


Database integrity tools

Adlib
an AXIELL product

Axiell ALM Netherlands BV

Copyright © 2010-2021 Axiell ALM Netherlands BV® All rights reserved. Adlib® is a product of Axiell ALM Netherlands BV®

The information in this document is subject to change without notice and should not be construed as a commitment by Axiell ALM Netherlands BV. Axiell assumes no responsibility for any errors that may appear in this document. The software described in this document is furnished under a licence and may be used or copied only in accordance with the terms of such a licence.

Though we are making every effort to ensure the accuracy of this document, products are continually being improved. As a result, later versions of the products may vary from those described here. Under no circumstances may this document be regarded as a part of any contractual obligation to supply software, or as a definitive product description.

Contents

1 ValidateDatabase **5**

2 RemoveTagsFromData **9**

3 IndexCheck **13**
3.1 Introduction 13
3.2 Running IndexCheck 14
3.3 Typical problems with indexes 16

4 LinkRefCheck **17**
4.1 Introduction 17
4.2 Running LinkRefCheck 19
4.3 Typical problems with link references 23

5 ConvertInternalLinks **25**

6 InternalLinkCheck **31**

7 RemoveLanguageFromData **32**
7.1 To convert a multilingual field into a unilingual one 32
7.2 Remove certain multilingual data 32
7.3 The log 33

8 AddDataLanguage **34**

1 ValidateDatabase

The purpose of the *ValidateDatabase.exe* tool is to check one or more Adlib databases for any tags that have not been defined in the data dictionary (your .inf database structure files) and report on them. The tool works on SQL databases only.

In Adlib it has always been possible to store data in field tags which haven't been defined in the data dictionary. You can do that by associating an undefined tag with a screen field, by assigning a value to an undefined tag by means of an adapl or via an import mapping to one or more undefined tags. This functionality has certain advantages, but a disadvantage definitely is that you'll easily lose track of what tags are actually in use. This may lead to data corruption once you define a tag in the data dictionary, for purpose x, while that tag is already in use on a screen, for example, for purpose y. Therefore it's better to define all used tags in the data dictionary.
The Adlib API as used by wwwopac.ashx 7.2 and higher, as well as Axiell Collections, implement a strict policy concerning the definition of field tags: all tags used in records that you read or write using this software, must have been defined in the data dictionary. So when you're about to start using wwwopac.ashx for your website or Axiell Collections, you may encounter errors because of this. The ValidateDatabase tool can help you identify the tags causing the problems, after which you can still define them in the relevant .inf and solve the issue.

To be precise: the tool produces a list of valid (already defined) and invalid tags for which no definition has been found. It also counts the number of appearances of both the defined and undefined tags in the records. Further, it has a `fix` option which automatically creates provisional, generic field definitions for all unknown tags, using some default settings. Although using the `fix` option is a quick fix you may also, afterwards, consider checking the relevant records for redundant data and clean them up, and edit the settings of the automatically created fields to actually reflect the data you store in those fields.

The *ValidateDatabase.exe* tool must be called from the operating system command line – type `cmd` in the Windows Explorer address field and press **`Enter`** to open a command line window - with the following syntax:

```
ValidateDatabase <data folder> [database name | *] [check|fix]
```

Everything between [and] is optional. | means: enter either one of both arguments. Leave out all brackets.

Either specify a single database name (the name of an .inf file with or without the .inf extension) or use * to check all databases in the data folder. Use `check` to just report a list of defined and undefined tags or use `fix` to automatically create field definitions for all undefined tags.

For example, to check just the *collect* database if you placed the *\ValidateDatabase* folder containing the tool files as a subfolder in your Adlib *\data* folder and call the executable from its own folder:

```
ValidateDatabase ..\ collect check
```

Example of a partial result:

```
C:\Adlib Software\Model application 4.2 NL
SQL\data\ValidateDatabase>validatedatabase ..\ collect
check
Started: collect
Unknown tags:
LS,CO,%B,X8,CN,CH,BS,BH,CD,BP,BI,x1,x2,CR
Tag count in record (invalid tags)
LS: 15
CO: 7
%B: 15
x1: 15
x2: 6
CR: 1
Tag count in record (valid tags)
vm: 670
vi: 105
tx: 685
PB: 5
Lq: 4
Completed: collect
```

To write the entire list to a text file in the current folder, use a syntax like:

```
ValidateDatabase ..\ collect check > myundefinedtags.txt
```

To automatically create field definitions for all undefined tags in collect, you would use:

```
ValidateDatabase ..\ collect fix > myundefinedtags.txt
```

This will create field names with the syntax:

`provisional_field_for_<tag>`

For example:

```
provisional_field_for_LS
```

These fields will be normal text fields of undefined length, having only an English field name. It is recommended to rename these fields and set their other properties according to their purpose, using Adlib Designer.

2 RemoveTagsFromData

The purpose of the *RemoveTagsFromData.exe* tool is to remove tags and their content from your records in one or more Adlib databases. The reason you may want to do this is not just to clean up your databases but also because Axiell Collections and the Adlib API as used by wwwopac.ashx 7.2 and higher implement a strict policy concerning the definition of field tags: all tags used in records that you read or write using the API or Collections, must have been defined in the data dictionary. So when you're about to start using wwwopac.ashx 7.2 or higher, you'll either have to make field definitions for all undefined tags (using the *ValidateDatabase.exe* tool and Adlib Designer) or you'll have to remove the undefined tags from your data using the RemoveTagsFromData tool. The tool works on SQL databases only.

