



Change Log

Version 1.0

- .0: First stable release with full documentation and running time reduction on OrderModifier.

Version 0.1.8

- .0: Minor, but usefull changes
 - DatabaseTarget has a new attribute which is allowing to continue the write/update process when error happens.

```
target:
  type: Database
  url: sqlite:///database.db
  table: t_table
  createTable: true
  continueOnError: true
```

- Migration on big dataset running time optimization.
- Execute function on DatabaseTarget.

You could use to load data into special or multiple table in one time or trigger changes (deleted, changed, ...) on records based on migration differences.

In 0.1.7 inactivate deleted records was sluggish, currently it's quite easy.

```
metl-differences -d delete.yml migration/current.pickle
migration/prev.pickle
```

where delete.yml is the following:

```
target:
  type: Database
  url: sqlite:///database.db
  fn: mgmt.inactivateRecords
```

mgmt.py is contains:

```
def inactivateRecords( connection, delete_buffer,
other_buffer ):

    connection.execute(
        """
        UPDATE
            t_table
        SET
            active = FALSE
        WHERE
            id IN ( %s )
        """ % ( ', '.join( [ b['key'] for b in delete_buffer ]
    ) )
    )
```

- .1: Added logger attribute to Source/Manipulation/Target elements to define specific logger method.
- .2: AppendAllExpand get skipIfFails attribute
- .3: Neo4j Target added
- .4: mETL-transfer command added to migrate and copy whole databases
- .5: Minor fix on mETL transfer
- .6: Fixed a bug in mETL-transfer when using on big datasets sometimes lost source connection.
- .7: AppendAllExpand has a new ability to do not walk the whole directory.

Version 0.1.7

- .0: Major changes and running time reduction.
 - Changed PostgreSQL target to load data more efficient (12x speed boost) by creating a workaround for psycopg2 and SQLAlchemy's slow behaviour.
 - JSON file loading now replaced to standard json package (from demjson) because faster with big (>100MB) files.
 - BigInteger type is added to handle 8bit length numbers.
 - Pickle type is added to handle serialized BLOB objects.

IMPORTANT: The alternate PostgreSQL target will work with only basic field types and lower case column names.

- .1: Added GoogleSpreadsheetTarget to write and update Spreadsheet files.
- .2: Added AppendAll expander to append files content by walking a folder.

Version 0.1.6

- .0: Changed XML converter to [xmIsquash](#) package.


IMPORTANT: It has a new XML mapping technique, all XML source map must be updated!

- For element's value: path/for/xml/element/**text**
- For element's attribute: path/for/xml/element/attributename
- .0: Fixed a bug in XML sources when multiple list element founded at sub-sub-sub level.
- .0: Fixed a bug with htaccess file opening in CSV, TSV, Yaml, JSON sources.
- .1: Fixed a bug where map ending was *
- .1: Added SetWithMap modifier and Complex type
- .2: Fixed a bug in List expander when field's value was empty.
- .2: Split transform could split a list items too.
- .2: Clean transform removes new lines.
- .3: Added Order modifier.
- .4: Added basic aggregator functions.
- .5: Added dinamicSheetField attribute to XLS target to group your data in different sheets.
- .6: Added KeepByCondition filter.
- .7: JSON, Yaml source rootIterator and XML source itemName attributes are working like fieldMaps.
- .8: Absolute FieldMap (starts with / mark) usage for JSON, XML, YAML files.
- .9: Database source has a resource attribute to handle sql statements via file.
- .10: Database source has a params attribute to add parameters to statements.
- .11: Fields has a new limit attribute for database targets. Easy to add new database types if necessary.
- .12: Boolean conversion is working for String correctly.
- .13: Added JoinByKey modifier to easily join two sources and fill out some fields.
- .14: Added `metl-generate` command to generate automaticaly Yaml configuration files.

Version 0.1.5

- .0: htaccess file opening support.
- .1: List type JSON support for database target and source.
- .1: ListExpander with map ability.

Version 0.1.4

- .0: First public release.
- .1: Remove elementtree and cElementTree dependencies.
- .2: TARR dependency link added, PyXML dependency removed.
- .3: JSON target get a compact format parameter to create pretty printed files.
- .4: Update TARR dependency.
- .5: Add missing dependency: python-dateutil
- .6: Fixed xml test case after 2.7.2 python version.
- .7: Fixed List type converter for string or unicode data. It will not split the string!
- .8: Fixed JSON source when no root iterator given and the resource file is contains only one dictionary.
- .9: Added a new operator  to convert dict into list in mapping process.
- .10: Fixed a bug in Windows when want to open a resource with absolute path.
- .11: Added ListExpander to expand list information into single fields.
- .12: XML source open via http and https protocols.



Overview

mETL is an ETL tool which has been especially designed to load elective data necessary for CEU. Obviously, the program can be used in a more general way, it can be used to load practically any kind of data. The program was written in Python, taking into maximum consideration the optimal memory usage after having assessed the Brewery tool's capabilities.

Presentations

1. [Extract, Transform, Load in Python](#) - Bence Faludi (@bfaludi), Budapest.py Meetup
Our solutions to create a new Python ETL tool from scratch.
2. [mETL - just another ETL tool?](#) - Dániel Molnár (@soobrosa), Budapest Database Meetup

A practical rimer on how to make your life easier on ETL processes - even without writing loader code.

Thirty-seconds tutorial

First of all let's see the most common problem. Want to load data into database from a text or binary file. Our example file is called authors.csv and file's structure is the following:

```
Author,Email,Birth,Phone
Duane Boyer,duaneboyer@yahoo.com,1918-05-01,+3670636943
Jonah Bazile,jonahbazile@live.com,1971-10-05,+3670464615
William Teeple,williamteeple@gmail.com,1995-07-26,+3670785797
Junior Thach,juniorthach@msn.com,1941-08-10,+3630589648
Emilie Smoak,emiliesmoak@msn.com,1952-03-08,+3670407688
Louella Utecht,louellautecht@yahoo.com,1972-02-28,+3670942982
...
```

First task to generate a Yaml configuration for mETL. This configuration file is contains the fields and types, transformation steps and source and target data. Write the following into the terminal. `config.yml` will be configuration file's name, and the example file's type is `CSV`.

```
$ metl-generate csv config.yml</pre>
```

The script will give you information about the correct attributes what you have to fill out.

```
Usage: metl-generate [options] CONFIG_FILE SOURCE_TYPE

Options:
  -h, --help                show this help message and exit
  -l LIMIT, --limit=LIMIT   Create the configuration file with examining LIMIT
                           number of records.

  --delimiter=DELIMITER
  --quote=QUOTE
  --skipRows=SKIPROWS
  --headerRow=HEADERROW
  --resource=RESOURCE
  --encoding=ENCODING
  --username=USERNAME
  --password=PASSWORD
  --realm=REALM
  --host=HOST
```

Have to add the following attributes for the generator script:

- **headerRow**: File has a header in the first row.
- **skipRows**: Because it has a header you should skip one row.
- **resource**: File's path.

