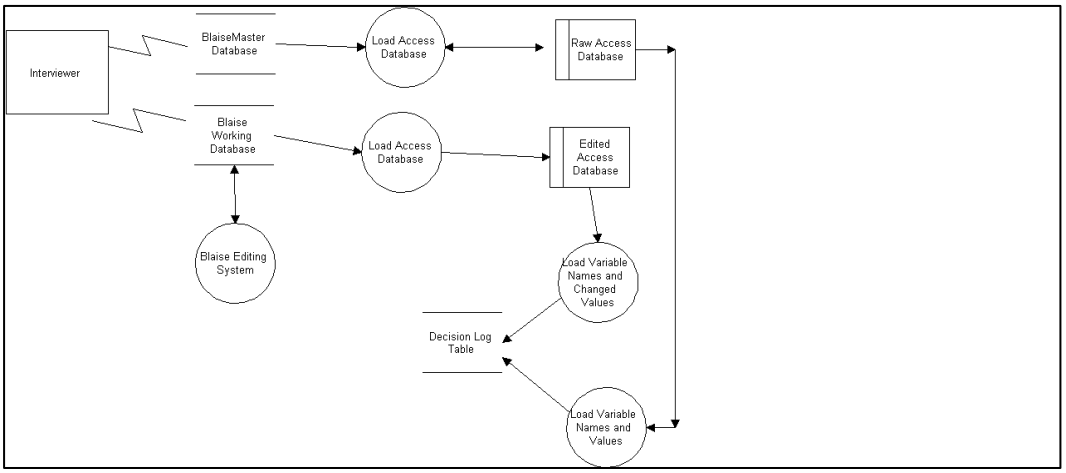


Figure 2. Data flow for previous BES

As data collected by an interviewer arrived for the system to edit (typically in a temporary Blaise BDB file), two separate paths were followed. In order to maintain an easily accessible version of the "original" data (i.e. all values as they were entered by the interviewer), the data were stored in a Blaise Master Database whose values were never changed. From this database, the initial data were also extracted into a Microsoft Access database, and then loaded into the Decision Log Table. The latter is a "vertical" table, with the values of each instrument field being stored as separate rows in the table.

As represented by the lower path on the data flow, the raw data were also stored into a Blaise Working Database. This is the database that was used by the editing application. Once each case was cleaned, the data were extracted into an ASCII file and then stored into the Edited Access Database. Then, all of the changed data from this database were extracted and stored in the decision log table, which is what we used to track data changes.

While this system worked reasonably well, we felt that the architecture could be improved. As the discussion above indicates, the system required a number of different data stores, and data transformations were required between each data store. We used a variety of applications to develop these transformations, including Manipula, Cameleon, the Blaise API, Visual Basic, Microsoft Access, and the SQL Server import/export wizard. Although the transformations were not particularly complex, they did require us to develop and maintain a large volume of code.

Implementing Blaise Datalink with its versioning capabilities allowed us to greatly simplify the design of the system. Rather than having the large number of heterogeneous data stores and storage formats that characterized the previous version of the system, we are able to use SQL Server for all of our data storage needs. We now require a relatively limited number of tables, which can reside within a single SQL Server database.

We did decide to maintain a "vertical table" for change tracking, in addition to the tables that are provided by Blaise Datalink. The Blaise implementation of versioning inserts a complete new set of rows into the database tables whenever any values for a case are changed. This made it difficult to identify the changes in an automated fashion; we found ourselves writing a considerable amount of code to compare rows, dates, and variable values in order to determine which variables and values were changed. Our previously-developed vertical table approach allowed us to store all information about changes in a central location, thus greatly simplifying the task of reporting on data modifications. We streamlined our earlier approach by only storing the variable names and values that changed, rather than all variables. This improvement greatly reduced the number of entries in this table – more than tenfold in some situations, depending on the size of the instrument.