Before you start removing tags you must be absolutely sure that you are not deleting valuable data, such as undefined link reference tags or screen tags for example. Basically, the procedure to follow is this:

1. Make a backup of your SQL database to start with (just to be safe).
2. Subsequently use the ValidateDatabase tool to check one or more of your Adlib databases for any tags that have not been defined in the data dictionary and report on them.
3. Then use the RemoveTagsFromData tool with the `check` option to identify the records containing certain undefined tags. Or search on any of the problematic tags using the *Advanced search* in Adlib for Windows – e.g. use a search statement like: `<tag> = *`, and replace `<tag>` by the relevant field tag – to find the same records.
4. Now preferably check each identified or found record in the search result for the content of the tag: you can check all tags and their values stored in a record if you look at the record contents via **`Ctrl+R`**, while in detailed display mode (in Adlib for Windows). Watch out for incidental tags that have only been defined on screen (associated with a screen field): the contents of these fields are plainly visible in detailed presentation of a record, are properly stored in the database and you'll want to keep them, but a field definition in the data dictionary is still missing. You can use Designer to search for screens possibly containing a specific undefined tag: unfortunately there's no real quick way to determine if any of a list of undefined tags appears anywhere on a screen in the data source associated with the relevant database. However,

the contents of an undefined tag (obtained with **Ctrl+R**) in some randomly picked records may give you an idea of the likeliness of the tag appearing anywhere on screen.
Also watch out for undefined tags starting with the lowercase letter "l" and/or containing a number, especially tags like l1, l2, etc. as these could be link reference tags in use by linked fields: you should keep these tags and make a proper field definition for them, using Adlib Designer. In Designer you can check for every linked field if it uses a defined or undefined link reference tag by looking at the *Forward reference* property: if the property displays a field name, then the tag/field has been defined in the data dictionary all right; if the property displays just a tag, it probably hasn't been defined.

5. Finally, if you are sure your undefined tag contains no valuable data, you can remove it from these records using the RemoveTagsFromData tool with the `fix` option.

When the procedure saves an edited record, no indexes will be updated! The updating of indexes is not required if you use the tool for its intended purpose, which is to delete tags that have entered your records unintentionally at one time or another, during an import or because of some adapl procedure for example. These tags won't have indexes anyway because the fields themselves have never been specified in the data dictionary.

The *RemoveTagsFromData.exe* tool must be called from the operating system command line with the following syntax:

```
RemoveTagsFromData <data folder> [fix|check] [database list | *]
[tag list] [priref]
```

Everything between [and] is optional. | means: enter either one of both arguments. Leave out all brackets.

- The `<data folder>` is mandatory. You can enter a relative path.
- Use the literal value `fix` to remove all specified tags from the specified records or use `check` if you would just like to know in which records the specified tags have been found (and not remove any tags just yet). The default argument is `check`.
- For the `[database list]`, either specify a single database name (the name of an *.inf* file with or without the *.inf* extension), use a comma-separated list of database names (e.g. `collect,document` without a space behind the comma) or use `*` to check all databases in the data folder.

- The `[tag list]` must be a comma-separated list of the Adlib tags you wish to remove or check (e.g. `T6,aA,w9` without a space behind the commas). Instead of specifying a list you can enter `*` to target all tags in the database: with the `fix` argument this would effectively **empty** the specified records entirely (the records are not deleted but emptied)!
  If a tag contains a smaller-than (<) or larger-than (>) character you need to escape that character by enclosing it in double quotes. So a field tag like `<n` needs to be entered as `"<"n`.
- Use the optional `[priref]` argument to specify either a single record (its record number) from which the specified tags have to be removed or leave the argument out (or enter `*`) to fix or check all records in the specified databases.

For example, if you placed the *\RemoveTagsFromData* folder containing the tool files as a subfolder in your Adlib *\data* folder and call the executable from its own folder:

```
RemoveTagsFromData ..\ check collect BP
```

Example of a `check` result:

```
Started 12:11:01
Started checking database 'COLLECT.inf'
Found tag 'BP', occurrence '1' in record '151'
Found tag 'BP', occurrence '1' in record '153'
Found tag 'BP', occurrence '1' in record '154'
Found tag 'BP', occurrence '1' in record '155'
500 records processed, 0 records modified in 00:00:00,
speed: 384142 recs/minute
Completed checking database 'COLLECT.inf'
Completed '569' records, modified '0' records
```

To write the result to a text file in the current folder, use a syntax like:

```
RemoveTagsFromData ..\ check collect BP > CheckRecords.txt
```

The result allows you to check these records manually first. To actually remove the tag and its contents from these records, you could execute:

```
RemoveTagsFromData ..\ fix collect BP
```

In this example the result would be:

```
Started 12:19:06
Started fixing database 'COLLECT.inf'
Found tag 'BP', occurrence '1' in record '151'
Found tag 'BP', occurrence '1' in record '153'
Found tag 'BP', occurrence '1' in record '154'
```

```
Found tag 'BP', occurrence '1' in record '155'
500 records processed, 4 records modified in 00:00:00,
speed: 137012 recs/minute
Completed fixing database 'COLLECT.inf'
Completed '569' records, modified '4' records
```