Run the command with the attributes:

```
$ metl-generate --resource authors.csv --headerRow 0 --skipRows 1 csv  
config.yml</pre>
```

Script will create the Yaml configuration which could be used by mETL. You could write the configuration manually but `metl-generate` will examine the rows and determine the correct field's type and mapping.

```
source:  
  fields:  
    - map: Phone  
      name: Phone  
      type: BigInteger  
    - map: Email  
      name: Email  
      type: String  
    - map: Birth  
      name: Birth  
      type: DateTime  
    - map: Author  
      name: Author  
      type: String  
  headerRow: '0'  
  resource: authors.csv  
  skipRows: '1'  
  source: CSV  
target:  
  silence: false  
  type: Static
```

Modify the `target` because currently it will write out the information into the stdout. You have to add the database target.

```
...  
target:  
  url: postgresql://username:password@localhost:5432/database  
  table: authors  
  createTable: true
```

Script will create the table and load data into the PostgreSQL database automatically. Run the following command to start the process:

```
$ metl config.yml
```

It's done. mETL knows many source and target types and supports transformations and manipulations as well.

Documentation

Capabilities

The actual version supports the most widespread file formats with data migration and data migration packages. These include:

Source- types:

- CSV, TSV, XLS, XLSX, Google SpreadSheet, Fixed width file
- PostgreSQL, MySQL, Oracle, SQLite, Microsoft SQL Server
- JSON, XML, YAML

Target- types:

- CSV, TSV, XLS, XLSX, Google SpreadSheet - with file continuation as well
- Fixed width file
- PostgreSQL, MySQL, Oracle, SQLite, Microsoft SQL Server - with the purpose of modification as well
- JSON, XML, YAML
- Neo4j

During the development of the program we tried to provide the whole course of processing with the most widespread transformation steps, program structures and mutation steps. In light of this, the program by default possesses the following transformations:

- **Add:** Adds an arbitrary number to a value.
- **Clean:** Removes the different types of punctuation marks. (dots, commas, etc.)
- **ConvertType:** Modifies the type of the field to another type.
- **Homogenize:** Converts the accentuated letters to unaccentuated ones. (NFKD format)
- **LowerCase:** Converts to lower case.
- **Map:** Changes the value of a field to another value.
- **RemoveWordsBySource:** Using another source, it removes certain words.
- **ReplaceByRegexp:** Makes a change (replaces) by a regular expression.
- **ReplaceWordsBySource:** Replaces words using another source.
- **Set:** Sets a certain value.

- **Split:** Separates words by spaces and leaves a given interval.
- **Stem:** Brings words to a stem. (root)
- **Strip:** Removes the unnecessary spaces and/or other characters from the beginning and ending of the value.
- **Sub:** Subtracts a given number from a given value.
- **Title:** Capitalizes the first letter of every word.
- **UpperCase:** Converts to upper case.

Four groups are differentiated in case of manipulations:

1. Modifier

Modifiers are those objects that are given a whole line (record) and revert with a whole line. However, during their processes they make changes to values with the usage of the related values of different fields.

- **JoinByKey:** Merge and join two different record.
- **Order:** Orders lines according to the given conditions.
- **Set:** Sets a value with the use of fix value scheme, function or another source.
- **SetWithMap:** Sets a value in case of a complicated type with a given map.
- **TransformField:** During manipulation, regular field transformation can be achieved with this command.

2. Filter

Their function is primarily filtering. It is used when we would like to evaluate or get rid of incomplete or faulty records as a result of an earlier transformation.

- **DropByCondition:** The fate of the record depends on a condition.
- **DropBySource:** The fate is decided by whether or not the record is in another file.
- **DropField:** Does not decrease the number of records but field can be deleted with it.
- **KeepByCondition:** The fate of the record depends on a condition.

3. Expand

It is used for enlargement if we would like to add more values to the present given source.

- **Append:** Pasting a new source file identical to the used one after the actual one being used.
- **AppendAll:** Run over a folder and append the file's content into the process.
- **AppendBySource:** A new file source may be pasted after the original one.
- **Field:** Collects columns as parameters and puts them into another column with the columns' values.
- **BaseExpander:** Class used for enlargement, primarily used when we would like to multiply a record.
- **ListExpander:** Splits list-type elements and puts them into separate lines.
- **Melt:** Fixes given columns and shows the rest of the columns as key-value

pairs.

4. **Aggregator**

Aggregators are used to connect and arrange data.

- **Avg**: Used to determine the mean average.
- **Count**: Used to calculate figures.
- **Sum**: Used to determine sums.

Component figure

Source

source

- CSV
- Database
- FixedWidthText
- GoogleSpreadsheet
- JSON
- Static
- XLS
- XML
- Yarn

fields (*)

name

type & finalType

- BigInteger
- Boolean
- Complex
- Date
- DateTime
- Float
- Integer
- List
- Pickle
- String
- Text

transforms (*)

- Add
- Clean
- ConvertType
- Homogenize
- LowerCase
- Map
- RemoveWordsBySource
- ReplaceByRegexp
- ReplaceWordsBySource
- Set
- Split
- Stem
- Strip
- Sub
- Title
- UpperCase

defaultValue

map

key

defaultValues

map

Manipulations

modifier

- JoinByKey
- Order
- Set
- SetWidthMap
- TransformField

filter

- DropByCondition
- DropBySource
- DropByField
- KeepByCondition

expand

- Append
- AppendAll
- AppendBySource
- BaseExpander
- Field
- ListExpander
- Melt

aggregator

- Avg
- Count
- Sum

Target

type

- CSV
- Database
- FixedWidthText
- GoogleSpreadsheet
- JSON
- Neo4j
- Static
- TSV
- XLS
- XML
- Yarn

Installation

As a traditional Python package, installation can the most easily be carried out with the help of the following command int he mELT directory:

```
$ python setup.py install
```

or

```
$ easy_install mETL
```

Then the package can be tested with the following command:

```
$ python setup.py test
```

The package has the following dependencies: `python-dateutil`, `xlrd`, `gdata`, `demjson`, `pyyaml`, `sqlalchemy`, `xlwt`, `nltk`, `tarr`, `xlutils`, `xmlsquash`, `qspread`, `py2neo`

On Mac OSX before installation, one needs to have the following packages installed. Afterwards all packages are installed properly.