We started to develop Visual Basic modules to keep track of the cases being worked, identifying and extracting all changes from the database tables once updates had been completed. We then realized that triggers and stored procedures within SQL Server would provide a more elegant approach. Knowing that Blaise inserts new rows whenever a data change is made, we created an insert trigger for each database table. When the trigger fires, it calls a stored procedure that compares the values of variables from the "current" row to those in the row representing the previous version of the data. It also takes into account Don't Know and Refused entries. For each difference that is found, we insert a new row into the vertical table that includes the case ID, variable name, new value, and the login id of the editor who made the change.

```
create trigger [Place Instance Schema Name Here].Place Table Name Here_Update
  on [Place Instance Schema Name Here].[Place Table Name Here]
  for insert as
begin
  if exists (select table_name
             from Information_schema.tables
             where table_name='Place Table Name Here_Inserted' and
                 table_schema='Place Instance Schema Name Here')
  begin
    drop table [Place Instance Schema Name Here].[Place Table Name Here_Inserted]
  end
  select a.*
    into [Place Instance Schema Name Here].[Place Table Name Here_Inserted]
    from (select * from inserted) as a

  exec [dbo].usp_DPQA_DataUpdates 'Place Instrument Name Here',
                                                  'Place Table Name Here','Place Key Name Here',
                                                  'Place Instrument Schema Name Here',
                                                  'Place Instance Schema Name Here'

end
```

Figure 3. Trigger template

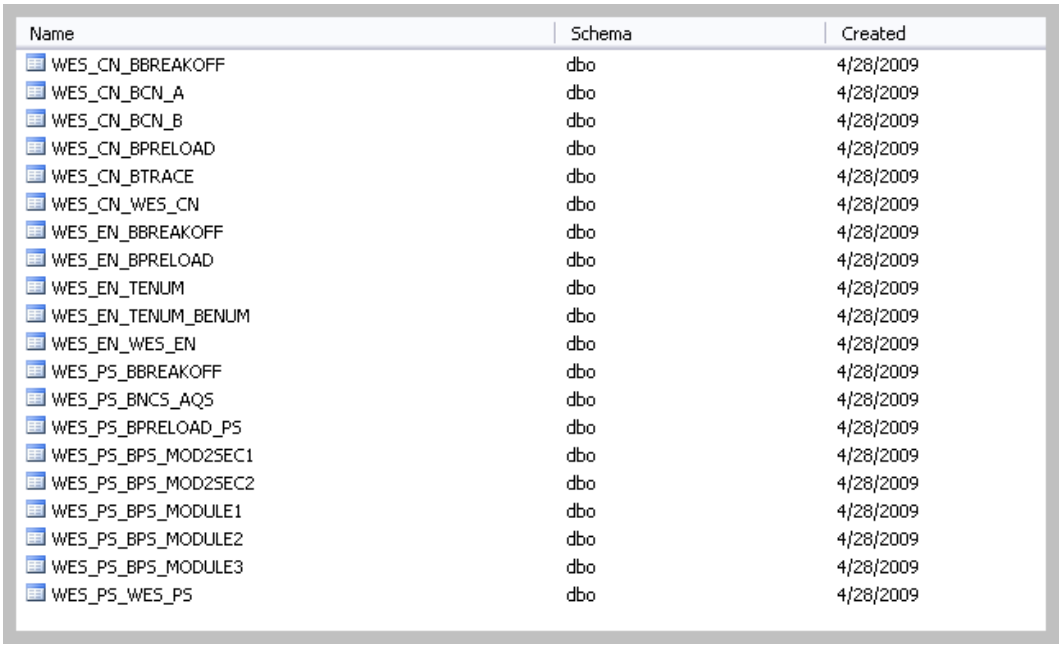
Figure 3 above shows the template that we use to create the triggers for each of the project tables, of which there may be 50-200 or more. As we discuss later in this paper, we created a number of tools to expedite the process of setting a new project up in BES, one of which automated the process of creating these triggers.

3.3 How Datalink Stores the Data

As described earlier, Blaise Datalink offers a number of data storage options (data partition types), and it also allows you to request "generic" storage. The way in which in stores data within your database depends on your choices when you set up your BOI file. For the editing system, we use the Flat Blocks data partition type with the generic data storage option (so that we can take advantage of versioning).

This combination of options results in a "hybrid" storage model, featuring some tables that are specific to the data model and some that are truly generic. In order to query the data outside of Blaise, you need to understand both sets of tables and how they interact.

For the Flat Blocks data partition type, the actual Blaise data are stored in Instrument Tables, which are specific to a Blaise data model. Figure 4 shows the tables for one implementation of the editing system. This implementation contains three data models: WES_CN, WES_EN, and WES_PS. Each table's name begins with the name of the data model, and there is one table for each block that is defined within the model.