3 IndexCheck

3.1 Introduction

The purpose of the command-line Adlib IndexCheck tool is to check whether indexes are correct, by reading all records again and comparing the current contents of the index to what is supposed to be indexed. If wrong keys appear in the current index, or if values from records are still missing in it, IndexCheck can automatically perform repairs to yield a correct index. (This of course asummes that IndexCheck is flawless in this respect.) To be more precise:

1. For all indexes, the orphan keys will be removed from the index tables: orphan keys are keys which point to non-existing records. This will be done for *all* indexes, not just for the indexes provided to IndexCheck on the command-line.
2. All records, or only the record(s) indicated in the command-line statement, will be looked up in the SQL database one by one, and for each record the following actions will be executed:
    a. Of the provided indexes (`list_of_tags`), the keys to be indexed will be retrieved from the record (let's call them record keys).
    b. For the free-text (word) indexes with keys appearing in the record keys, which have no matching word in the *wordlist* yet, a new word number will be determined and this word (the missing key) and its number will be added to the *wordlist*.
    c. Of the provided indexes, the existing keys will be retrieved from the index tables (let's call them index keys).
    d. If there are more record keys than index keys, then those extra keys will be added to the index table. IndexCheck will report this action as "MissingIndexValue…".
    e. If there are more index keys than record keys, then the index keys which no longer occur in the records will be removed from the index table. IndexCheck will report this action as "ExtraIndexValue…".

3.2 Running IndexCheck

1. For safety reasons, create a backup of your Adlib application and database, if you haven't done that yet. See the Installation guide for Museum, Library and Archive for more information. It is also wise to try out IndexCheck in a test environment before applying it to a live database.
2. Copy the IndexCheck files to a temporary folder on the machine that also runs the SQL server. This will spare your local network the extra load. (The Adlib *data* folder doesn't need to be on that same server though.)
3. To run IndexCheck as efficiently as possible, make a few settings for the database. Open the properties of your database in Microsoft SQL Server Management Studio (or a similar tool), set the *Autogrowth* size on the *Files* page to 100 MB, and on the *Options* page, set the *Recovery model* to *Simple*.
4. If you also want to run LinkRefCheck, then always run IndexCheck before LinkRefCheck, not the other way around.
5. IndexCheck can be controlled by command-line parameters. To provide these parameters, open a command line window and execute IndexCheck using the following syntax:

   ```
   <(path to)indexcheck> <path_to_the_data_folder> [fix|check]
   [list_of_databases] [list_of_tags] [priref or priref range]
   [true|false] [logFile]
   ```

   The data folder is mandatory and should point to the location where the *.inf* files are stored. Everything between [] is optional, but if you want to use e.g. the last parameter, you'll also have to use all the preceding ones.

   - If you use the `check` parameter, then IndexCheck will only report errors and does nothing to fix them, although it will also report keys that would have to be added to, or deleted from, the indexes. On the other hand, if you use `fix`, IndexCheck will check, add and delete relevant keys and will fix some other errors. The default value is `check` (or use an asterisk instead).
     Use an asterisk to indicate the position of parameters with a default value, if other entered parameters require a non-default setting. See the examples.

- `list_of_databases` is a comma-separated list (use no spaces in such lists) of Adlib-databasestructure files to check (names of .inf files without the extension). Not providing this list, or entering an asterisk, will check *all* Adlib databases.
- `list_of_tags` is a comma-separated list of Adlib tags of which the index tables must be checked. By default, IndexCheck checks all indexes. Entering an asterisk at this position also checks all indexes.
- `priref or priref range` is either the record number of a single record to process or a priref range in the format `startpriref-endpriref`. (The priref range option is available from version 1.10.1.848.) By default (if you leave a record number or record number range), IndexCheck processes all records in the provided databases. Entering an asterisk at this position also processes all records.
- `true` ignores all word indexes. LinkRefCheck requires correct indexes but doesn't do anything with word indexes. So when you run IndexCheck for this purpose, ignoring the word indexes improves the performance of IndexCheck. Moreover, Designer has an option to reindex all word indexes. The default value is `false`. Entering an asterisk at this position also means `false`.
- If you want reports of errors in indexes to be written to a log file, then provide the name of that text file in `logFile`.

6. After running IndexCheck, an overview of the results will be displayed, per database showing the number of records processed, the total number of deleted keys (in the *Extra* column), the total number of keys added to indexes (in the *Missing* column), the number of errors and the run time.
   Any errors will be listed individually as well. Of conversion errors (when field data is of the wrong data type and cannot be converted to the proper type), the relevant field tag, its field occurrence, the actual data of the field occurrence that caused the problem and the priref (record number) of the relevant record will be shown: you will have to correct those errors manually, e.g. by editing those records in Adlib or Axiell Collections.

