Generic Data Storage with Blaise 4.8 Datalink

Arno Rouschen, Statistics Netherlands

1. Introduction

In Blaise 4.5 / BCP 1.0 it was possible to read / write data in external 'data stores', for example, relational databases and Excel spreadsheets. This is realized by using Microsoft's OLE DB technology. The concept of Blaise working together with OLE DB data sources is called the Blaise Datalink.

From Blaise 4.6 / BCP 2.0 on it became also possible to store Blaise questionnaire data in relational database management systems (RDBMS), like Oracle and Microsoft SQL Server. Within this and higher versions of Blaise you can use such a BOI file in the same way as a native Blaise database file. Blaise OLE DB wizards are available to create a BOI file and the corresponding database tables for any Blaise questionnaire. The resulting BOI file contains connection parameters and information about available tables and how to retrieve field data from these tables.

In Blaise 4.7 Enterprise Blaise Datalink has become a mature technology that can be used in both regular or internet survey scenarios. With Datalink it became easy to run an internet survey on a Blaise IS server and to store the collected data in a secure relational database server behind your firewall. BOI files can now be used to access the most popular databases, but also Blaise data files and even ASCII files.

Datalink has evolved further in Blaise 4.8. Although it is already a nice feature that you are able to store Blaise questionnaire data in an RDBMS, we had some ideas and also had some requests from clients to make the storage more generic; it would especially be nice if we could share database tables between questionnaires. In previous Blaise Datalink aware versions, BOI files were tailor-made for a particular survey and each Blaise questionnaire had its own set of database tables.

Blaise 4.8 Datalink has been extended and can define, next to the tailor-made BOI files, so called generic BOI files. Generic BOI files open the way to a centralized input data store; all survey data can be stored in a common set of tables in a relational database.

Another new feature of Blaise 4.8 Datalink is versioning. You can now have more versions of the same record in the database. This opens many opportunities, as you will see later in this paper.

2. Generic versus non-generic BOI files

If you create a BOI file for a Blaise dictionary then several table definitions will be created in order to store Blaise record information in a relational database. This information includes questionnaire data, form status information, answers to open questions, remarks and other data.

Until now these tables were strongly tied to the data model that was used with the BOI file; column sizes depended on the exact field sizes as defined in the data model, primary key fields defined in a Blaise data model were also used in the database tables and only required data columns were included in the database tables. The table names depended

on the name of the data model, i.e. BOI table definitions were non-generic and tailor-made.

These non-generic BOI files are still supported in Blaise 4.8 (in fact they are the default), but it is now also possible to define so-called generic BOI files. The primary goal of these generic BOI files is to share tables between data models as much as possible. Generic BOI files have generic table structures which can be used to store data of multiple data models.

2.1 Generic BOI files in Blaise 4.8

If you intend to store Blaise questionnaire data in a relational database and also want to store this data in a generic way, then it is still required to create a BOI file for each Blaise dictionary that you use.

Data partition types, which are introduced in Blaise 4.6 and BCP 2.0, can also be used with generic BOI files. A data partition type defines in which structure data must be stored. We have the following data partition types:

- Flat; no blocks each field in your data model has a corresponding column in a table.
- Flat; blocks same as above, but now there will be a separate table for each block definition.
- In depth data is stored in a field – status – value way. Each data type has its own column.
- In depth text same as in depth; except that field values are stored as a string
- Stream stores the data of a record as a stream per block

It is perfectly okay to create a generic BOI file which uses the in depth data partition type for data model A and another generic BOI file which uses a flat storage structure for data model B. The added value of using generic BOI files is that most tables can be shared between the data models

Here is an illustration how table access works in both non-generic and generic scenarios. In case of generic BOI files database tables will be shared if possible, while in the non-generic situation each BOI file has its own set of tables.

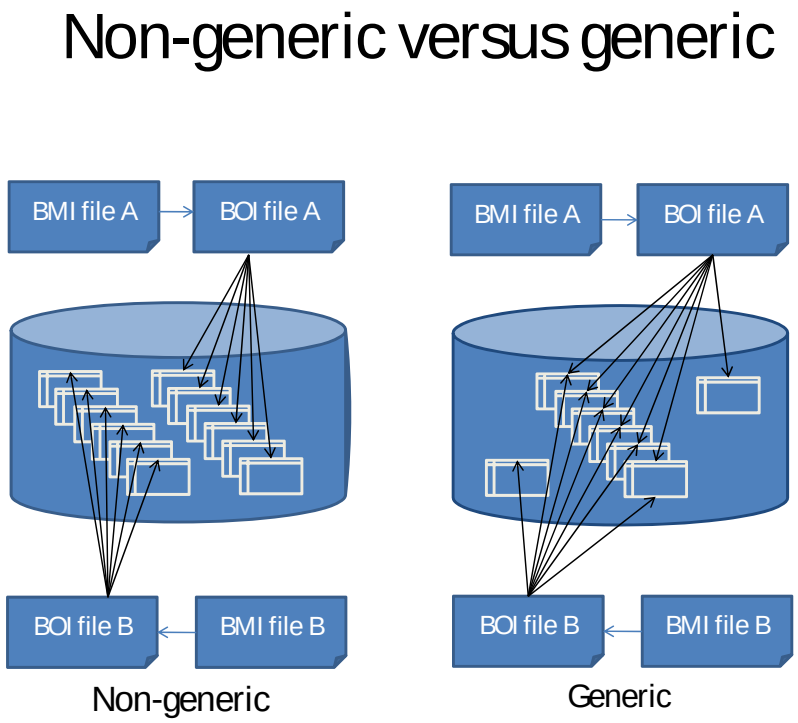

Figure 2.1 Table access with both non-generic and generic BOI files

2.1 Generic BOI files purposes

The main goal of generic BOI files is to share database tables as much as possible between Blaise data models. Generic BOI files can be used to setup a centralized data store. This is possible because these BOI files define tables which can be used with any Blaise data model and as a result have the possibility to store data from multiple surveys.

In such a scenario Blaise questionnaire data is stored in a centralized way in only a few predefined and fixed table structures. Because the number of needed tables decreases, the maintenance burden for your system and database administrators decreases also. The less tables the better is the motto.

Another advantage of using generic BOI files is that it becomes easier to embed Blaise cases and data within your own systems, because the generic BOI tables have a fixed predefined structure, which makes it possible that each survey can be linked in the same way. The concept of generic BOI files will be explained in more detail below.

2.2 Generic and non-generic tables

If we take a closer look at the tables which are created in generic BOI files, then we can also distinguish between non-generic and generic tables. Generic tables have always the same structure no matter which data model is being used. Because they have always the same structure, they are perfect candidates to be shareable.

Here is a listing of the generic tables, which are currently used by Blaise Datalink:

- Dictionary information table; stores the dictionaries which are part of the input data store
- Case information table; stores the cases of a survey
- Form information table; contains among others the form status information
- Block information table; contains the check status information of each block
- In Depth data tables; stores data in depth in a field – value way
- Remark table; contains the remarks with fields
- Open table. Table to store answers to open questions

There are just two types of tables who aren't generic. An example of such a table is a flat data table; in a flat table each column is mapped to a particular Blaise field of your data model. Needless to say that these tables only can be used with the data model on which they are based.

Example flat data table:

```
DATAMODEL test

FIELDS
        A: integer
        B: integer

ENDMODEL
```

If you create a flat data table for this data model then the data table will contain a column for field A and another column for field B.