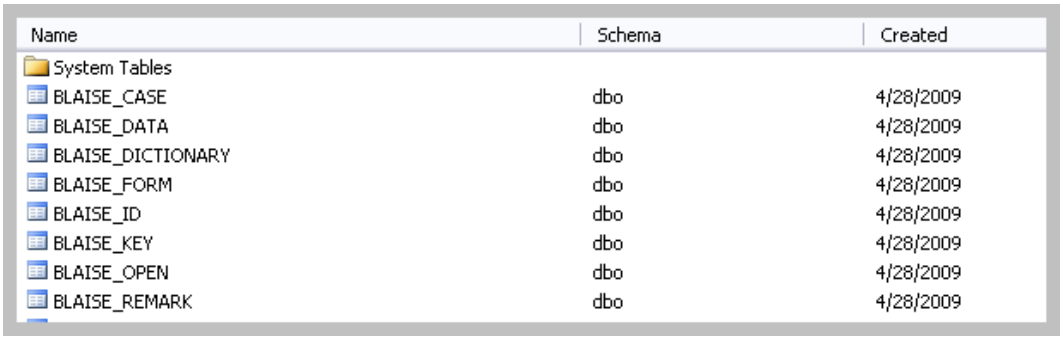


Name	Schema	Created
WES_CN_BBREAKOFF	dbo	4/28/2009
WES_CN_BCN_A	dbo	4/28/2009
WES_CN_BCN_B	dbo	4/28/2009
WES_CN_BPRELOAD	dbo	4/28/2009
WES_CN_BTRACE	dbo	4/28/2009
WES_CN_WES_CN	dbo	4/28/2009
WES_EN_BBREAKOFF	dbo	4/28/2009
WES_EN_BPRELOAD	dbo	4/28/2009
WES_EN_TENUM	dbo	4/28/2009
WES_EN_TENUM_BENUM	dbo	4/28/2009
WES_EN_WES_EN	dbo	4/28/2009
WES_PS_BBREAKOFF	dbo	4/28/2009
WES_PS_BNCS_AQS	dbo	4/28/2009
WES_PS_BPRELOAD_PS	dbo	4/28/2009
WES_PS_BPS_MOD2SEC1	dbo	4/28/2009
WES_PS_BPS_MOD2SEC2	dbo	4/28/2009
WES_PS_BPS_MODULE1	dbo	4/28/2009
WES_PS_BPS_MODULE2	dbo	4/28/2009
WES_PS_BPS_MODULE3	dbo	4/28/2009
WES_PS_WES_PS	dbo	4/28/2009

Figure 4. Instrument tables

Each of these Instrument Tables contains one column for each non-block field defined within the block, along with a set of 5 special columns that comprise the primary key for the table. JOINKEY and DMKEY uniquely identify a given Blaise case or form, while BEGINSTAMP distinguishes among different versions of the same form. Of course, a block type may be used for more than one field in a data model, and it is common to define an array of blocks. The BLOCKID and INSTANCE fields are used to differentiate these repeating occurrences of block data within a form.

Blaise Datalink, when generic storage has been chosen, also uses a set of common Generic Tables, which are shared among all of the data models in the database. These are shown in Figure 5. Most of them also contain the JOINKEY, DMKEY, and BEGINSTAMP columns, which are the key (pun intended) to joining records from the various instrument tables and generic tables.



Name	Schema	Created
System Tables		
BLAISE_CASE	dbo	4/28/2009
BLAISE_DATA	dbo	4/28/2009
BLAISE_DICTIONARY	dbo	4/28/2009
BLAISE_FORM	dbo	4/28/2009
BLAISE_ID	dbo	4/28/2009
BLAISE_KEY	dbo	4/28/2009
BLAISE_OPEN	dbo	4/28/2009
BLAISE_REMARK	dbo	4/28/2009

Figure 5. Blaise Generic Tables

3.4 Querying the Data

The Flat Blocks data partition stores data in a relatively natural way. For instance, if you have a BJob block in your Commute7Mod data model whose fields include EmpName and Distance, you will wind up with a database table called COMMUTE7MOD_BJOB that will have a corresponding set of columns. In spite of this superficial simplicity, however, there are some important things to keep in mind when querying Datalink data from outside of Blaise.