Examples:

- `indexcheck "..\Adlib software\data" check "collect,document" * * * "check2.txt"`

- `indexcheck "..\Adlib software\data" fix "document" TI * * "check2.txt"`
- `indexcheck "..\Adlib software\data" fix * * * * "fix3log.txt"`
- `indexcheck "..\Adlib software\data" check`

3.3 Typical problems with indexes

- Because of some bug in a specific version of Adlib for Windows, Axiell Collections, import.exe or Adlib Designer, errors may have been introduced into an index.
- If the definition of a field has changed, while the index definition hasn't, keys may be too long, cut off, or of the wrong data type, etc.
- Because of some bug, duplicate words may appear in the wordlist index (which has to be unique* per data language). You can use Adcopy to solve this problem.
  Without Adcopy, you'd have to remove all duplicates in the same language from the wordlist manually, directly in the SQL table. You could also clear the entire wordlist index this way.
  In both cases you'll have to reindex all word indexes for all Adlib databases, but the second way takes more time to complete, which means more downtime. See the SQL Server and Oracle document for more information about managing the SQL database directly.

  * The SQL index on the word number of every word in the *wordlist* must always be unique, even for identical words in different languages. In Microsoft SQL Server Management Studio you can check this: open the *Tables* node of your SQL database in the *Object explorer*, and underneath it open the *Indexes* node. In the properties of the *wordlistnumberindex* you can see if it has been set to *Unique*.

4 LinkRefCheck

4.1 Introduction

The purpose of the command-line Adlib LinkRefCheck tool is to make sure that the reference in the link reference tag of linked fields in a SQL database points to an existing linked record, and that no value is stored in the linked field itself. It also empties any accidentally stored merged-in fields, because merged-in fields shouldn't be stored in the database either.
If a correct link reference is present (pointing to an existing record in the linked database), and the linked field value itself and any merged-in field values do not appear in the stored record, then things are how they should be, so nothing is changed. Different erroneous situations for a linked field can in principle exist though, which will all be fixed by LinkRefCheck as follows:

- A correct link reference is present (pointing to an existing record in the linked database), and the linked field tag itself appears in the stored record, either filled in or empty. LinkRefCheck will check if the local value in the tag matches the linked record. If not, LinkRefCheck reports the error. The linked field tag plus its value will always be removed, so the link reference remains.

- No link reference is present but a value has been stored in the linked field itself. LinkRefCheck will check whether the value appears in the linked database. If it does, the record number will be copied to the link reference tag of the linked field, and the linked field tag plus value will be removed. If it doesn't, a new linked record will be created and its record number will be copied to the linkref tag of the linked field, after which the linked field tag plus value will be removed.
  If the indexed value appears in the linked database multiple times, LinkRefCheck cannot select the proper linked record (it won't check data in the records themselves) and it will generate a message stating that multiple terms were found. The linked field tag plus value won't be removed, so you'll have to make a proper link in the relevant record manually later on (and then run LinkRefCheck again).
  When a record is forced into the linked database, LinkRefCheck has no knowledge of datasets. In the Thesaurus there usually are no datasets, so no problem there, but in a database with datasets the new record would just get the highest, first available record

number, regardless of the dataset containing that record number. In this case you would have to move (derive) records from one dataset to another manually later on, if records were forced into the wrong dataset, and maybe also edit automatically stored data like the material type.
Links on secondary, term indexed long text fields are problematic. For example, from the Collect database to the Document database there's such a link on the `documentation.title` field. In Document, the title field is primarily indexed as a word index and secondarily as a term index on a dummy field. The link from Collect must use this secondary index. LinkRefCheck will also use this index to find out if a title stored in the linked field occurs in Document already (also when earlier forced by LinkRefCheck), but when it forces a new title record in Document, it cannot store the title in the proper title field because it doesn't know the tag: it'll store the title in the dummy field instead. This means the forced records will have an empty title field, so again, you'll have to edit such forced records later on by hand to fix this. You'll know which records were forced into the linked database because of the messages LinkRefCheck generates during the process.

- The link reference points to a non-existing record in the linked database, and the linked field tag itself has (correctly) not been stored in the record. LinkRefCheck reports the error and will remove the faulty link reference.
- The link reference points to a non-existing record in the linked database, and the linked field tag itself has a value. LinkRefCheck will first remove the erroneous link reference. Then it will check whether the linked field value appears in the linked database. If it does, the record number will be copied to the link reference tag of the linked field, and the linked field tag plus value will be removed. If it doesn't, a new linked record will be created and its record number will be copied to the linkref tag of the linked field, after which the linked field tag plus value will be removed. LinkRefCheck will report the error.
- For a reversely linked field in record A a correct link reference to a record B exists while for the associated reversely linked field in record B no link reference to record A exists. LinkRefCheck will add the missing record A link reference to the relevant link reference field in record B.
- LinkRefCheck does not deal with inherited fields. Prior to version 1.10.1.1031, inheritance could lead to an error processing certain records, giving an *Object reference not set to an instance of an object* error.

4.2 Running LinkRefCheck

1. For safety reasons, create a backup of your Adlib application and database, if you haven't done that yet. See the Installation guide for Museum, Library and Archive for more information. It is also wise to try out LinkRefCheck in a test environment before applying it to a live database.

2. LinkRefCheck requires that all source and destination fields used as merged-in fields with a linked field (see the *Linked field mapping* properties tab of a linked field in the data dictionary (the database structure) in the Designer Application browser) have actual field definitions in the data dictionary, so it's not enough to use undefined tags. Use the Designer Application tester tool on your application to find out if there are such undefined merge tags in your data dictionary. If so, you should look up each relevant mapping and decide if source and/or destination field definitions should be added to the data dictionary or whether to remove that mapping: if both source and destination tags have no field definition, you can remove that row from the relevant mapping; if only the target tag has no field definition, you should create one for it; if the target field definition exists while the source tag doesn't, you should decide for yourself.