The other non-generic table type is the Key information table. This is an auxiliary table which Blaise Datalink uses for retrieving records according to a Blaise key. Key information tables will be created in case of in depth and stream data partition types, because these partition types don't provide fast access to individual Blaise key field values. In these cases Blaise key fields are stored redundant; in depth or in a stream depending on your data partition type and also in a flat structure in the Key Information table.

I want to emphasize that despite the fact that flat data tables and key info tables cannot be shared between data models the other tables still can. If you open a generic BOI file in OLE DB Workshop you can see which tables are shareable.

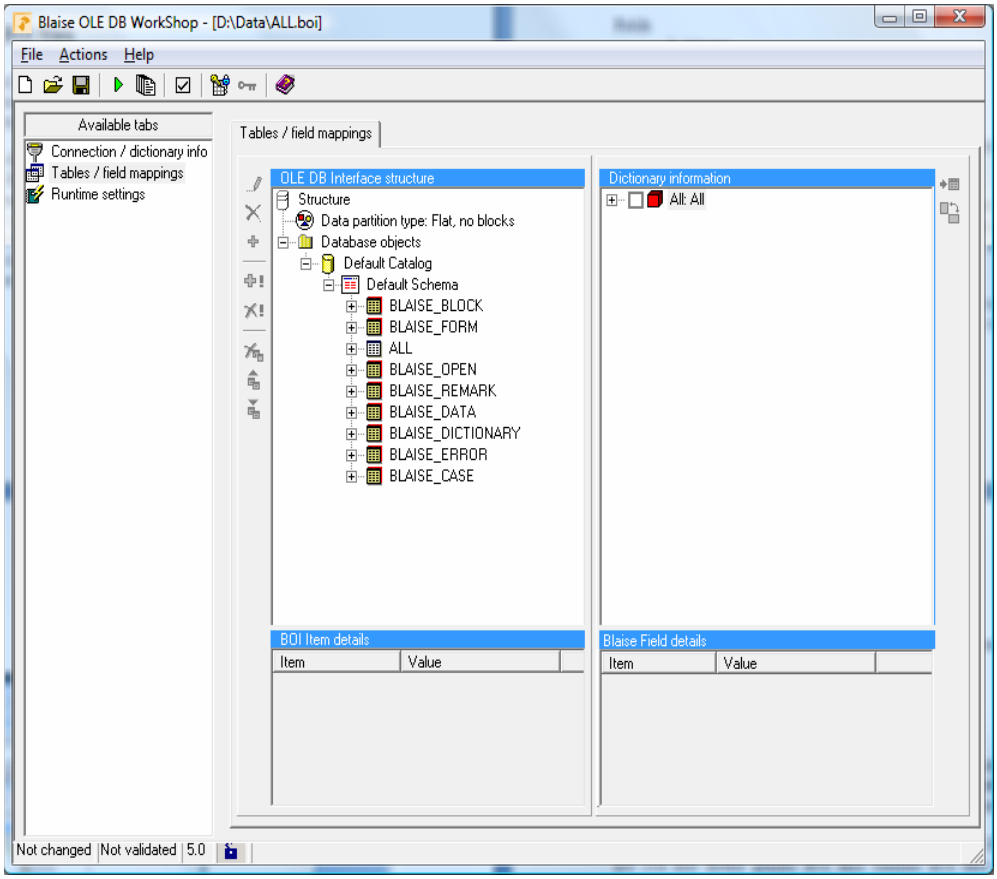

Figure 2.2 BOI Table definitions in OLE DB Workshop

Shareable tables have another (yellow) icon. By default their names begin with "BLAISE". In this example there is only one non-generic table, table ALL. This is because this is a flat table with field mappings, i.e. each column in the table maps to a Blaise field of the associated data model.

See the appendix for more information about the available table types and their predefined structures.

3. Generic BOI files concepts

Generic BOI files use the following three important concepts, which allow us to store data from multiple data models in the same tables:

3.1 Common primary key

The most important difference between non-generic and generic BOI files is the primary key which they are using for their database tables. Non-generic BOI files use the primary key fields of the data model as primary key columns in the database tables. Generic BOI files on the other hand use a fixed and predefined primary key which is common to all generic BOI files.

This common primary key has the following columns:

JOINKEY	Contains a unique integer which identifies a case.
DMKEY	Contains the data model key
BEGINSTAMP	Contains the begin time of the period of validity of a particular record. This value has to be unique in combination with JOINKEY and DMKEY. This column is used for versioning.

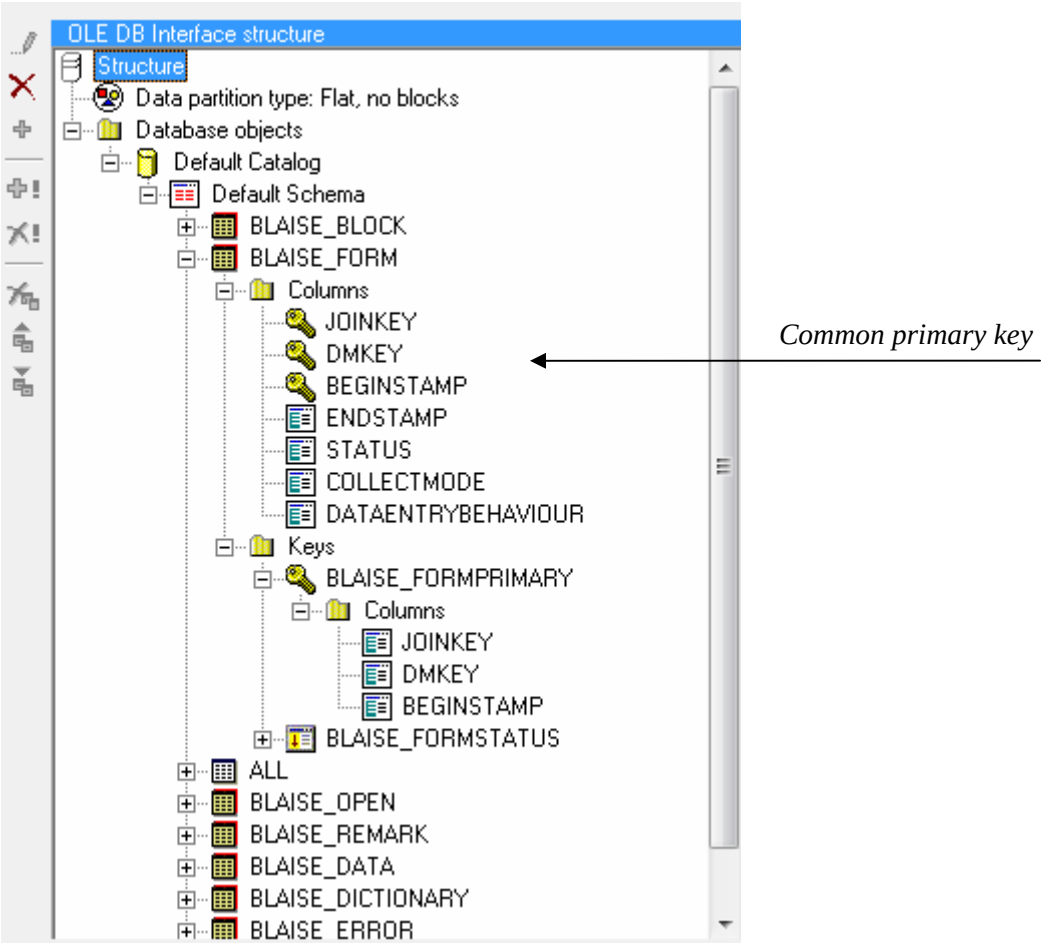

Figure 3.1 Structure of the generic Form information table

JOINKEY serves as a unique identifier for a case within Blaise. JOINKEYS will be automatically generated by the target database if a record does not exist already.