- XCode
- [Macports](#)

On Linux before installation, one needs to check that they have `python-setuptools` and in case of its absence it need to be installed with the help of `apt-get install`.

Running of the program

The program is a collection of console scripts which can be built into all systems and can even be timed with the help of cron scripts.

The programme is made up of the following scripts:

1. `metl`: A complete process can be started with the help of it on the basis of the YAML file as a parameter. The processes in the configuration should all be described by the configuration file including the exact route of input and output files.

Usage: metl [options] CONFIG.YML

Options:

-h, --help show this help message and exit

-t TARGET_MIGRATION_FILE, --targetMigration=TARGET_MIGRATION_FILE
During running, it prepares a migration file
from the
state of the present data.

-m MIGRATION_FILE, --migration=MIGRATION_FILE
Conveyance of previous migration file that
was part of
the previously run version.

-p PATH, --path=PATH Conveyance of a folder, which is added to the
PATH
variable in order that the link in the YAML
configuration could be run on an outside
python file.

-d, --debug Debug mode, writes everything out as stdout.

-l LIMIT, --limit=LIMIT
One can decide the number of elements to be
processed.
It is an excellent opportunity to test huge
files with
a small number of records until everything
works the
way they should.

-o OFFSET, --offset=OFFSET
Starting element of processing.

-s SOURCE, --source=SOURCE
If the configuration does not contain the
path of the
resource, it could be given here as well.

2. **metl-walk**: Its task is to apply the YAML file to every folder that act as parameter. The configuration files in this case do not have to contain the accessibility of input file as the script automatically carries out their substitution.

Usage: metl-walk [options] BASECONFIG.YML FOLDER

Options:

-h, --help	show this help message and exit
-p PATH, --path=PATH	Conveyance of a folder, which is added to the PATH
	variable in order that the link in the YAML configuration could be run on an outside python file.
-d, --debug	Debug mode, writes everything out as stdout.
-l LIMIT, --limit=LIMIT	One can decide the number of elements to be processed.
	It is an excellent opportunity to test huge files with
	a small number of records until everything works the
	way they should.
-o OFFSET, --offset=OFFSET	Starting element of processing.
-m, --multiprocessing	Turning on multiprocessing on computers with more than
	one CPU. The files to be processed are to be put to
	different threads. It is to be used exclusively for
	Database purposes as otherwise it causes problems!

3. **metl-transform**: Its task is to test the transformation steps of a field in a YAML file. As parameters, it requires the name of the field and the value on which the test should be based. The script will write out the changes in value step by step.

Usage: metl-transform [options] CONFIG.YML FIELD VALUE

Options:

-h, --help	show this help message and exit
-p PATH, --path=PATH	Conveyance of a folder, which is added to the PATH
	variable in order that the link in the YAML configuration could be run on an outside python file.
-d, --debug	Debug mode, writes everything out as stdout

4. **metl-aggregate**: Its task is to collect all the possible values to the field given as a parameter. Based on these values, a Map is easily made for the records.

Usage: metl-aggregate [options] CONFIG.YML FIELD

Options:

-h, --help	show this help message and exit
-p PATH, --path=PATH	Conveyance of a folder, which is added to the PATH variable in order that the link in the YAML configuration could be run on an outside python file.
-d, --debug	Debug mode, writes everything out as stdout.
-l LIMIT, --limit=LIMIT	One can decide the number of elements to be processed. It is an excellent opportunity to test huge files with a small number of records until everything works the way they should.
-o OFFSET, --offset=OFFSET	Starting element of processing.
-s SOURCE, --source=SOURCE	If the configuration file does not contain the resource path, it can be given here as well.

5. **metl-differences** : Its task is to compare two different migrations. Its first parameter is the recent migration whereas the second parameter is the older migration. The script lets us know the number of elements that have become part of the new migration, the number of elements that have been modified and the number of elements that have been left unchanged or deleted.

```
Usage: metl-differences [options] CURRENT_MIGRATION LAST_MIGRATION
```

Options:

```
-h, --help          show this help message and exit
-p PATH, --path=PATH Conveyance of a folder, which is added to the
PATH
                    variable in order that the link in the YAML
                    configuration could be run on an outside
python file.
-d DELETED, --deleted=DELETED
                    Configuration file for receiving keys of the
deleted
                    elements.
-n NEWS, --news=NEWS Configuration file for receiving keys of the
new
                    elements.
-m MODIFIED, --modified=MODIFIED
                    Configuration file for receiving keys of the
modified
                    elements
-u UNCHANGED, --unchanged=UNCHANGED
                    Configuration file for receiving keys of the
                    unmodified elements.
```

6. **metl-generate**: Prepares a YAML file from a chosen source file. In order that a configuration can be made, the initialisation and source parameters of the source are needed.

```
Usage: metl-generate [options] SOURCE_TYPE CONFIG_FILE
```

7. **metl-transfer**: Transfer all data from one database to another.

```
Usage: metl-transfer CONFIG.YML
```

Functioning

The tool **uses a YAML file for configuration**, which describes the route of the realisation as well as all the needed transformation steps.

Outline of the functioning of an average program:

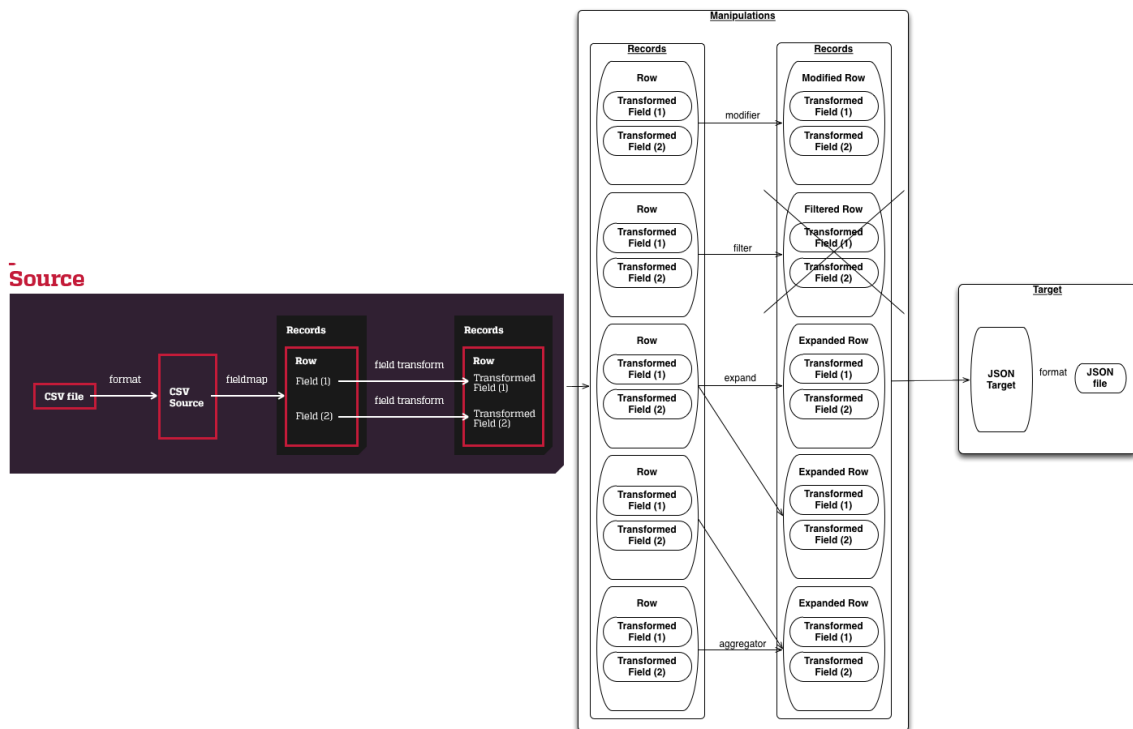
1. The programme reads the given source file.
2. Line by line, the program fills in the fields with the values with the help of a setting.
3. Different transformations are carried out individually in each field.