3. Copy the LinkRefCheck files to a temporary folder on the machine that also runs the SQL server. This will spare your local network the extra load. (The Adlib *\data* folder doesn't need to be on that same server though.)

4. To run LinkRefCheck as efficiently as possible, make a few settings for the database. Open the properties of your database in Microsoft SQL Server Management Studio (or a similar tool), set the *Autogrowth* size on the *Files* page to 100 MB, and on the *Options* page, set the *Recovery model* to *Simple*.

5. Only run LinkRefCheck after making sure that your indexes are correct, in other words: by using IndexCheck (or Adcopy) first.

6. LinkRefCheck can be controlled by command-line parameters. To provide command-line parameters, open a command line window – type `cmd` in the Windows Explorer address field and press **`Enter`** to open a command line window - and execute LinkRefCheck using the following syntax:

```
<(path to)linkrefcheck> <path_to_the_data_folder>
[fix|check|createtables] [list_of_databases] [list_of_tags]
[priref] [user_name] [log_file]
```

The data folder is mandatory and should point to the location where the *.inf* files are stored. Everything between [] is optional:

- If you use the parameter `check`, then LinkRefCheck will only report errors and does nothing to fix them. If you use `fix`, LinkRefCheck both checks and fixes any errors. The `createtables` parameter checks if there are index definitions for which no matching SQL table exists yet and then creates those tables. The default value is `check` (or use an asterisk instead).
  Use an asterisk to indicate the position of a parameter with a default value, if not all optional parameters must be used in the default setting. See the examples.
- `list_of_databases` is a comma-separated list (use no spaces) of Adlib-databasestructure files to check (names of .inf files without the extension). Not providing this list, or entering an asterisk, checks all Adlib databases.
- `list_of_tags` is a comma-separated list of Adlib tags of linked fields to check. By default, LinkRefCheck checks all tags. Entering an asterisk at this position also checks all tags.
- `priref` is either the number of a single record to check, a comma-separated list of prirefs to check (separate only by a comma and use no spaces) or a range of prirefs (provide the first and last priref in the range and separate them by a hyphen without spaces around it). The list and range options are available from LinkRefCheck version 1.10.1.1025. By default (without providing any prirefs), LinkRefCheck checks all records in the provided databases. Entering an asterisk at this position also checks all records.
- `user_name` is an optional user name that must be written to the management details in records edited by LinkRefCheck, if you don't want LinkRefCheck to use your own login name (the system user) for this purpose. Date and time of editing and the current database name will also be written in the record. LinkRefCheck uses the standard Adlib tags `nm`, `dm`, `tm` and `vm` to write to. If you built your database yourself, and are using these tags for other purposes, then be aware that these tags can be overwritten by LinkRefCheck. Entering an asterisk means LinkRefCheck will use the system user name.

- If you want reports of faulty link references to be written to a log file, then provide the name of that text file in `log_file`. The log file will always contain an overview comparable to the following:

  LinkRefCheck started...Check, 5-11-2010 17:10:00
  Running in folder ..\CMS.20102969\data
  Starting linkref check for database address
  Finished linkref check for database address, 17902 records checked, 0 updated.
  Link ref check completed, 5-11-2010 18:32:33

  The following messages may appear in the log file:

Message example	**Meaning**
Added missing reverse linkref 10128, link ref tag = ly in database address, record 4426	A missing reverse link was added.
Deleted link to non-existent record from record 127, field=BC, linkref=125, LinkRefTag=ly, Linked database=address	A link (the link reference tag) to a non-existing record was removed by LinkRefCheck. A possible cause for the error might have been that in the past, feedback links hadn't been set, and records in the linked database had been removed when references to them in other databases were still present.
Deleted merge data for record: 10985, Tag=BD, Occ=1, 'Data=London Historical Museum, LinkRef = 4296, LinkRefTag=lz, Linked database=address	A local (filled or empty) linked field was removed because the linked field itself should not appear in the record, only the link reference tag should.
Mismatch in record: 10277 between link reference and linked field; data is removed; reference is preserved, Occ=3, linkref=ly, value=16282, linked field=BC, 'data=Brandt,	The contents of the linked field (which shouldn't have been stored) does not match the contents of the field in the linked record referenced in the link reference field. The link has been repaired by

W.A.'	deleting the linked field and keeping the link reference tag and its contents.
New domain forced in 'thesau', priref = 3037, domain = 'SU144'	A domain was added to a linked Thesaurus record.
Record forced in 'thesau', priref = 104296, key = 'Landscape'	A linked record has been forced into the Thesaurus.
Removed circular link 10080, link ref tag = ly from database address, record 10080	A link to the record itself (a circular link) was removed.
Resolved link for record: 10599, Tag=CH, Occ=1, "Data='van'", LinkRefTag=li, linkRef=21174, Linked database=thesau	A non-processed link was processed by means of the value in the linked field, so the link reference tag was filled after which the linked field was removed.