3.4.1 Distinguishing Current from Historical Records

Remember that when the versioning function of Datalink is turned on, Blaise saves all versions of the data, not just the most recent. While this is desirable when you want to keep track of changes, you have to be sure to filter your SQL queries when you only want to process the latest version of the data.

Figure 6 shows the COMMUTE7MOD_BJOB table records from a simple data model. Note that there are two sets of records for the case with JOINKEY=1 and DMKEY=1, distinguished by their BEGINSTAMP values. The records with the latest values for BEGINSTAMP are current, while the others are historical. (Also note that the EMPNAME for the job with BLOCKID=15 and INSTANCE=1 was changed from CDC to NCHS.)

	JOINKEY	DMKEY	BEGINSTAMP	BLOCKID	INSTANCE	EMPNAME	DISTANCE
1	1	1	2009-04-25 16:46:49.803	4	1	Westat	2
2	1	1	2009-04-25 16:46:49.803	6	2	McDonalds	7
3	1	1	2009-04-25 16:46:49.803	15	1	CDC	12
4	1	1	2009-04-26 13:59:43.520	4	1	Westat	2
5	1	1	2009-04-26 13:59:43.520	6	2	McDonalds	7
6	1	1	2009-04-26 13:59:43.520	15	1	NCHS	12
7	2	1	2009-04-26 13:50:26.370	4	1	Universal Studios	4
8	2	1	2009-04-26 13:50:26.370	6	2	Disney	27
9	2	1	2009-04-26 13:50:26.370	26	1	Waiter on the ...	17

Figure 6. Table with current and historical data

The primary mechanism that Blaise Datalink uses to track record versions is the generic table BLAISE_FORM. Figure 7 below depicts this table for the same datamodel and set of data. You can see that there are two records for the case with JOINKEY=1 and DMKEY=1, indicating that two different versions of that form are represented in the database, while there are only single versions for the cases with JOINKEY values of 2 and 3. Blaise uses a special value of ENDSTAMP, 9999-12-31 00:00:00.000, for rows in this table that correspond to current data. Therefore, the JOINKEY, DMKEY, and BEGINSTAMP values for such rows can be used to identify "current" data rows in any of the other tables. For other rows in BLAISE_FORM, the BEGINSTAMP and ENDSTAMP values can be used if desired to identify the time period during which records with corresponding JOINKEY / DMKEY / BEGINSTAMP values were valid. ENDSTAMP, in other words, effectively acts as an expiration date.

	JOINKEY	DMKEY	BEGINSTAMP	ENDSTAMP	STATUS	COLLECTMODE	DATAENTRYBEHAVIOUR	STREAMDATA
1	1	1	2009-04-25 16:46:49.803	2009-04-26 13:59:43.520	1	-1	3	0xFFFE5600450052005300490004...
2	1	1	2009-04-26 13:59:43.520	9999-12-31 00:00:00.000	1	-1	3	0xFFFE5600450052005300490004...
3	2	1	2009-04-26 13:50:26.370	9999-12-31 00:00:00.000	1	-1	3	0xFFFE5600450052005300490004...
4	3	1	2009-04-26 14:17:29.240	9999-12-31 00:00:00.000	1	-1	3	0xFFFE5600450052005300490004...

Figure 7. BLAISE_FORM table

3.4.2 Remarks, Open Text, Don't Know, and Refused

Interviewer remarks and open-text fields are not stored in the Instrument Tables. Instead, they are stored in the generic tables BLAISE_REMARK and BLAISE_OPEN, respectively. Figure 8 shows a sample remark record from our Commute7Mod data model.

	JOINKEY	DMKEY	BEGINSTAMP	FIELDID	REMARKTEXT
1	3	1	2009-04-26 14:17:29.240	9	Not sure how Junior can be 10 years old -- and the only person in the HH

Figure 8. Sample remark record

We see that this table contains a REMARKTEXT column, our usual trio of ID variables (JOINKEY, DMKEY, BEGINSTAMP), and a column called FIELDID. This latter column can be used to identify the field for which the remark was entered, in association with the BLAISE_ID table. This table associates a numeric identifier with each block and field in your data model, making it the equivalent of the Rosetta Stone for understanding how your data are stored. (Note that the creation of this table is optional when creating a new BOI file – be sure to

include it if you want to access your data outside of the Blaise environment!) Figure 9 illustrates a portion of this table.