*DMKEY i*s the key of the data model which belongs to the record data. Data stored with Blaise Datalink has always a link to its meta, because Blaise wants to know what the data represents. So we have no data without meta data!

Here is some more information about the data model key and why the key is important. Whenever Datalink opens a connection to the target database, a check occurs whether the data model that is being used is already registered in the Data Warehouse.

If the data model is not registered yet, then Blaise Datalink generates a DMKEY for the data model and also adds a new row with this new key to the *Dictionary Information* table. This is a fixed and predefined table which is included in all generic BOI files. The table stores all known data models along with some information, among others the paths of the BMI and BOI file which must be used.

The *Dictonary Information* table stores also the check sum of the data model. You can look at this check sum as a unique identifier for a version of a data model. If you change something in your data model, then the check sum will differ from the previous version of your data model. Blaise Datalink uses this check sum to distinguish between versions of data models. If a check sum differs from the check sum from an already stored data model, then a new DMKEY will be generated and as a result data will be stored along with the new meta information.

BEGINSTAMP is the begin time of the period of validity of a particular record. Whenever a record is stored for the first time then *BEGINSTAMP* will be filled with the time stamp of that moment. Begin stamps are generated automatically in the database. The main goal of the BEGINSTAMP column is that it can be used to distinguish between versions of records. If versioning is enabled then a new version of the record will be saved with a new begin stamp. You can read more about versioning in the next chapter.

Although generic BOI files use this common primary key in order to store data in a generic way, Blaise end users are not aware of this; it works completely transparent; users just need to define, as usual, primary and secondary keys in their Blaise data models.

3.2 Fixed table structures

Generic BOI files use also fixed and predefined table structures, while non-generic BOI files use tailor made table definitions. In the latter case only columns which are actually needed will be part of the table definitions. For example; if you have a data model with only integer fields and you choose the in depth data partition type then the resulting in depth data table will contain only a data column to store the integers. If you choose to create a generic BOI file then the table will have columns to store the other data types too. This is because other (future) data models might need these columns. By defining columns for each data type, regardless whether they are needed, the table can be shared between data models.

3.3 Use of maximal allowed column widths

Generic BOI files use maximum column widths supported by the particular database. Non-generic BOI files determine the column width which is needed to store the information. For example; if you have a data model with string fields and you choose the in depth data partition type then the resulting in depth data table will define a STRINGDATA column which has a width which corresponds to the largest string field size in your data model. If you choose to create a generic BOI file however, then the STRINGDATA column width will be set to the maximum column width supported by the database, again because future data models might require such a width.

4. Versioning

A new and interesting feature of Blaise 4.8 Datalink which we haven't told about yet, is versioning. Versioning can be used to store multiple versions of a Blaise record in the database. These versions can be retrieved on demand, for example with help from our new tool: Blaise OLE DB Data Center.

If you are using versioning then each record has a begin stamp which discriminates between the versions. Versioning can be switched on and off by altering the BOI runtime settings in OLE DB Workshop. Versioning can only be used with generic BOI files.

4.1 Versioning concepts

Our implementation of versioning works with so called time stamps. We have introduced two columns in order to enable versioning, namely:

BEGINSTAMP	This column contains the begin time of the period of validity of a particular record. This value has to be unique in combination with JOINKEY and DMKEY.
ENDSTAMP	This column contains the end time of the period of validity of a particular record.

These columns are defined as date time columns in the target database. Their precision is seconds for most databases. *BEGINSTAMP* is also part of the common primary key and stores the begin time of the period of validity of a record.

ENDSTAMP contains the end time of the period of validity of a record and has to be always greater than the *BEGINSTAMP* of that record. If *ENDSTAMP* is empty then the record is the actual record.

In its simplest form versioning works as follows. Whenever a record is written to the database then Blaise Datalink checks whether there is already an older version of the record. If this is the case then this older record will be closed by updating its *ENDSTAMP* column. Blaise Datalink will also insert a new row for the new version of the record. This row will have a new *BEGINSTAMP*, which is filled with the execution date and time. The *ENDSTAMP* column of this newly inserted row will be left empty.

When we take a closer look to versioning enabled records, we can differentiate between *actual* and *historical* records. *Actua*l records are the records which contain the current values. Their *ENDSTAMP* column is empty. If a new version of a record becomes available then the *ENDSTAMP* column will be filled with the current timestamp. The

record becomes historic, because we have a newer one. *ENDSTAMP* contains the end time of the period of validity of the record.

4.2 Transparency

Versioning works completely transparent to the user, i.e. the user doesn't see whether versioning is enabled. He/she just uses the DEP, Manipula and Dataviewer as usual by using the keys defined in the Blaise data model. Despite the fact that there could be more versions of a record in the target database, only *actual* records will be used by the Blaise client tools.

4.3 Versioning example

Let's say we have a Blaise record and we haven't any versions of it yet in the target database. If we write this record to the database (we save the record in the DEP for example) then our database might look as follows:

	JOINKEY	DMKEY	BEGINSTAMP	ENDSTAMP	STAT...	COLLECTMODE	DATAENTRYBEHAVIOUR
1	997	1	2007-06-18 08:49:54.873	NULL	8	-1	0

We just have taken one table (Form information table) for this example, but all BOI involved tables will have the same primary key after the insertion of our record. The following important things happened behind the scene during the insert:

1. A *JOINKEY* has been generated for the case record. This is because we didn't have a record for this case yet. The *JOINKEY* serves as an identifier for a case in Blaise. *JOINKEYS* are generated automatically by the target database
2. You also see a *DMKEY* column. In this column you find the key of the data model that is being used. Data model keys will be generated at the moment that you open a Blaise Datalink for a Blaise data model for the first time.
3. *BEGINSTAMP* has also been generated by the database and contains the begin period of validity for the record.
4. *ENDSTAMP* is empty at the moment, because it is the only and thus the actual record. The other columns are not important for now.

Now let's add a new version of the record. This might be done as follows. Get the current record in the DEP, alter a value of a field and save the record. Here is our Form information table again:

	JOINKEY	DMKEY	BEGINSTAMP	ENDSTAMP	STAT...	COLLECTMODE	DATAENTRYBEHAVIOUR
1	997	1	2007-06-18 08:49:54.873	2007-07-17 12:27:39.520	8	-1	0
2	997	1	2007-07-17 12:27:39.520	NULL	1	-1	3

There are now two rows in the table; one row for the 'old' record and one for the new version. In the 'old' row *ENDSTAMP* has been filled with the date and time stamp of the executed write action. The same timestamp has been used to fill the *BEGINSTAMP* column of the new record. Please notice that not only the Form information table has been updated, but that all involved BOI tables have been updated according to this mechanism.

You can also see that the form status changed from 8 (= Unprocessed) to 1 (=Clean) and data entry behaviour changed from 0(=Unchecked editing) to 3(=Strict Interviewing). This is because the record was loaded and checked by the data entry program.