4. The final transformed field reaches the first manipulation where further filtering or modifications can be done to the whole line. Each manipulation sends the converted and processed line to the next manipulation step.
5. After reaching the target object, the final line is written out to the given file type.

Let's examine in detail all the components used during functioning and then let's take a look at how we can make from the listed steps YAML configuration files.

This document has two main objectives. Firstly, it defines how one can describe tasks in a YAML configuraion. Secondly, through an example, lets us glance at the python code. Also, it helps us make additional conditions and modifications when the basic tools prove to be insufficient.

Workflow



Configuration file (Yaml)

All mETL configuration files are made up of the following:


```
source:
  source: <source_type>
  ...

manipulations:
  - <listing_of_manipulations>
  ...

target:
  type: <target_type>
  ...
```

Out of these, the listing of manipulations is not necessary as in easier processes they're not even needed. Interestingly, configuration files can be adapted indefinite times, therefore similar configuration files can be made from the same „abstract” configuration. Let us examine the following example.

Let the file be called **base.yml**:

```
source:
  fields:
    - name: ID
      type: Integer
      key: true
    - name: FORMATTED_NAME
      key: true
    - name: DISTRICT
      type: Integer
    - name: LATITUDE
      type: Float
    - name: LONGITUDE
      type: Float

target:
  type: Static
```

And the next **fromcsv.yml**:

```

base: base.yml
source:
  source: CSV
  resource: input/forras.csv
  map:
    ID: 0
    FORMATTED_NAME: 1
    DISTRICT: 2
    LATITUDE: 3
    LONGITUDE: 4

```

The above file was derived from the first one and only showed where the source file and its fields are if it had to process data from the CSV source file. If we had a TSV file, then the following configuration can be written for it (where the CVS-style sign for separation is represented not by `,` but by `\t`)

```

base: fromcsv.yml
source:
  source: TSV
  resource: input/forras.tsv

```

Source

All processes start with a source file, from which the the data are retrieved. There are unique types, which all have their own settings. Their role is complex as during the ETL procedure any transformation or manipulation can retrieve further sources in order to do their operations. Their number and interlocking are critical. The **all source type** have the following data:

- **source:** Source type
- **fields:** List of fields
- **map:** Map/interlocking of fields. Not necessary here, can be given at the level of fields as well.
- **defaultValues:** Default values for the fields. Not necessary here, can be given at the level of fields as well.

An example of the configuration of a Static source YAML file:

```

source:
  source: Static
  sourceRecords:
    - [ 'El Agent', 'El Agent@metl-test-data.com', 2008, 2008 ]
    - [ 'Serious Electron', 'Serious Electron@metl-test-data.com', 2008,

```

```

2013 ]
- [ 'Brave Wizard', 'Brave Wizard@metl-test-data.com', 2008, 2008 ]
- [ 'Forgotten Itchy Emperor', 'Forgotten Itchy Emperor@metl-test-
data.com', 2008, 2013 ]
- [ 'The Moving Monkey', 'The Moving Monkey@metl-test-data.com', 2008,
2008 ]
- [ 'Evil Ghostly Brigadier', 'Evil Ghostly Brigadier@metl-test-
data.com', 2008, 2013 ]
- [ 'Strangely Oyster', 'Strangely Oyster@metl-test-data.com', 2008,
2008 ]
- [ 'Anaconda Silver', 'Anaconda Silver@metl-test-data.com', 2006, 2008
]
- [ 'Hawk Tough', 'Hawk Tough@metl-test-data.com', 2004, 2008 ]
- [ 'The Disappointed Crow', 'The Disappointed Crow@metl-test-data.com',
2008, 2013 ]
- [ 'The Raven', 'The Raven@metl-test-data.com', 1999, 2008 ]
- [ 'Ruby Boomerang', 'Ruby Boomerang@metl-test-data.com', 2008, 2008 ]
- [ 'Skunk Tough', 'Skunk Tough@metl-test-data.com', 2010, 2008 ]
- [ 'The Nervous Forgotten Major', 'The Nervous Forgotten Major@metl-
test-data.com', 2008, 2013 ]
- [ 'Bursting Furious Puppet', 'Bursting Furious Puppet@metl-test-
data.com', 2011, 2008 ]
- [ 'Neptune Eagle', 'Neptune Eagle@metl-test-data.com', 2011, 2013 ]
- [ 'The Skunk', 'The Skunk@metl-test-data.com', 2008, 2013 ]
- [ 'Lone Demon', 'Lone Demon@metl-test-data.com', 2008, 2008 ]
- [ 'The Skunk', 'The Skunk@metl-test-data.com', 1999, 2008 ]
- [ 'Gamma Serious Spear', 'Gamma Serious Spear@metl-test-data.com',
2008, 2008 ]
- [ 'Sleepy Dirty Sergeant', 'Sleepy Dirty Sergeant@metl-test-data.com',
2008, 2008 ]
- [ 'Red Monkey', 'Red Monkey@metl-test-data.com', 2008, 2008 ]
- [ 'Striking Tiger', 'Striking Tiger@metl-test-data.com', 2005, 2008 ]
- [ 'Sliding Demon', 'Sliding Demon@metl-test-data.com', 2011, 2008 ]
- [ 'Lone Commander', 'Lone Commander@metl-test-data.com', 2008, 2013 ]
- [ 'Dragon Insane', 'Dragon Insane@metl-test-data.com', 2013, 2013 ]
- [ 'Demon Skilled', 'Demon Skilled@metl-test-data.com', 2011, 2004 ]
- [ 'Vulture Lucky', 'Vulture Lucky@metl-test-data.com', 2003, 2008 ]
map:
  name: 0
  year: 2
defaultValues:
  name: 'Empty Name'
fields:
- name: name
  type: String
  key: true
- name: time
  type: Date
  finalType: String
transforms:
  - transform: ConvertType

```

```
      fieldType: String
    - transform: ReplaceByRegex
      regexp: '^[0-9]{4}-[0-9]{2}-[0-9]{2}$'
      to: '$1'
    - name: year
      type: Integer
```

The example is long and may contain data and structure not known as of yet, these will be analysed in depth later on.