	DMKEY	ID	TYPE	NAME
1	1	6	F	HHSize
2	1	7	F	Person[1].Name
3	1	8	F	Person[1].Gender
4	1	9	F	Person[1].Age
5	1	10	F	Person[1].MarStat

Figure 9. Selected rows from BLAISE_ID table

As can be seen from the example, the field that is referenced in the BLAISE_REMARK table (ID value of 9) is Person[1].Age. This makes it relatively easy to join these two tables to identify the fields that correspond to each remark.

The text for Blaise fields with type Open are stored in a similar manner, using the BLAISE_OPEN table.

Blaise field attributes (Don't Know, Refused) are also stored in a generic table, rather than in an instrument-specific table. In this case, the BLAISE_DATA table is used, as is shown in Figure 10.

	JOINKEY	DMKEY	BEGINSTAMP	FIELDID	STATUS	STRINGDATA	INTEGERDATA	FLOATDATA	DATETIMEDATA
1	1	1	2009-04-25 16:46:49.803	46	8	NULL	NULL	NULL	NULL
2	1	1	2009-04-25 16:46:49.803	50	4	NULL	NULL	NULL	NULL
3	1	1	2009-04-26 13:59:43.520	46	8	NULL	NULL	NULL	NULL
4	1	1	2009-04-26 13:59:43.520	50	4	NULL	NULL	NULL	NULL
5	2	1	2009-04-26 13:50:26.370	10	8	NULL	NULL	NULL	NULL

Figure 10. BLAISE_DATA table

Since we are using the Flat Blocks data partition type, the last four columns of this table will always be null. (When one of the In Depth partition types is in effect, these columns are used to store the actual data values.) As with BLAISE_REMARK and BLAISE_OPEN, FIELDID identifies the field in question, while JOINKEY, DMKEY, and BEGINSTAMP identify the case and version. STATUS contains a numeric value corresponding to the field's attribute; Don't Know values are represented by a STATUS value of 4, while Refusals produce a value of 8. Only fields that have a special attribute for a given case have rows in this table.

As with remarks and open text, it is relatively easy to link the BLAISE_DATA and BLAISE_ID tables to identify which Blaise fields have Don't Know or Refused values for a particular case. Unfortunately, if you want to issue a SQL query that shows all of the field values for a particular block, pulling in the attribute information is not nearly as simple. Such columns will have null values in the corresponding instrument table for the block, but we might want to know specifically whether the response was Don't Know or whether it was Refused. Repeating blocks complicate this even more, since each instance of a field will have a separate FIELDID value.

Although cumbersome, the information in the BLAISE_ID table does make it possible to construct such queries. We can take advantage of the fact that the ID values of blocks and fields in the table are assigned in a regular pattern. For instance, for the Marstat field in our BPerson block, we can query the BLAISE_ID table to determine the id values for that field and for its block, as shown in Figure 11.

	DMKEY	ID	TYPE	NAME
1	1	3	B	Person[1]
2	1	14	B	Person[2]
3	1	25	B	Person[3]
4	1	36	B	Person[4]
5	1	47	B	Person[5]
6	1	58	B	Person[6]
7	1	69	B	Person[7]
8	1	80	B	Person[8]
9	1	91	B	Person[9]
10	1	102	B	Person[10]
11	1	10	F	Person[1].MarStat
12	1	46	F	Person[2].MarStat
13	1	82	F	Person[3].MarStat
14	1	118	F	Person[4].MarStat
15	1	154	F	Person[5].MarStat
16	1	190	F	Person[6].MarStat
17	1	226	F	Person[7].MarStat
18	1	262	F	Person[8].MarStat
19	1	298	F	Person[9].MarStat
20	1	334	F	Person[10].MarStat