4.4 Versioning and write actions

As said, versioning can be switched on and off by altering the corresponding BOI runtime setting. This can be done in the Runtime settings tab within OLE DB Workshop.

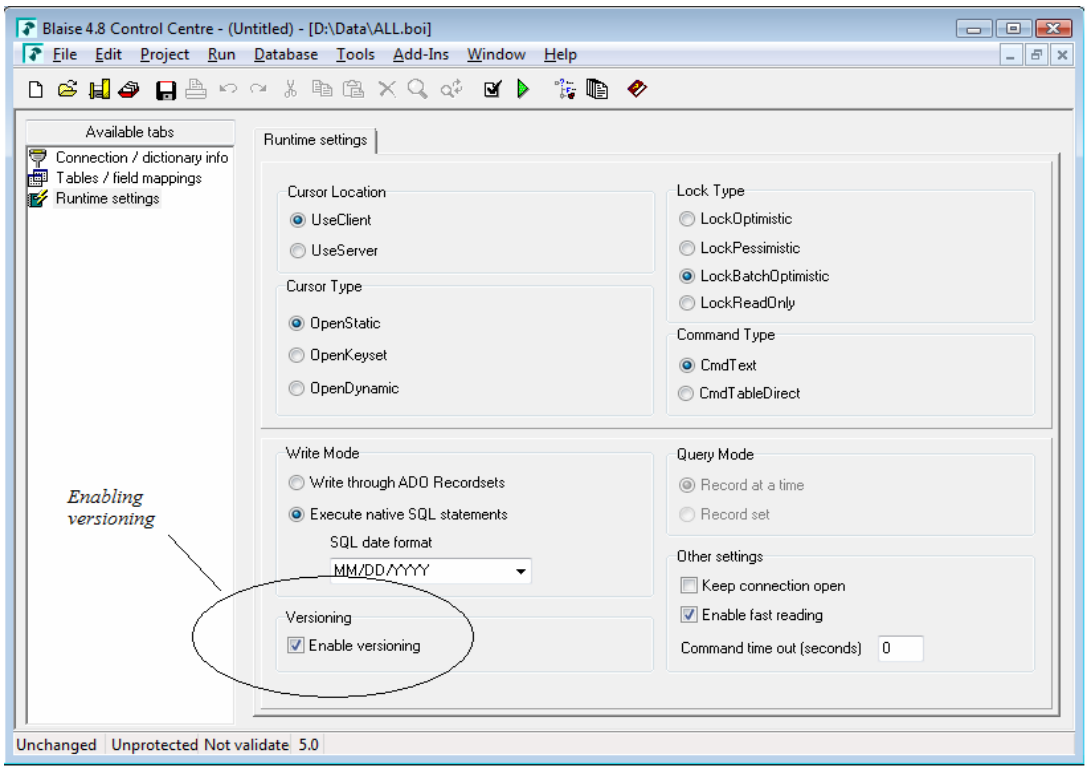

Figure 4.4.1 Versioning can be enabled by checking 'Enable versioning'.

Blaise supports several write actions and each of these actions has a different execution strategy. Here is a listing of these actions and the behavior in case we haven't enabled versioning:

- *WriteRecord*: writes a Blaise record to the database. If your data model has a primary key and a record with this primary key doesn't exist yet, then a new record will be created, otherwise the existing record will be overwritten with the new values. If you don't have a primary key then a new record will be added.
- *CreateRecord*: creates a new record in the database. If your data model has a primary key and a record with this primary key doesn't exist yet, then a new record will be created, otherwise an error will be raised, because you want to create a record which already exists. If you don't have a primary key then a new record will be added.
- *UpdateRecord*: updates an existing record in the database. If your data model has a primary key and a record with this primary key doesn't exist yet, then an error

will be raised, because you want to update a record which doesn't exists yet. If you don't have a primary key then an error will be raised.

As you see the working of these write actions is really straightforward. Now we have introduced versioning however, the situation becomes a little more complex. Here is a table which shows you the results of a write action in a particular situation:

Action / Setting / History	Versioning disabled		Versioning enabled	
	No record history present	Record history is present	No record history present	Record history is present
WriteRecord	Create new record BEGINSTAMP = current time and ENDSTAMP = empty	If there is an actual record (ENDSTAMP is empty), then update this record, otherwise create new record with BEGINSTAMP = current time and ENDSTAMP = empty	Create new record BEGINSTAMP = current time and ENDSTAMP = empty	If there is an actual record (ENDSTAMP is empty), then update ENDSTAMP of this record with current time. Also create a new record with BEGINSTAMP = current time and ENDSTAMP = empty
UpdateRecord	Error	If there is an actual record (ENDSTAMP is empty), then update this record, otherwise error	Error	If there is an actual record (ENDSTAMP is empty), then update ENDSTAMP of this record with current time. Also create a new record with BEGINSTAMP = current time and ENDSTAMP = empty
CreateRecord	Create new record BEGINSTAMP = current time and ENDSTAMP = empty	If there is an actual record (ENDSTAMP is empty), then error, otherwise create new record with BEGINSTAMP = current time and ENDSTAMP = empty	Create new record BEGINSTAMP = current time and ENDSTAMP = empty	If there is an actual record (ENDSTAMP is empty), then error, otherwise create new record with BEGINSTAMP = current time and ENDSTAMP = empty
DeleteRecord	Do nothing	If there is an actual record (ENDSTAMP is empty), then update ENDSTAMP of this record with current time.	Do nothing	If there is an actual record (ENDSTAMP is empty), then update ENDSTAMP of this record with current time..

Figure 4.4.2 Write actions and their result in a particular situation.

In general versioning works as follows:

The current actual record will become a historical record if a newer version of that record becomes available. In that case the old version gets an ENDSTAMP. The new version will be inserted with a new BEGINSTAMP and becomes the actual record. Its ENDSTAMP column will be left empty.

4.3 Versioning opportunities

If you enable versioning you will be able to see and analyze the history of Blaise records. With our new tool Blaise OLE DB Data Center you can analyze the difference between versions of records and see how and when changes to records have been applied.

This is nice, but working with versioning and especially with BEGIN- and ENDSTAMPS opens also opportunities which you might not have realized yet. One of the most important features is that we now are able to look at our survey data at a specific moment and in a particular time frame, i.e. you could ask the Blaise system to retrieve the situation of a survey at moment X and compare this to the situation at moment Y, i.e. we have the possibility now to analyze the survey progress and data quality.

4.4 Versioning requisites

Versioning can only be enabled with generic BOI files. You will also have to use a target database for which we have enabled versioning. This is because the target database must have enough support for date time columns in the way Blaise Datalink uses them. The database, for instance, must have the capability to generate JOINKEYS and BEGINSTAMPS automatically. Not all databases have support for those features. At the moment of writing we have enabled versioning for Oracle, Microsoft SQL Server and MS Access databases. We expect however to add more databases to that list in the near future.

5. Generic BOI files in practice

If you want to store the data of your survey in a relational database, you can use Blaise OLE DB Workshop to create a BOI file for your data model. Please read the Datalink chapter in the Blaise Online Assistant for a thorough explanation about creating BOI files and the available create options. In short you will have to do the following: First enter a connection string and Blaise dictionary in the Connection tab. Then go to the Tables / Field mappings tab and double-click the ‘Create table definitions for dictionary’ option. Now a ‘Create table structure for dictionary’ dialog pops up. If you want to create a generic BOI file then you must use the ‘Advanced’ tab.