The source is therefore responsible for the following:

1. Description of type and format of the file containing the data (source)
2. Description of processed data (fields)
3. Defining the interlocking between them (map)

Let us examine how we can describe the type of files containing data.

CSV

Source type used with CSV files. Its parameters of intialisation:

- **delimiter:** The sign used for separation in CSV files. By default `,` is used.
- **quote:** The character used to protect data if we're using the above mentioned demiliter. By default `"` is used.
- **skipRows:** Regulate the number of lines to be left out from the beginning of the CSV file. By default we do not leave lines out at all.
- **headerRow:** Lets us know which line contains the header of the CSV file. If given, then the interlocking will be achieved not through index (ordinal number) but through the name of a column.

Further parameters for source data:

- **resource:** Path of a CSV file, which can even be an URL.
- **encoding:** Coding of a CSV file. By default `UTF-8` should be set.

An extract example of YAML configuration with CSV source

```
source: CSV
resource: path/to/file/name.csv
delimiter: "|"
headerRow: 0
skipRows: 1
```

Database

Source type for getting data from databases. Can perform more than one function, but first let us examine the necessary parameters for getting data.

- **url:** Connection URL of the database.
- **schema:** Scheme of the database, to which one can connect. Not necessary.
- **table:** Table of the database, from which the data are extracted.
- **statement:** Unique query can be given. If it is given, then there's no need to give the `table` parameter.

In light of this, let's see two examples of YAML configuration. Let the first be the `test` table of a `SQLite` database:

```
source: Database
url: sqlite:///tests/test_sources/test_db_source.db
table: test
```

The second one is a unique query from a `PostgreSQL` database:

```
source: Database
url: 'postgresql://felhasznalo:jelszo@localhost:5432/adatbazis'
statement: "select c.*, p.* from public.t_customer as c inner join
public.t_purchase as p on ( p.cid = c.id ) where p.purchase_date >=
CURRENT_TIMESTAMP - interval '2 months'"
```

FixedWidthText

Source type for using fixed width files. Parameter of its initialisation:

- **skipRows:** Leaves the given number of lines out from the beginning of a TXT file. By default, no lines are left out.

Further parameters for source data:

- **resource:** Path of a TXT file, which can even be an URL
- **encoding:** Coding of the TXT file. By default, it is `UTF-8`

An example of an XLS configuration:

```
source: FixedWidthText
resource: path/to/file.txt
skipRows: 1
```

GoogleSpreadsheet

It is also possible to use Google Spreadsheet as a source. It doesn't require much data for initialisation, however, for getting source data, it does require lots of parameters:

- **username:** Username
- **password:** Password
- **spreadsheetKey:** key of spreadsheet
- **spreadsheetName:** name of spreadsheet
- **worksheetId:** ID of worksheet
- **worksheetName:** name of the worksheet

None of the above is mandatory, however the source is unable to work without proper data. When supplying data, the following rules apply:

1. **Public Google SpreadSheet:** Only `spreadsheetKey` is required. Usage of public spreadsheets is not perfect if the file contains `:` and `,` characters, they can give problematic results. It is Google's fault, because in case of public documents, it does not give values back per cells, but as a complete text, without protecting characters.
2. **Not Public spreadsheet:** It is mandatory to give the `username` and `password` fields, and one of 'spreadsheetKey' or 'spreadsheetName'. If we wish to refer to a given spreadsheet, it is enough to supply one of `worksheetId` or `worksheetName`

An example of a YAML configuration of a public Google Spreadsheet.

```
source: GoogleSpreadsheet
spreadsheetKey: 0ApA_54tZDwKTdHNGNVFRX3g1aE12bXhzckRzd19aNnc
```

JSON

Source type used with JSON files. Initialisation parameter:

- **rootIterator:** Name of the root that contains the list of data. Not necessary to supply, but if it is not given, then the whole JSON file will be considered as one record. En masse that data can be processed with `metl-walk`

An example of the above `rootIterator`, where the value of the `rootIterator` is `items`:

```
{
  "items": [
    {
      "lat": 47.5487066254,
      "lng": 19.0546094353,
      "nev": "Óbudaisziget",
    },
    ...
  ]
}
```

Further parameters for source data:

- **resource**: Path of the JSON file, which can even be an URL.
- **encoding**: Coding of the JSON file. By default, we expect **UTF-8**.

An example of a YAML configuration:

```
source: JSON
resource: path/to/file.json
rootIterator: items
```

Static

Source type mainly used for testing, in which the configuration file contains the records. Has only one parameter:

- **sourceRecords**: List of data in arbitrary order.

Example from above:

```
source: Static
sourceRecords:
  - [ 'El Agent', 'El Agent@metl-test-data.com', 2008, 2008 ]
  - [ 'Serious Electron', 'Serious Electron@metl-test-data.com', 2008, 2013 ]
]
  - [ 'Brave Wizard', 'Brave Wizard@metl-test-data.com', 2008, 2008 ]
  - [ 'Forgotten Itchy Emperor', 'Forgotten Itchy Emperor@metl-test-data.com', 2008, 2013 ]
```

TSV

Source type used for TSV files. It's initialisation parameters:

- **delimiter**: The separation sign in a TSV file. By default we use **\t**.

- **quote**: The character we use to protect data if the text contains the above mentioned delimiter. By default we use `"`.
- **skipRows**: Sets the number of lines to be left out from the beginning of the TSV file. By default no lines are left out.
- **headerRow**: The number of row that contains the header of the TSV can be given here. If given, the setting can only be done by column name and NOT by index (ordinal number)

Further parameters for source data:

- **resource**: Path of the TSV file, which can even be a URL
- **encoding**: Coding of the TSV file. By default, it is `UTF-8`.