Figure 11. ID values for Person and MarStat

Noting that the ID values for the instances of the Person block field and its MarStat field repeat in regular patterns (11 and 36 respectively), we can construct a query like the following that displays selected fields from this table for the current version of all cases.

```
SELECT
  BP.JOINKEY, INSTANCE, NAME, GENDER, AGE,
  CASE
    WHEN MARSTAT = 1 THEN 'Yes'
    WHEN MARSTAT = 2 THEN 'No'
    WHEN MSATT.STATUS = 4 THEN 'DK'
    WHEN MSATT.STATUS = 8 THEN 'REF'
    ELSE 'INAPP'
  END AS MARSTAT_TEXT
FROM
  dbo.BLAISE_FORM AS BF
  INNER JOIN dbo.COMMUTE7MOD_BPERSON AS BP
  ON BF.ENDSTAMP = '9999-12-31 00:00:00.000'
    AND BF.JOINKEY = BP.JOINKEY
    AND BF.DMKEY = BP.DMKEY
    AND BF.BEGINSTAMP = BP.BEGINSTAMP
  LEFT JOIN dbo.BLAISE_DATA AS MSATT
  ON BP.JOINKEY = MSATT.JOINKEY
    AND BP.DMKEY = MSATT.DMKEY
    AND BP.BEGINSTAMP = MSATT.BEGINSTAMP
    AND MSATT.FIELDID BETWEEN 10 AND 334
    AND (BP.BLOCKID * 36) + 2 = MSATT.FIELDID * 11
ORDER BY
  BP.JOINKEY, INSTANCE
```

Clearly this is cumbersome – one would not want to code this by hand very often, if ever. Computers, fortunately, are good at such menial tasks, so you could certainly automate the process of developing a set of

queries or database views to meet your needs. These would require joining a separate instance of BLAISE_DATA for each field that allows Don't Know or Refused values, so performance would have to be evaluated.

3.4.3 Linking Up Associated Records

When you define an array of blocks in your data model, the INSTANCE variable in the corresponding Instrument Table can be used to identify the array element for a given row in the table. Thus, in our Commute7Mod data model, where we have an array of persons within a household, INSTANCE is equivalent to a person-number field. When arrays are nested, however, INSTANCE only works at the innermost level. For example, if we can have multiple jobs for each person in a household, INSTANCE identifies the number of the job for the person in question, but there is no direct way of identifying which person holds the job. Here again we can use the information in the BLAISE_ID table (see Figure 12 below) to derive a higher-level identifier should we need to do so – for instance, to join person-level and job-level items.

	DMKEY	ID	TYPE	NAME
1	1	3	B	Person[1]
2	1	4	B	Person[1].Job[1]
3	1	6	B	Person[1].Job[2]
4	1	8	B	Person[1].Job[3]
5	1	10	B	Person[1].Job[4]
6	1	12	B	Person[1].Job[5]
7	1	14	B	Person[2]
8	1	15	B	Person[2].Job[1]
9	1	17	B	Person[2].Job[2]
10	1	19	B	Person[2].Job[3]

Figure 12. IDs for Person and Job blocks (partial)

```
SELECT
  BP.JOINKEY,
  BP.INSTANCE AS PersNum,
  BP.NAME,
  BJ.INSTANCE AS JobNum,
  BJ.EMPNAME
FROM
  dbo.BLAISE_FORM AS BF
  INNER JOIN dbo.COMMUTE7MOD_BPERSON AS BP
  ON BF.ENDSTAMP = '9999-12-31 00:00:00.000'
    AND BF.JOINKEY = BP.JOINKEY
    AND BF.DMKEY = BP.DMKEY
    AND BF.BEGINSTAMP = BP.BEGINSTAMP
  INNER JOIN dbo.COMMUTE7MOD_BJOB AS BJ
  ON BP.JOINKEY = BJ.JOINKEY
    AND BP.DMKEY = BJ.DMKEY
    AND BP.BEGINSTAMP = BJ.BEGINSTAMP
    AND BJ.BLOCKID BETWEEN BP.BLOCKID + 1 AND BP.BLOCKID + 9
ORDER BY
  BP.JOINKEY, BP.INSTANCE, BJ.INSTANCE
```

Here again, automated views can be generated to hide the complexities of the SQL.

3.4.4 Beware of the Secret Data

At one point when we first started working with Datalink, we used SQL rather than a Blaise module to change some data values. After doing so, we were greatly surprised to look at the data with Blaise and still see the old values. The Statistics Netherlands team informed us that (in the words of the current documentation), “Fast reading is enabled by default for BOI files which support record streams. When using a BOI file that supports record streams then the complete data of a Blaise record will be stored as a binary stream in the relational database. ... if fast reading is enabled, then Blaise will read the record data from the record streams, instead of the data that is stored according [to] the given data partition type. This means that whenever you manually change the data in the 'normal' data tables, that you won't see the changes in the Blaise client tools like the DEP and Dataviewer.”

The documentation indicates that you can avoid this problem by disabling Fast Reading. We decided that the better part of valor was to limit our non-Blaise access to readonly queries, leaving all data changes in the hands of Blaise.