In the Advanced tab you have two options:

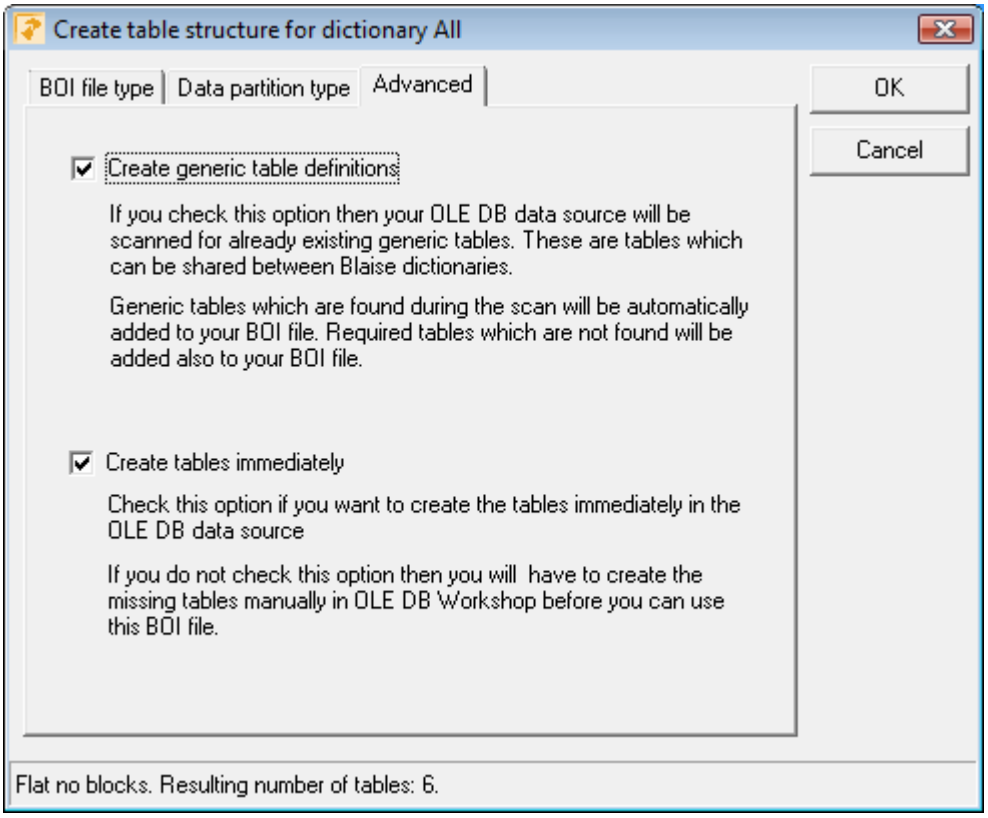

Figure 5.1 Advanced settings for generic BOI files

Check 'Create generic table definitions' if you want to create a generic BOI file for your data model. If you check the option then your OLE DB data source will be scanned for already existing generic tables. Found which are found are automatically added to your BOI file. Required tables which are not found will be added also. Note that the generic option is only enabled if your database supports versioning. See the specified requisites elsewhere in this paper.

After the generic BOI file has been created, OLE DB Workshop shows the table definitions that have been created.

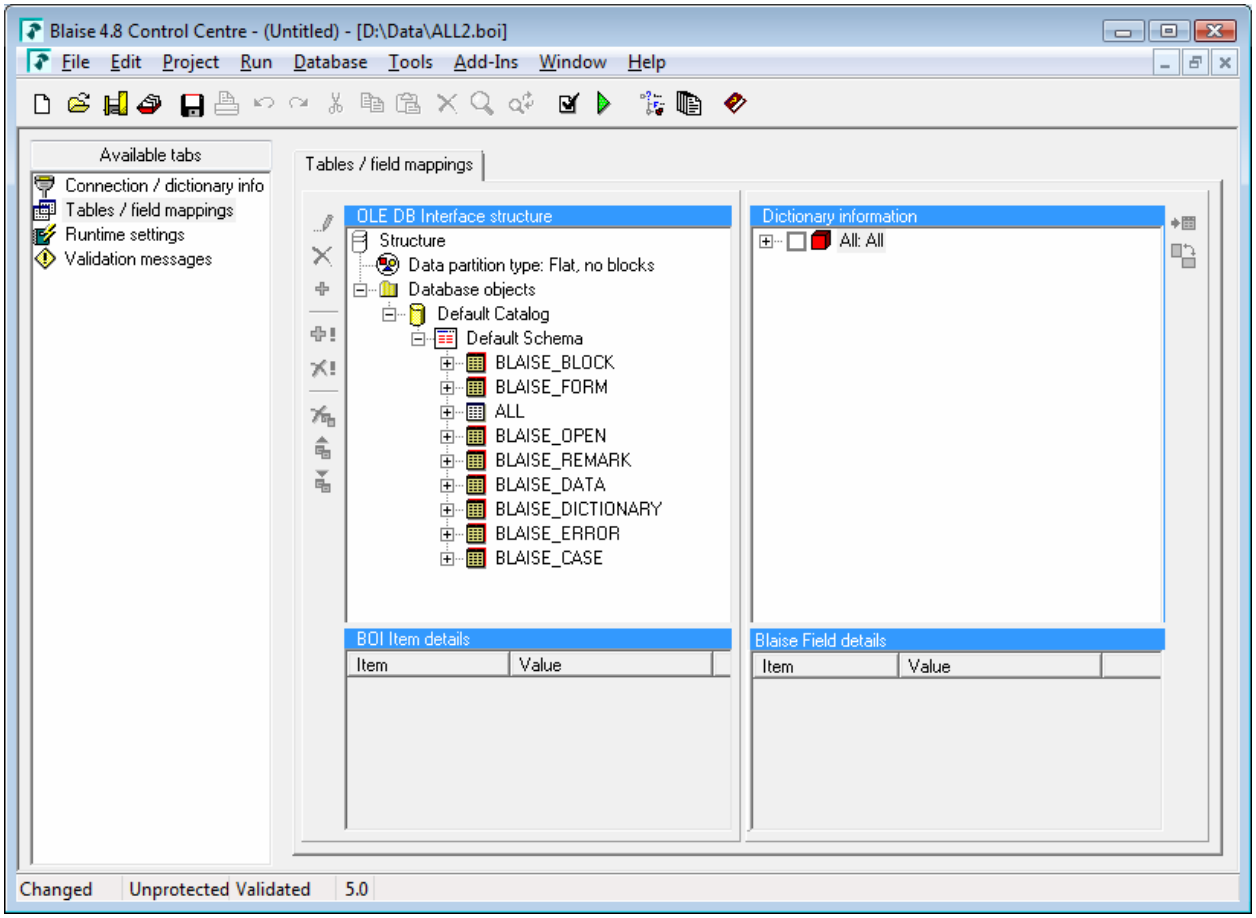

Figure 5.2.1 Generic table definitions displayed in OLE DB Workshop

We have a BOI file now with table definitions which can be used to store the data of the specified data model. This process has created a BOI file with logical table definitions; the tables themselves have not been created yet. The BOI file is a kind of blue print of what should be present in the database.