Example of YAML configuration from a TSV source:

```
source: TSV
resource: path/to/file.tsv
headerRow: 0
skipRows: 1
```

XLS

Source type used for XLS files. Its initialisation parameters:

- **skipRows**: The number of lines to be left out. By default no lines are left out.

Further parameters for source data:

- **resource**: Path of the XLS file, which can even be a URL.
- **encoding**: Coding of the XLS file. By default we expect `UTF-8`.
- **sheetName**: Name or number of the sheet of the XLS file.

Example of an XLS configuration:

```
source: XLS
resource: path/to/file.xls
skipRows: 1
sheetName: Sheet1
```

XML

Source type for XML files. Its initialisation parameters:

- **itemName**: Name of the block, containing the data. Not necessary to supply, however, in that case, the whole file is considered to be one record. En masse that data can be processed with `metl-walk`.

Example of giving `itemName` if the file contains more than one record. In this case, the value of `itemName` should be `item`.

```
<?xml version="1.0" ?>
<items>
  <item>
    <lat>
      47.5487066254
    </lat>
    <lng>
      19.0546094353
    </lng>
    <nev>
      Óbudaisziget
    </nev>
  </item>
  ...
</items>
```

Further parameters for source data:

- **resource:** Path of the XML file, which can even be a URL.
- **encoding:** Coding the XML file. By default, we expect `UTF-8`. The header of the XML file contains an encoding parameter, whose coding should be same as the file's coding.

Example of an XML configuration:

```
source: XML
resource: path/to/file.xml
itemName: item
```

Later on, during mapping one should take into account that the access to the XMLs and its routes the [xml2dict](#) package will be used, therefore when giving the value, the true value will be at the `text` attribute. An example of a setting in case of `latitude`, `longitude` and `name` fields.

```
map:
  latitude: lat/text
  longitude: lng/text
  name: nev/text
```

Yaml

Source type used for YAML files. Initialisation parameters:

- **rootIterator**: Name of the root element that contains the list of data

Example of the above rootIterator the the value of the **rootIterator** is **items**:

```
items:
- district_id: 3
  lat: 47.5487066254
  lng: 19.0546094353
  nev: "\xD3budaisziget"
```

Further parameters for source data:

- **resource**: Path of the YAML file, which can even be a URL.
- **encoding**: Coding of the YAML file. By default **UTF-8** is expected.

An example of a YAML configuration:

```
source: Yaml
resource: path/to/file.yml
rootIterator: items
```

A couple of pages above it has been mentioned that the source is responsible for the following:

1. Description of the type and format of the files containing data (source)
2. Description of processed data structure (field)
3. Settings/joining between the above two (map)

We've seen the first function, let us now examine how we describe processed data.

Field

It is mandatory to give the fields of the source in the case of all source files. Naturally, if any of the fields is not necessary for the process, it does not have to be included unless we want it to be appeared in the output. But those fields in which we would like to write values must be listed, as during the process there is no possibility to add new fields. All fields can possess the following values:

- **name**: Name of the field which must be unique.
- **type**: Type of the field, by default it is String.
- **map**: Description of mapping/interlocking. Not necessary to supply here, can be given at the source level as well.
- **finalType**: Final type of the field if any change was done by transformations compared to the original type.
- **key**: Whether it is a key field or not. To make us able to use the migration capabilities in the future, set this value for each field to 'true' in all cases. **defaultValue**: Can be

used only if there is no mapping defined for the field.

- **transforms**: Transformation steps.
- **limit**: Field length used in databases.
- **nullable**: Whether the field can be left empty or not. If not, it will be stored as empty text.

An example of a YAML configuration:

```
- name: uniqueness
  type: Float
```

Important functions in Python:

- **setValue(value)**: Set the field's value.
- **getValue()**: Get the current value of the field.

An example of a Python code:

```
f = Field( 'uniquename', FloatFieldType(), key = True )
f.setValue( u'5,211' )
print repr( f.getValue() )
# 5.211
```

FieldType

Each field possesses a type. The following types are handled by mETL currently:

- **Boolean**: True-false field.
- **Complex**: Complex type for any kind of data storage. It is worth using in the case of Dict/List when we need to work with the given value in the future.
- **Date**: Date type.
- **Datetime**: Date and time type.
- **Float**: Fractional number field type.
- **Integer**: Whole number field type.
- **List**: List type for any kind of data storage.
- **String**: Text field type.
- **Text**: Long text field type.

The type value is used for conversion. Its basic task is to convert an incoming value to the defined type of element. If the conversion is unsuccessful or has an empty value (e.g. empty text) then it will result in a None value. All field types can have **None** value, the value of the type is adequate this way also.

Example from Python:

```
print repr( DateFieldType().getValue( u'22/06/2013 11:33:11 GMT+1' ) )  
# datetime.date(2013, 6, 22)
```

Usage in YAML configuration file:

```
type: Date
```

The most interesting type is **List** since it is hard to imagine in the case of certain resources. In **XML** and **JSON** it will be stored in its original format, in **CSV** and **TSV** the Python list will be converted to text format, while in the case of **Database** it will be stored as **JSON** in the **VARCHAR** field.

Transforms

The process of field transforms within mETL is done by using TARR packages. Ordinary list should be used which can contain both transforms and statements. However, on the configuration side there is a possibility to arrange the steps in order to make it easier to read by using the 'then' structure.

Its functioning is simple, it checks the transforms by statements, and at the end, if the finalType value of the field differs from the earlier field type it tries to convert the value.

Let's examine the following YAML configuration for a field:

```

- name: district
  type: Integer
  finalType: String
  transforms:
    - transform: ConvertType
      fieldType: String
    - transform: Map
      values:
        '1': Budavár
        '2': null
        '3': 'Óbuda-Békásmegyer'
        '4': Újpest
        '5': 'Belváros-Lipótváros'
        '6': Terézváros
        '7': Erzsébetváros
        '8': Józsefváros
        '9': Ferencváros
        '10': Kőbánya
        '11': Újbuda
        '12': Hegyvidék
        '13': 'Angyalföld-Újlipótváros'
        '14': Zugló
        '15': null
        '16': null
        '17': Rákosmente
        '18': 'Pestszentlőrinc-Pestszentimre'
        '19': Kispest
        '20': Pestszenterzsébet
        '21': Csepel
        '22': 'Budafok-Tétény'
        '23': Soroksár

```

What needs to be noticed in the first place is that the `Integer` type was used during the read, so we can be sure that all values which cannot be handled as numbers are stored as `None`. Since the `finalType` value is `String`, a type change will be performed during the transform as well. The first transform is a `ConvertType` which processes the above type change, while the next step is the `Map` which gives different values for a certain value. This way we created a text value field from a number field where the original name of the districts are shown. All transformations must be named by the key word `transform`.