3.5 Tools

While we were developing the latest version of the Blaise Editing System, we found that deploying a new instance of the system still required a number of manual processes. In order to expedite the system deployment process, we devised a Generation Tool to automate the most repetitive parts of the process. This tool, as depicted in Figure 13, allows us to generate:

- the database schemas,
- the triggers for each table,
- the .BOI files and the Blaise generic tables,
- meta tables, and
- database views.

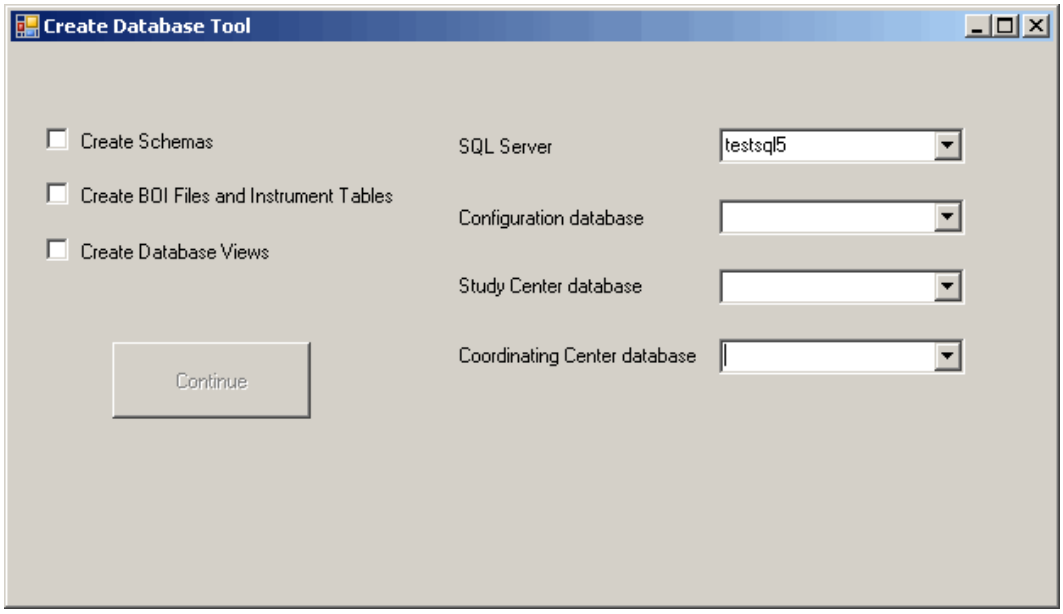
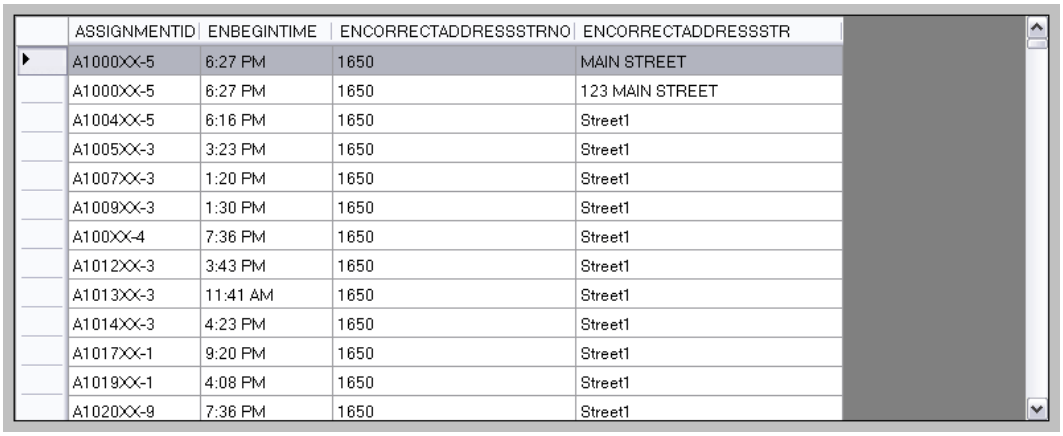


Figure 13. Generation Tool

The driver behind the Generation Tool is an .ini file. This file contains information about the SQL database, datamodels, schemas, the .boi template, and other study-specific items.