Examples:

- `linkrefcheck "..\Adlib software\data" check "collect,document" * * "AIS" "logfiles\check2.log"`
- `linkrefcheck "..\Adlib software\data" fix * * * * "logfiles\fix3.log"`
- `linkrefcheck "..\Adlib software\data" fix collect * 2,4,100 * "logfiles\fix3.log"`
- `linkrefcheck "..\Adlib software\data" fix collect * 1-120 * "logfiles\fix3.log"`
- `linkrefcheck "..\Adlib software\data" check`

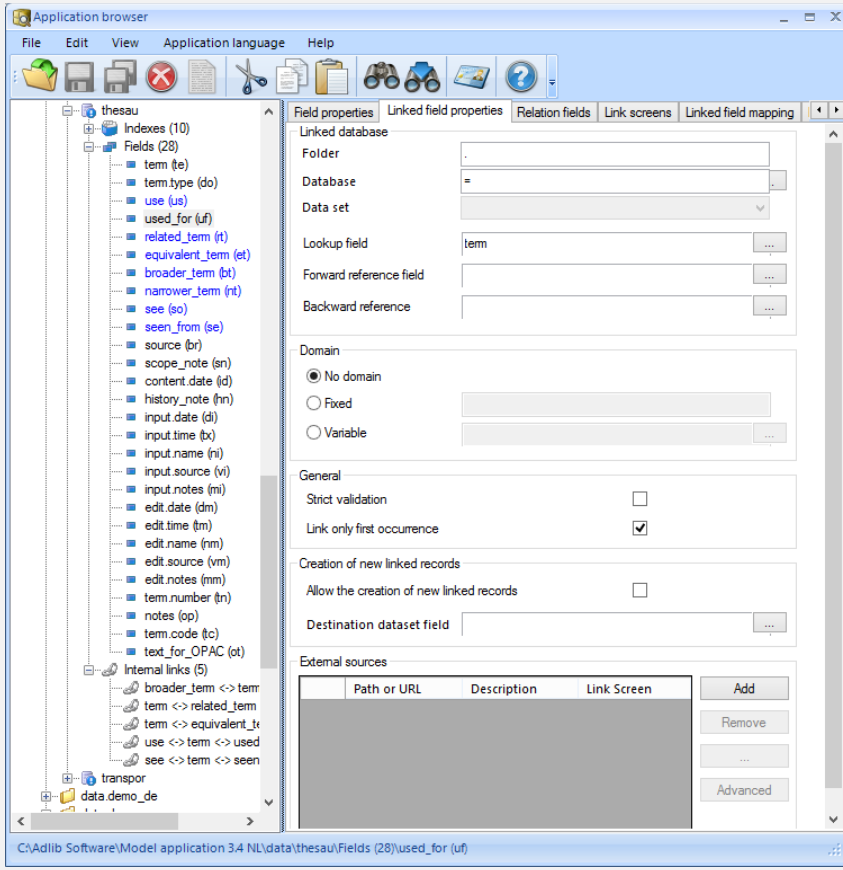
4.3 Typical problems with link references

- If many records have faulty link references (pointing to non-existing records), while feedback links are missing in the linked database, it is likely that someone removed authority records in the linked database without first cleaning up all the records referencing those authority records.
- If many records contain stored values in the linked fields, while they do have an associated link reference tag (filled in or not), it is likely that someone changed the definition of a normal field into a linked field while data was present in the normal field.

5 ConvertInternalLinks

The purpose of the Adlib *ConvertInternalLinks.exe* tool is to convert both the structure and contents of fields internally linked on value (as was the case in Adlib model applications older than version 4.2) in one or more tables in your Adlib SQL database to fields internally linked on reference. The tool takes care of everything, provided you've created a correct *preferences.xml* configuration file to instruct the tool about the relevant fields and databases.

Internal link types



Internally linked fields are fields that link to other fields in the same Adlib database. Typically, such fields appear in the *thesau* (*Thesaurus*) and *people* (*Persons and institutions*) databases, but other databases can have them as well.

Internally linked fields can be linked on value (the term registered in the linked record) or reference (the record number of the linked record). Adlib model applications older than version 4.2 typically have internally linked fields linked on value, as can be seen in the image above, where the *Forward reference field* property is empty for the *used_for* field (and all other internally linked fields): if this property does not contain a field name or tag to store the record number of the internally linked record in, the link will be on value.

Internally linked fields on reference have several advantages, amongst which: they allow for non-unique term indexes (so you can register identical terms with different meanings in separate records) and the displayed values in the internally linked fields on reference always reflect the current state of the linked record. So in model applications 4.2 and higher this type of internal link has become the default.

Adlib for Windows (*adlwin.exe*) can handle both types of internal links, but Axiell Collections and *wwwopac.ashx* (from version 3.7.14032), require the internal links in your database to have been linked on reference: if not, most API seaches on linked fields won't work any more and yield errors. This means that only if you wish to use Axiell Collections or the Adlib API to access pre-4.2 databases, then your database will have to be converted using the ConvertInternalLinks tool.