These types of transforms can be defined for each field one by one.

Before getting into the description of more difficult transforms, let's take a look at another simple example:

```
- name: district_roman
  type: Integer
  finalType: String
  transforms:
    - transform: ConvertType
      fieldType: String
    - transform: tests.test_source.convertToRomanNumber
```

Our aim with this field is to generate a Roman numeral from a whole number value. Since mETL does not have this transform by default, we use the content of an other package. If it is not an installed package, then for mETL, the **PATH** variable can be supplemented with the **-p parameter** to load the necessary Python package.

```
@tarr.rule
def convertToRomanNumber( field ):

    if field.getValue() is None:
        return None

    number = int( field.getValue() )
    ints = (1000, 900, 500, 400, 100, 90, 50, 40, 10, 9, 5, 4, 1)
    nums = ('M', 'CM', 'D', 'CD', 'C', 'XC', 'L', 'XL', 'X', 'IX', 'V',
            'IV', 'I')

    result = ""
    for i in range( len( ints ) ):
        count = int( number / ints[i] )
        result += nums[i] * count
        number -= ints[i] * count

    field.setValue( '%s.' % ( result ) )
    return field
```

With this method, we can easily add unique transforms to our project. This method has only one shortcoming - no further parameters can be given for the transforms defined this way, so it is not sure that it can be used for more general tasks.

See the StripTransform code as an example:

```

class StripTransform( metl.transform.base.Transform ):

    init = ['chars']

    # void
    def __init__( self, chars = None, *args, **kwargs ):

        self.chars = chars

        super( StripTransform, self ).__init__( *args, **kwargs )

    def transform( self, field ):

        if field.getValue() is None:
            return field

        field.setValue( field.getValue().strip( self.chars ) )
        return field

```

This method allows us to add transforms to the system that can accept further parameters in the following way:

```

transforms:
...
- transform: package.path.StripTransform
  chars: -
...

```

The above example is not the best since the word 'Transform' never needs to be added after the transform name that are default in mETL, and no paths need to be supplied either.

It was mentioned that transforms supported statements as well. Let's see an example YAML configuration for this case:

```

- name: intervalled
  type: Date
  map: created
  transforms:
    - statement: IF
      condition: IsBetween
      fromValue: 2012-02-02
      toValue: 2012-09-01
      then:
        - transform: ConvertType
          fieldType: Boolean
          hard: true
          defaultValue: true
    - statement: ELSE
      then:
        - transform: ConvertType
          fieldType: Boolean
          hard: true
        - transform: Set
          value: false
    - statement: ENDIF
  finalType: Boolean

```

The above reads a date field from which a true-false value is generated by the end of the process. To achieve this, statements are used. Like in every low-level programming language, **IF** needs to be closed with **ENDIF**. The above example examines whether the read date is between two intervals or not. If yes, it takes the 'true' value, otherwise it will end up as 'false'. The above example also shows several possibilities for the value set.

In the case of statements, the key word **statement** must be used instead of **transform**. For conditions, the key word **condition** is the needed one, but first, let's see what conditions exist and how their parametrization works.

Condition

Each condition uses the key word **condition**, but it does not have importance on its own, it is only used by statements and certain manipulation objects for decision making. One condition **decides the true or false value for exactly one field**, it cannot be used for entire lines or for correlations between fields!

Conditions work the following way in Python:

```

f = Field( 'uniquename', FloatFieldType(), key = True, defaultValue =
'5,211' )
print repr( IsBetweenCondition( 5.11, '5,2111' ).getResult( f ) )
# True

```


As in the case of ordinary transforms, unique conditions can be defined here also in the following way:

```
@tarr.branch
def IsGreaterThanFiveCondition( field ):

    return field.getValue() is not None and field.getValue() > 5
```

Also, the above can be done in a parameterized way:

```
class IsGreaterCondition( metl.condition.base.Condition ):

    init = ['value']

    # void
    def __init__( self, value, *args, **kwargs ):

        self.value = value

        super( IsGreaterCondition, self ).__init__( *args, **kwargs )

    # bool
    def getResult( self, field ):

        if field.getValue() is None:
            return False

        return field.getValue() > field.getType().getValue( self.value )
```

Here the code of one of the built-in conditions can be seen. The below version is longer than the above one as in this case, the number from which we want to get a bigger value can be conveyed as a parameter, and also a type conversion is processed on the result number to make the evaluation occur for the same types.

IsBetween

Field value falls into a given interval or not. It makes sense to use only in the case of **Integer**, **Float**, **Date**, **DateTime** types. Its parameters:

- **fromValue**: Minimum value of the interval
- **toValue**: Maximum value of the interval

An example of a YAML configuration:

```
condition: IsBetween
fromValue: 2012-02-02
toValue: 2012-09-01
```

IsEmpty

Checks whether the result field is empty or not. No parameters are expected and can be used for all types.

An example of a YAML configuration:

```
condition: IsEmpty
```

IsEqual

The value of the field is the same as the value of the parameter. This condition can be used for all types. Its parameter:

- **value:** The value that is examined during the comparison

An example of a YAML configuration:

```
condition: IsEqual
value: 2012-02-02
```

IsGreaterAndEqual

The value of the field is greater than or equal to the value of the parameter. This condition can be used for all types. Its parameter:

- **value:** The value that is examined during the comparison

An example of a YAML configuration:

```
condition: IsGreaterAndEqual
value: 2012-02-02
```

IsGreater

The value of the field is greater than the value of the parameter. This condition can be used for all types. Its parameter:

- **value:** The value that is examined during the comparison

An example of a YAML configuration:

```
condition: IsGreater
value: 2012-02-02
```

IsLessAndEqual

The value of the field is less than or equal to the value of the parameter. This condition can be used for all types. Its parameter:

- **value:** The value that is examined during the comparison

An example of a YAML configuration:

```
condition: IsLessAndEqual
value: 2012-02-02
```

IsLess

The value of the field is less than the value of the parameter. This condition can be used for all types. Its parameter:

- **value:** The value that is examined during the comparison

An example of a YAML configuration:

```
condition: IsLess
value: 2012-02-02
```

IsIn

The value of the field is one of the values of the parameter. This condition can be used for all types. Its parameter:

- **values:** The list of values that are examined during the comparison

An example of a YAML configuration:

```
condition: IsIn
values:
  - MICRA / MARCH
  - PATHFINDER
  - ALMERA TINO
  - PRIMASTAR
```

IsInSource

It works in a quite similar way as IsIn, but the values used in the examination are loaded from an other source file and the inclusion of the field value is checked from this other file.