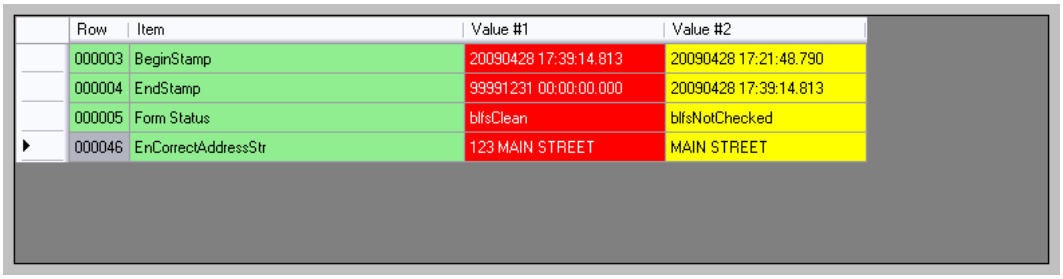
In addition to developing our own tools, we also found the new Blaise Data Center to be extremely useful for providing quick and easy *ad hoc* access to the data without having to write SQL queries or other programs (Figure 14).



	ASSIGNMENTID	ENBEGINTIME	ENCORRECTADDRESSSTRNO	ENCORRECTADDRESSSTR
▶	A1000XX-5	6:27 PM	1650	MAIN STREET
	A1000XX-5	6:27 PM	1650	123 MAIN STREET
	A1004XX-5	6:16 PM	1650	Street1
	A1005XX-3	3:23 PM	1650	Street1
	A1007XX-3	1:20 PM	1650	Street1
	A1009XX-3	1:30 PM	1650	Street1
	A100XX-4	7:36 PM	1650	Street1
	A1012XX-3	3:43 PM	1650	Street1
	A1013XX-3	11:41 AM	1650	Street1
	A1014XX-3	4:23 PM	1650	Street1
	A1017XX-1	9:20 PM	1650	Street1
	A1019XX-1	4:08 PM	1650	Street1
	A1020XX-9	7:36 PM	1650	Street1

Figure 14. Data Center Record View

One of the Data Center's most valuable features is its ability to export subsets of records and variables. We have also used it to verify the data changes made in the Blaise Editing System, as shown in Figure 15.



	Row	Item	Value #1	Value #2
	000003	BeginStamp	20090428 17:39:14.813	20090428 17:21:48.790
	000004	EndStamp	99991231 00:00:00.000	20090428 17:39:14.813
	000005	Form Status	blfsClean	blfsNotChecked
▶	000046	EnCorrectAddressStr	123 MAIN STREET	MAIN STREET

Figure 15. Data Center version differences

4. Conclusion

The latest version of Blaise Datalink, with its generic storage and versioning capabilities, provides a solid foundation for systems where tracking data changes is a requirement and you also need to query the data outside of a pure Blaise environment. The incorporation of Datalink into the latest version of our Blaise Editing System has resulted in a system that offers enhanced flexibility for our projects while also being simpler and easier to maintain.

5. References

Dulaney, Rick, and Boris Allan. "A Blaise Editing System at Westat." *Proceedings of the 7th International Blaise Users Conference.* September 2001.

< http://www.blaiseusers.org/2001/papers/Dulaney--Blaise_Editing_System_at_Westat.pdf > (April 26, 2009).

Gowen, Linda, and Pat Clark. "Lifecycle Processes to Insure Quality of Blaise Interview Data." *Proceedings of the 11th International Blaise Users Conference.* September 2007.

< http://www.blaiseusers.org/2007/papers/D4%20-%20Lifecycle%20Processes%20to%20Insure%20Quality%20of%20Data.pdf > (April 26, 2009).

Purohit, Yogeeta, and Latha Srinivasamohan. "Conversion of Blaise Databases to Relational Databases." *Proceedings of the 11th International Blaise Users Conference.* September 2007.

< http://www.blaiseusers.org/2007/papers/P1%20-%20Conversion%20of%20Blaise%20Databases%20To%20Relational%20Db.pdf > (April 26, 2009).

Rouschen, Arno. "Generic Data Storage with Blaise 4.8 Datalink." *Proceedings of the 11th International Blaise Users Conference.* September 2007.

< http://www.blaiseusers.org/2007/papers/A2%20-%20Generic%20Data%20Storage.pdf > (April 26, 2009).