Follow the steps below to execute the conversion:

1. Edit the *preferences.xml* file that comes with the tool. You can specify multiple Adlib databases (SQL tables) and per Adlib database multiple internally linked fields. You should always convert *all* internal links in a database, not just some of them. The link reference tags you enter in this file must be tags that do not appear in the relevant data dictionary already: they will be added to the *.inf* file by the conversion procedure. An example of a *preferences.xml* file is the following:

```
<?xml version="1.0" encoding="utf-8"?>
<Preferences xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <DatabasePath>C:\Adlib\data</DatabasePath>
  <DatabaseCollection>
    <Database>
      <Name>thesau</Name>
      <LinksCollection>
        <Link>
          <BaseTag>te</BaseTag>
          <Tag>us</Tag>
          <LinkrefTag>l1</LinkrefTag>
        </Link>
        <Link>
          <BaseTag>te</BaseTag>
          <Tag>uf</Tag>
          <LinkrefTag>l2</LinkrefTag>
        </Link>
        <Link>
          <BaseTag>te</BaseTag>
          <Tag>bt</Tag>
          <LinkrefTag>l3</LinkrefTag>
        </Link>
        <Link>
          <BaseTag>te</BaseTag>
          <Tag>nt</Tag>
          <LinkrefTag>l4</LinkrefTag>
        </Link>
        <Link>
          <BaseTag>te</BaseTag>
          <Tag>rt</Tag>
          <LinkrefTag>l5</LinkrefTag>
        </Link>
        <Link>
          <BaseTag>te</BaseTag>
          <Tag>et</Tag>
          <LinkrefTag>l6</LinkrefTag>
        </Link>
        <Link>
          <BaseTag>te</BaseTag>
          <Tag>so</Tag>
          <LinkrefTag>l7</LinkrefTag>
        </Link>
        <Link>
          <BaseTag>te</BaseTag>
          <Tag>se</Tag>
          <LinkrefTag>l8</LinkrefTag>
        </Link>
      </LinksCollection>
    </Database>
  </DatabaseCollection>
</Preferences>
```

XML element	Meaning
`DatabasePath`	the path to your Adlib *data* folder.
`Database`	must contain the mapping for a single table.
`Name`	the name of the relevant *.inf* file.
`Link`	must contain a single field mapping
`BaseTag`	the tag of the lookup field for the link
`Tag`	the tag of the linked field
`LinkrefTag`	a new tag for the link reference field

2. Create a backup of your SQL database and Adlib *data* subfolder. This conversion may have far-reaching consequences if anything goes wrong. Therefore you should create a backup of your database and *.inf* files before you start the procedure, just to be safe. That way, you can always repair any errors. See the Installation guide for Museum, Library and Archive for more information about creating backups.

3. Make sure that no-one is working with Adlib and run *ConvertInternalLinks.exe* from a location that has (write) access to the Adlib *data* subfolder. The *preferences.xml* file should be present in the folder from which you run *ConvertInternalLinks.exe*. To write a log file with the result of the procedure, add `> log.txt` to the command line, e.g.:

   ```
   C:\Temp\ConvertInternalLinks.exe > log.txt
   ```

   The created log file will resemble the following:

```
Starting database 'thesau'
Creating clone of database 'thesau'
Copying records from 'thesau' to 'thesau_clone'
100 records processed in 00:00:03, speed: 1755 recs/minute
200 records processed in 00:00:03, speed: 193529 recs/minute
300 records processed in 00:00:03, speed: 214264 recs/minute
400 records processed in 00:00:03, speed: 499950 recs/minute
500 records processed in 00:00:03, speed: 666600 recs/minute
600 records processed in 00:00:03, speed: 239976 recs/minute
700 records processed in 00:00:03, speed: 333300 recs/minute
800 records processed in 00:00:03, speed: 499950 recs/minute
900 records processed in 00:00:03, speed: 428529 recs/minute
....
....
....
232600 records processed in 00:00:26, speed: 428529 recs/minute
232700 records processed in 00:00:26, speed: 749925 recs/minute
232800 records processed in 00:00:26, speed: 599940 recs/minute
Modifying data dictionary
Modifying linked field 'use' to linkref linked with tag 'l1
Modifying linked field 'used_for' to linkref linked with tag 'l2
Modifying linked field 'broader_term' to linkref linked with tag 'l3
Modifying linked field 'narrower_term' to linkref linked with tag 'l4
Modifying linked field 'related_term' to linkref linked with tag 'l5
Modifying linked field 'equivalent_term' to linkref linked with tag 'l6
Modifying linked field 'see' to linkref linked with tag 'l7
Modifying linked field 'seen_from' to linkref linked with tag 'l8
Copying records back from 'thesau_clone' to 'thesau'
Removing temporary clone of database 'thesau'
Database 'thesau' completed
```

In the background, the procedure performs the following tasks:
1. a clone (SQL) table will be created for the processed database;
2. all records that contain internal links will be copied to the clone table and will be adjusted with link references after which the locally stored terms will be removed;
3. the data dictionary (*.inf* file) of the source database will be adjusted with the new link reference fields (as specified in your configuration file), the linked fields are upgraded with the link reference tags, any merged-in fields will be added, indexes will be created for the link reference fields and local indexes will be removed;
4. the records from the clone table will then be copied back to the original table and the indexes will be updated;
5. the clone table will be removed.