Be careful whenever you drop and create tables in the database!

Because we are working with a generic BOI files, it could be possible that some tables or even all the tables already exist in the database, because they were created by another generic BOI file earlier. This is because generic BOI files use the same table structures and, as a default, the same names in each generic BOI file. So you have to be careful with dropping and creating tables, because tables could exist already and contain data of other questionnaires!

To check whether a BOI file is ready to be used, you can validate the BOI file by choosing 'Project – Validate' from the menu. If everything is okay, then you will get the message 'OLE DB content is valid' otherwise errors / inconsistencies are listed in the 'Validation messages' tab. Missing tables will be listed there.

After validation you can see which tables are missing. You can create missing tables by selecting them in OLE DB Workshop and right clicking them. Choose 'Create table in OLE DB data source. Do this for all the tables which have not been created yet.

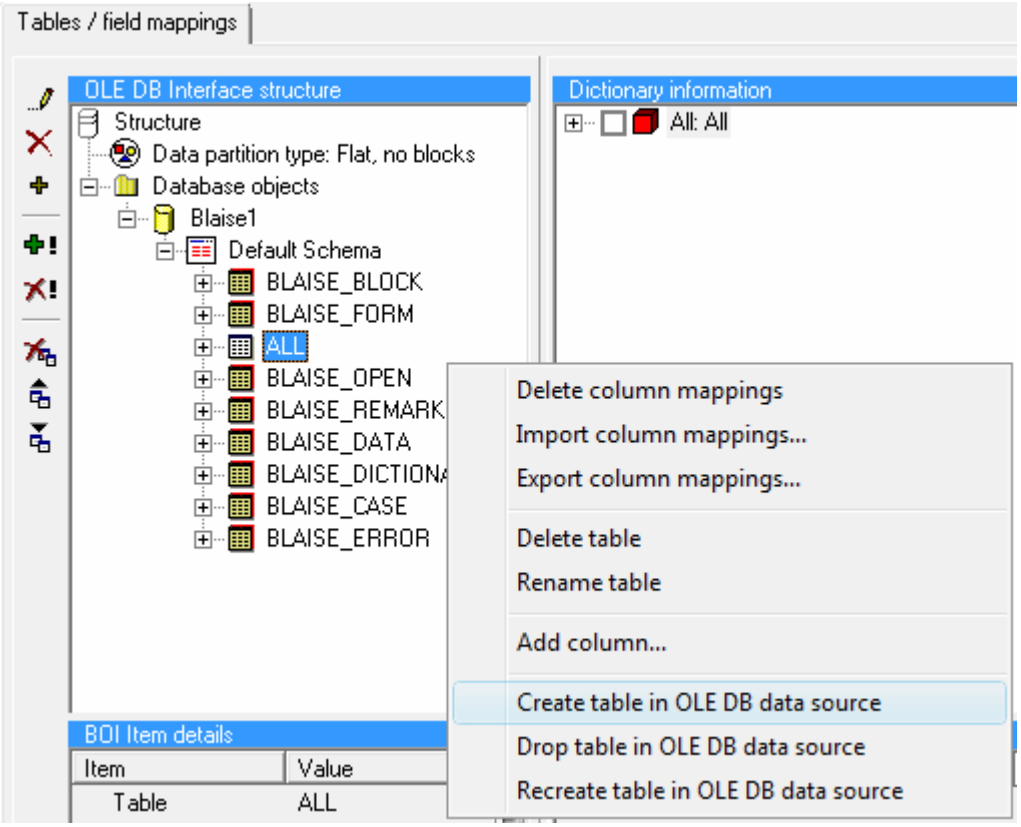

Figure 5.2.2 Create missing tables in the OLE DB data source

6. Blaise OLE DB Data Center

Generic BOI files offer many opportunities. If you store questionnaire data that belongs to different surveys in the same set of tables, you can speak of an input data store. Because questionnaire data is stored in a centralized way and as a result the available data can be overwhelming, we must have a way to look at and extract questionnaire data in an easy way. Also we might be interested in the progress of a survey. Blaise 4.8 Datalink supports versioning, so we must also be able to look at the history of records.

That is what the new Blaise OLE DB Data Center is all about. Its main purpose is to let Blaise users view and extract data in a uniform and easy way, without bothering about underlying table structures. The idea is that the view and extraction mechanism works exactly the same for all BOI files, so you have to learn it only once and then you can apply it to all BOI data.

I will explain the main features of this new tool below. Please notice that at the moment of writing Blaise OLE DB Data Center was still under development and that features might not be available in the final version and that screenshots may differ from the final version.

6.1 Data connections

Blaise OLE DB Data Center uses data connections in order to access the questionnaire data in a relational database. There are two types of data connections that can be made. The first type is an OLE DB connection string to a target database. The second type is a BOI file.

The difference between these two connection types is that in case of an OLE DB connection string the target databases will be scanned for registered dictionaries and accessible tables, while in case of a BOI file only BOI tables and its associated Blaise dictionary will be loaded into OLE DB Data Center.

Data connections are displayed in OLE DB Explorer which is the tree view on the left part of the screen.

OLE DB Explorer is the entry point of the application; you have to specify a data connection before you can do anything with OLE DB Data Center

In this example you see an instance of OLE DB Explorer with two connections: one to a BOI file and one with a connection string to a SQL server database.

Each connection shows the available Blaise dictionaries and accessible database tables.

Data connections are saved to file between sessions with OLE DB Data Center, i.e. you don't need to specify these connections again if you start up the application.

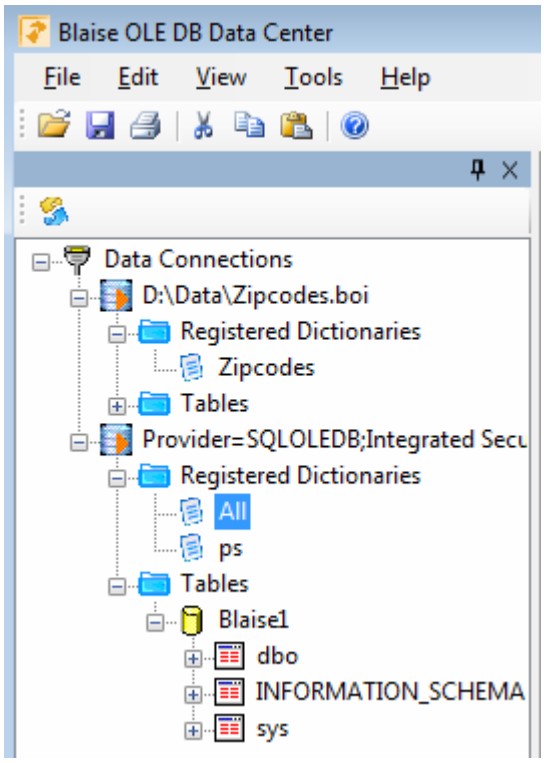

Figure 6.1 OLE DB Explorer in action.

6.2 Combining Meta and Data

If you are using Blaise Datalink then there must always be meta information available about the data that has been stored in the relational database. Datalink must always be able to access the Blaise dictionary that is associated with a particular BOI file. The guaranteed availability of meta is a big plus. Ask yourself: how often does it happen that data has been stored somewhere and no one knows what the data represent? Blaise Datalink simply requires meta in order to get the right data for each field in your Blaise data model out of the database.