4. The new link reference field names will be formatted like `<tag>_linkref`, so *l8_linkref* for example, while the index names will be formatted like `<tag>_linkr`, so *l8_linkr* for example. If you want, you can adjust adjust these names afterwards. You could change the link reference field names format to `<linked_field_name>.lref` for example, as is custom in Axiell Collections applications. Changing the field names of these new link reference fields has no impact elsewhere.
   For the automatically generated index names it can be wise to check if they are unique (case-insensitive) within the relevant database table: if not, you'll have to change the relevant name. After you change an index name, you'll have to reindex it to create a new SQL table for it. For completeness' sake (although not a requirement) you may then remove the SQL table for the index that was created by this tool automatically, because it no longer serves a purpose. Removing an index table can be done with SQL Server Management Studio.

6 InternalLinkCheck

Of internal links, *InternalLinkCheck.exe* checks if the forward and backward references match up and reports any errors. If you use the optional fix parameter, those errors are repaired as well.

The tool runs both on internal links where the links are defined on term, as well as on internal links where the links are defined on link reference.

The syntax of InternalLinkCheck.exe is:

```
InternalLinkCheck <data folder> <database> <tag|*> [fix]
```

in which `fix` is optional: without it, InternalLinkCheck only performs a check. The tag must be one of the tags in an internal link definition or an asterisk:

- If the tag is the central tag in an internal link definition with three tags, then all internal links with that central tag will be checked/fixed.
- If the tag is one of the (non-central) relation tags (like nt, bt, rt etc.) then only that specific internal link will be checked/fixed.
- If tag = *, then all internal links in the provided database will be checked/fixed.

Example:

```
InternalLinkCheck ..\data thesau te fix
```

7 RemoveLanguageFromData

RemoveLanguageFromData.exe can be used is two different ways: to convert a multilingual field into a unilingual one or to remove specific multilingual data.

Be advised to create a backup before applying this tool.

7.1 To convert a multilingual field into a unilingual one

This way to use the tool will remove all language attributes (of possibly multiple languages) and only keeps the data of the primary language.

If in the command-line command "nl-NL" is set as the primary language, while there are nl-NL and en-GB values, then all en-GB values are deleted entirely, while the nl-NL values are kept without language attribute.

Change all relevant fields to NON-multilingual before you run the tool. (The multilingual settings in the pbk can stay as they are, if desired.)

The syntax of calling the tool (which can be displayed by starting the tool without any parameters) is as follows:

```
RemoveLanguageFromData <data folder> [fix (default)|check]
[*|databases (comma separated list)] [primary language (de-
fault="en-GB")]
```

7.2 Remove certain multilingual data

The alternative way to use this tool (available from 2020-09-07) is to use it to remove data in specified languages (along with their language attributes of course) from the database whilst maintaining the multilingual character of the field.

You can specify to remove data in one of more languages. Specify an empty primary language, using two double quotes. The syntax then becomes (with an optional last parameter):

```
RemoveLanguageFromData <data folder> [fix (default)|check]
[*|databases (comma separated list)] [primary language (de-
fault="en-GB", so use "")] [specific languages to remove
(comma separated list of language attribute codes)]
```

Example, to remove data in languages iv-IV and fr-FR:

```
RemoveLanguageFromData C:\Collections\data fix collect ""
"iv-IV,fr-FR"
```

Monolingual data (data without language attributes) should never occur in multilingual fields, but if it does anyway, you can remove it (even if it has proper multilingual data too), by leaving the last parameter empty as well, like so for example:

```
RemoveLanguageFromData C:\Collections\data fix collect ""
""
```

7.3 The log

The check or fix log will only be visible in the command line window. Note that when using the `check` option, nothing is changed although the log implies it has: it states that values were deleted, while actually they weren't. The `fix` option reports the same log and has actually deleted values and/or language attributes.

8 AddDataLanguage

Multilingual fields should never contains values without language attribute. This may occur though if previously monolingual fields which already contained data, have been made multilingual by marking the relevant checkbox in the field properties in the .inf.
In Collections, if you right-click the field that was made multilingual while it already had data in it (and it won't show in the displayed record in Collections anymore), and select *Properties* in the pop-up menu, you should still see the field value without language attribute. After this tool has run with the `fix` parameter, the language attribute should be visible in the properties and the value itself should be visible in the record itself when the proper data language is active.

The command-line *AddDataLanguage.exe* is used to either just report on all missing language attributes on monolingual field values in multilingual fields or to fix them as well by adding a language attribute to the values.

Be advised to create a backup before applying this tool.

Syntax:

```
AddDataLanguage <data folder> <language> [check|fix] [database] [field]
```

- `<data folder>` - mandatory; path to the data folder of the application
- `<language>` - mandatory, in the format 'xx-YY' (e.g. `en-GB` or `de-DE`)
- `[check|fix]` - optional, choose between `check` or `fix`: `check` reports the missing language attributes and `fix` also adds them to the record; when the attribute is left out it works like `check`
- `[database]` – a comma separated list of databases to be repaired: use `*` for all databases; when the attribute is left out it works like `*`
- `[field]` – a comma separated list of fields to be repaired; use `*` for all multilingual fields; when the attribute is left out it works like `*`

The tool reports all the missing language attributes, per database, record, field and occurrence. The tool's report can be redirected to a file.

A command-line example:

```
AddDataLanguage "\\fs01\Support\TEST
MANAGEMENT\TestCases\BugFixTestApplicatiesErik\Eriks test
application 4.5.2\data" en-GB check document
```

For a `check`, nothing is done with the data language. Only with `fix` is the change applied. Do a `check` first, then a `fix`.