How does Blaise Datalink knows which meta must be used?

The answer is simple: a Blaise OLE DB Interface (BOI) file contains a path and search path to the Blaise dictionary that must be used along with the data.

If you are using generic BOI files then the associated dictionaries are not only stored in the BOI files themselves, but also registered in a Dictionary table in the target database. This opens the possibility to provide a catalog function of registered dictionaries and BOI files which are used in an input data store.

Each dictionary has a data model key, which is also part of the common primary key of each database table. In this way Blaise Datalink is able to distinguish which data belongs to which data model. You can see this catalog function in OLE DB Explorer. The listed dictionaries in the Registered Dictionaries node have been retrieved from the Dictionary information table which is located in the relational database.

6.3 Viewing and extracting data

Whenever you open a data connection in OLE DB Data Center the Explorer searches for dictionaries which can be used.

Found dictionaries are listed under the Registered Dictionaries node, which becomes available after a data connection has been opened. If you select a dictionary then you can pop up a dictionary menu by clicking the right mouse button:

If you click View/Extract Data menu entry then a window is opened that can be used to view and extract questionnaire data.

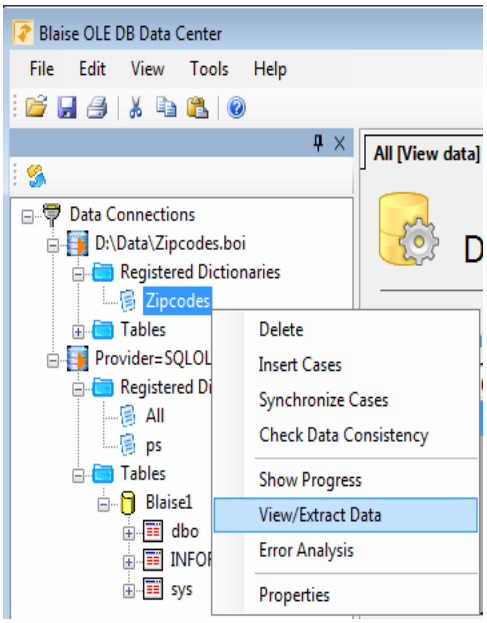

Figure 6.3.1 Dictionary menu

A nice thing about viewing and extracting data with OLE DB Data Center is that you only have to learn once how to specify what you want to see and extract and that it works for all questionnaires in the same way.

After you selected ‘View/Extract Data’ from the submenu, the screen looks like this:

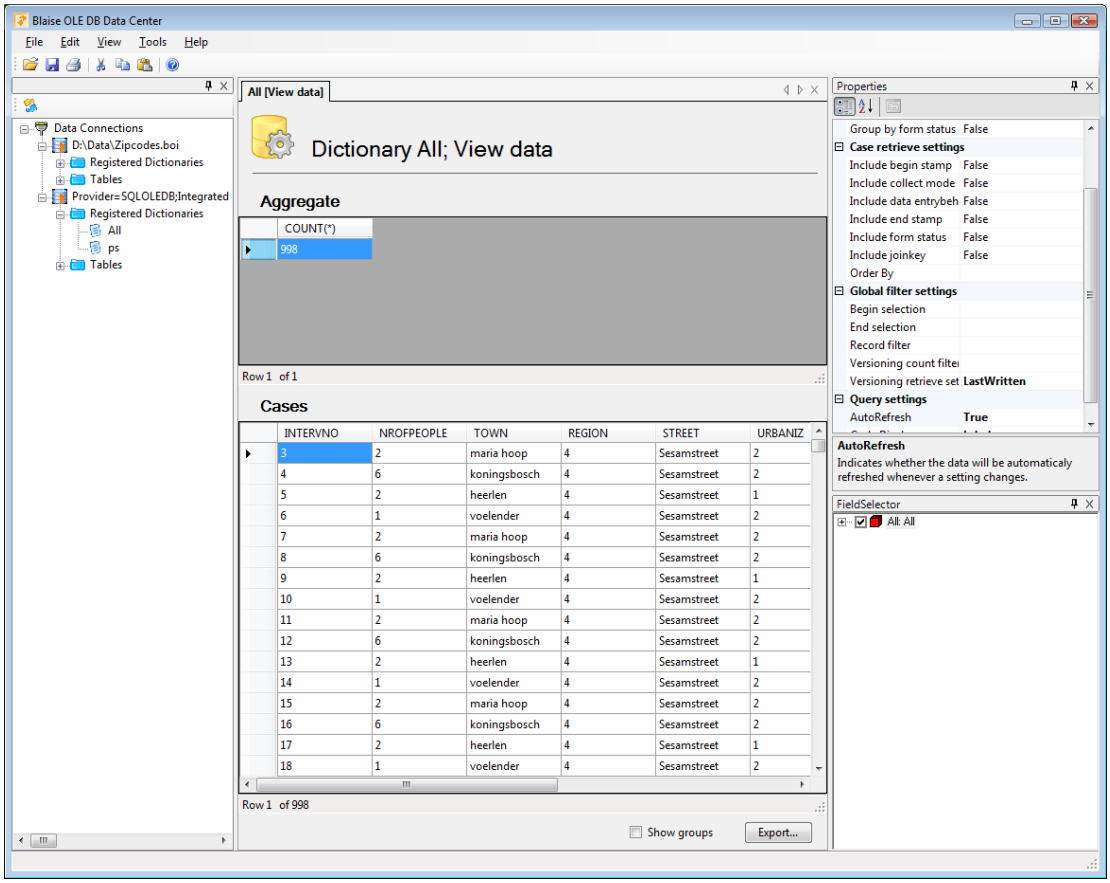

Figure 6.3.2 View/Extract Data: initial screen

The screen is now divided into four parts. On the left you see OLE DB Explorer. From here you can initiate other actions at any moment. In the middle we have a ‘View data’ document in which data is displayed and on the right you can find Properties and a Field Selector. You can use the latter two to specify which data you want to retrieve from the target database.

The ‘View data’ document, which is the active document right now, is divided into an *Aggregate* part and a *Cases* part. These two parts work together. *Aggregate* contains, as the name already indicates, aggregated data. The *Cases* part contains Blaise cases which are involved in the selected row in the *Aggregate* section.

In this screenshot the count (number of records) of data model All is displayed in the upper section, while the cases sections contains the cases which are involved in this count.

How do the available screen parts interact with each other?

You can use the Properties window on the right to specify retrieve settings. This Properties window contains retrieve settings in case you are viewing and extracting data. Examples of retrieve settings are field and record filters, but also group by settings.

The available retrieve settings can be divided into four parts.

Aggregate settings are settings that work on the aggregate part of the screen. Here you can specify how data will be grouped. Data can be grouped by form status, data entry behavior, data collect mode and by field data.

Case settings can be used to influence which case information will be retrieved in the bottom half of the data document. You can include join keys, begin- and end stamps, form status, et cetera. These settings work in conjunction with the field selector, which you can use to specify which field data should be retrieved.

Global filter settings work on both aggregates and cases. These settings can be used specify a begin and end time and a record filter. There are also some settings which are related to versioning.

Finally there are also some *general* settings.

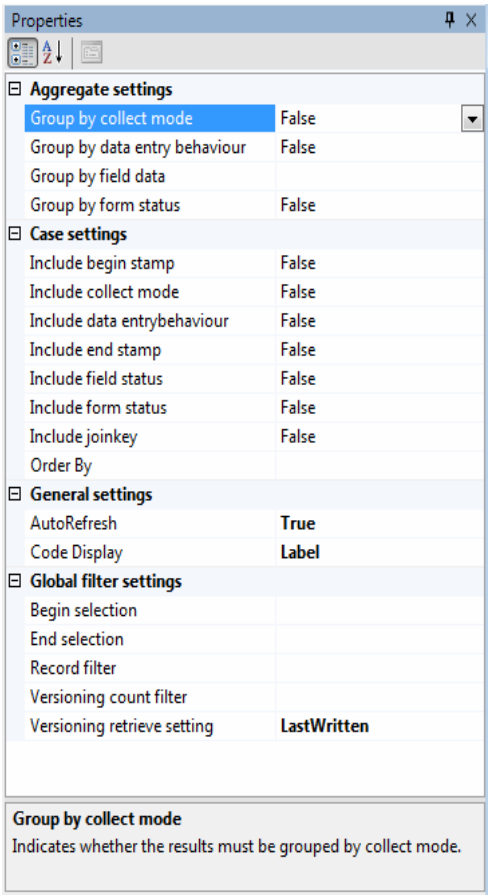

Figure 6.3.3 Available settings

By using retrieve settings in combination with field selector you can retrieve data from the database. Once the information is displayed in the *Cases* part of the screen, you can export the data to other applications and file formats. Blaise OLE DB Data Center supports several output formats. You can, for instance, copy the data and paste it in Microsoft Excel or SPSS or export the data to a text file.

These export facilities are still under development at the moment of writing, so I can't go into much detail about it yet, but it should be more concrete during the time that this paper is presented.

6.4 Administrative tasks

Blaise OLE DB Data Center has two execution modes: administrator and user. If you are logged on as an administrator then administrative tasks will be available next to the default features. The extra mode has been introduced because some people must be able to administrate the surveys in the IDW.

Tasks you can think of are the following

- Adding and deleting surveys
- Inserting and deleting cases
- Importing data
- Synchronizing cases
- Performing data validation checks

At the moment of writing this tool is still under development, so features might not be available in the final version.

7. Conclusion

Blaise 4.8 Datalink has a number of interesting new features. You can use generic BOI files to setup an Input Data Warehouse in which the data of multiple Blaise surveys is stored. Blaise end users can use Blaise OLE DB Data Center to view and extract the data they want in an easy and uniform way.

We also have seen another new feature of Blaise 4.8 Datalink which is versioning. Versioning makes it possible to store the history of a Blaise record. This gives us the possibility to view the changes made to a record in time but also to look at a survey within a particular time frame. It is possible to analyze the progress of the survey and also the progress in data quality.

8. Appendix Predefined generic table structures

Generic BOI files use predefined fixed table structures as said earlier. In this section you find an overview of these tables and their structure. These table structures are defined in any generic BOI file.

8.1 Dictionary information table

The Dictionary Information table stores information about the dictionaries which are used in the IDW.

The default name of this table is BLAISE_DICTIONARY. Its structure looks like this:

DMKEY	Data model key
DATAMODELNAME	Name of the data model
CHECKSUM	Check sum of the data model
BMI	Path to the associated dictionary file
BOI	Path to the associated BOI file
SEARCHPATH	Dictionary search path
ADDED	Date and time of adding
COLLECTMODES	Supported collect modes for this dictionary. Not implemented yet

8.2 Case information table

The Case Information table is used to store the JOINKEYs of a survey. If a new, non existing case is written to the database then a new JOINKEY is generated and stored into this table.

The default name of this table is BLAISE_CASE. Its structure looks like this:

JOINKEY	Unique integer which identifies a case.
DMKEY	Data model key
COLLECTMODE	Default collect mode for this case. Collect mode has not been implemented yet.

This table can possible be extended by a user defined data model. We were exploring this possibility at the moment of writing this paper.

8.3 Form information table

The Form Information table is used to store the status information about a form. The table contains also information about the collection mode of the form and the data entry behavior which was being used at the moment the record was written.

The default name of this table is BLAISE_FORM. Its structure looks like this:

JOINKEY	Unique integer which identifies a case.
DMKEY	Data model key
BEGINSTAMP	Begin time of the period of validity of a particular record. This value has to be unique in combination with JOINKEY and DMKEY. This column is used for versioning.
ENDSTAMP	End time of the period of validity of a particular record. If this column has an empty value than it is the actual record.

STATUS	Form status of the record.
COLLECTMODE	Collect mode of the record. At the moment of writing this was not implemented
DATAENTRYBEHAVIOUR	Data entry behavior of the record.

8.4 Block information table

The Block Information table is used to store the block check status and error information. This is information that Blaise needs to set suppressed errors et cetera.

The default name of this table is BLAISE_BLOCK. Its structure looks like this:

JOINKEY	Unique integer which identifies a case.
DMKEY	Data model key
BEGINSTAMP	Begin time of the period of validity of a particular record. This value has to be unique in combination with JOINKEY and DMKEY. This column is used for versioning.
ENDSTAMP	End time of the period of validity of a particular record. If this column has an empty value than it is the actual record.
BLOCKID	ID of the block for which the error information is stored.
STATUS	Status of the block.
STREAMDATA	Stream that contains the data of the block.

8.5 In depth data / status table

The In Depth data / status table will be used in case you select one of the two in depth data partition types or if your data model has don't know or refusal fields.

The default name of this table is BLAISE_DATA. Its structure looks like this:

JOINKEY	Unique integer which identifies a case.
DMKEY	Data model key
BEGINSTAMP	Begin time of the period of validity of a particular record. This value has to be unique in combination with JOINKEY and DMKEY. This column is used for versioning.
ENDSTAMP	End time of the period of validity of a particular record. If this column has an empty value than it is the actual record.
FIELDID	ID of the field to which the data belongs.
STATUS	Field status of the field
STRINGDATA	Field values stored as text (data partition type in depth text) or answers to fields with string data type (data partition type in depth)
INTEGERDATA	Answers to fields with integer and enumeration data type (data partition type in depth)
FLOATDATA	Answers to fields with real data type (data partition type in depth)
DATETIMEDATA	Answers to fields with date and time data type (data partition type in depth)

8.6 Remark table

Remarks to questions are written to this table.

The default name of this table is BLAISE_REMARK. Its structure looks like this:

JOINKEY	Unique integer which identifies a case.
DMKEY	Data model key
BEGINSTAMP	Begin time of the period of validity of a particular record. This value has to be unique in combination with JOINKEY and DMKEY. This column is used for versioning.
ENDSTAMP	End time of the period of validity of a particular record. If this column has an empty value than it is the actual record.
FIELDID	ID of the field for which the remark has been created.
REMARKTEXT	Text of the remark.

8.7 Open table

This table stores answers to open (memo) fields. Only open fields which have a non-empty answer text will be stored.

The default name of this table is BLAISE_OPEN. Its structure looks like this:

JOINKEY	Unique integer which identifies a case.
DMKEY	Data model key
BEGINSTAMP	Begin time of the period of validity of a particular record. This value has to be unique in combination with JOINKEY and DMKEY. This column is used for versioning.
ENDSTAMP	End time of the period of validity of a particular record. If this column has an empty value than it is the actual record.
FIELDID	ID of the open field.
STATUS	Field status of the open field
OPENTEXT	Answer text