Parameter:

- **join**: How the field is called in the other source which contains the value the IsIn condition will be applied to.

There are other parameters as well since the entire **Source** configuration must be attached to this condition.

An example of a YAML configuration:

```
condition: IsInSource
source: Yaml
resource: examples/vins.yml
rootIterator: vins
join: vin
fields:
  - name: vin
    type: String
```

IsMatchByRegexp

It uses a regular expression for the evaluation of the field value. The given field is considered successful if the regular expression interlocks with it. Its parameters:

- **regexp**: Regular expression to be examined.
- **ignorecase**: Ignore the lower and upper case during the assessment of the regular expression. It differentiates them by default!

An example of a YAML configuration:

```
condition: IsMatchByRegexp
regexp: '^.*[0-9]+.*$'
ignorecase: false
```

Statement

The key word **statement** must be used instead of **transform**. Statements can only be used during the transform steps of fields to create the final form of field value and to do a successful data cleanup. Statements **can be embedded into each other without amount limit** but all of them must be closed.

IF

It serves as an "If, then" condition just as in regular programming languages. Each **IF** must be followed by an **ENDIF** later on. Its parameter:

- **condition**: Condition with all its needed parameters.

It is not necessary, but in the YAML configuration a **then** can be added to it as well which contains the transforms belonging to it.

IFNOT

It serves as an "If not, then" condition. The same rules and parameters apply to it as for **IF**. Its parameter:

- **condition**: Condition with all its needed parameters.

ELIF

This statement is used if the condition is not met but we want to define a new one. It can be used between **IF** and **ENDIF**, but always before **ELSE**. The same rules and parameters apply to it as for **IF**. Its parameter:

- **condition**: Condition with all its needed parameters.

ELIFNOT

The same rules and parameters apply to it as for **ELIF**, but this is met if the condition is denied. Its parameter:

- **condition**: Condition with all its needed parameters.

ELSE

It is used if the condition is not met and we do not want to set further conditions, but a path without condition is needed where the transform is able to go. Must be used between **IF** and **ENDIF**. If **ELIF** or **ELIFNOT** is present, this statement must be added after them. It does not have any parameter.

ENDIF

Key word to close an **IF** condition. It does not have any parameter.

ReturnTrue

It exists from the condition and ends the transforms. It does not have any parameter.

Transform

It was mentioned that the list of transforms could be defined to fields. These transformations are labelled with the key word **transform** after which the name of the used transformation must be added. The followings are available in the system.

Add

It adds a number to the value of the field. Can be used only in the case of **Integer** and **Float** fields. Its parameters:

- **number**: The number with which we want to increase the value of the field.

An example of a YAML configuration:

```
- transform: Add
  number: 4
```

An example for the result of the above transform:

```
12
=> 16
```

Clean

Removes the different staves from the defined field. It is important that it can be used only in the case of **String** and **Text** fields. Its parameters are not mandatory, but can be redefined:

- **stopChars**: Which characters to remove from the values of the field. By default: „!?“
- **replaces**: List of value pairs prescribing what to replace with what as part of the clean process.

An example of a YAML configuration:

```
- transform: Clean
  replaces:
    many: 1+
```

An example for the result of the above transform:

```
' That is a good sentence, which is contains many english word! '
=> 'That is a good sentence which is contains 1+ english word'
```

ConvertType

Modifies the type of the field to an other type. Since not all field types can be converted to an other type without loss, more parameters need to be set here.

- **fieldType**: Name of a new field type.
- **hard**: With this forced modification request, the current values are destroyed. It has a false value by default.
- **defaultValue**: Sets default value by forced modification. No default values are defined by default.

If we want to convert a date into text, there is no need for hard mode since this conversion is quite easily processed and in a lucky case, it can be executed in the other direction as well. Though the hard mode must be used to convert the value of a date field to Boolean. It is important to note that **the finalType value of the field does not work on hard principle**, so that must be assured with a transform before.

An example of a YAML configuration:

```
- transform: ConvertType
  fieldType: Boolean
  hard: true
  defaultValue: true
```

or

```
- transform: ConvertType
  fieldType: String
```

Homogenize

It changes the accentuated characters to non-accentuated ones in the case of **String** and **Text** fields. It is a very frequent process if we want to pair values to data coming from other sources since the quality of the data can be questionable, but this way we can make crosschecks easily. No parameter is expected.

An example of a YAML configuration:

```
- transform: Homogenize
```

An example for the result of the above transform:

```
u'árvíztűróttűrkőrfúrógépÁRVÍZTŰRÓTTŰRKÖRFÚRÓGÉP  
=> 'arvizturotukorfurogeparvizturotukorfurogep'
```

LowerCase

It changes the field value to lowercase in the case of **String** and **Text** fields. No parameter is expected.

An example of a YAML configuration:

```
- transform: LowerCase
```

An example for the result of the above transform:

```
'That is a good sentence, which is contains many english word!'  
=> 'that is a good sentence, which is contains many english word!'
```

Map

It changes the value of the field to other values. It needs to be given as key-value pairs. It works appropriately only in the case of **String** and **Text** field types. Its parameters:

- **values**: Group of key-value pairs that contains the convert values.
- **ignorecase**: Whether to ignore the difference between lowercase and uppercase during the evaluation.
- **elseValue**: Not a mandatory parameter. It should be given if we want to modify all values to anything not included in the defined list.
- **elseClear**: Not a mandatory parameter. It should be given if we want to clear all values that are not among the values list.

An example can be the previously seen YAML configuration:


```
- transform: Map
  values:
    '1': Budavár
    '2': null
    '3': 'Óbuda-Békásmegyer'
    '4': Újpest
    '5': 'Belváros-Lipótváros'
    '6': Terézváros
    '7': Erzsébetváros
    '8': Józsefváros
    '9': Ferencváros
    '10': Kőbánya
    '11': Újbuda
    '12': Hegyvidék
    '13': 'Angyalföld-Újlipótváros'
    '14': Zugló
    '15': null
    '16': null
    '17': Rákosmente
    '18': 'Pestszentlőrinc-Pestszentimre'
    '19': Kispest
    '20': Pestszenterzsébet
    '21': Csepel
    '22': 'Budafok-Tétény'
    '23': Soroksár
```

which results in the following

```
'4'
=> 'Újpest'
```

RemoveWordsBySource

It removes words from an arbitrary sentence in **String** or **Text** field types by using an other source file. Words are separated by spaces, so it is highly recommended to run **Clean** before the removal.

It does not have own parameters, but the entire **Source** configuration is needed in this case also. The source can contain only one field. If more fields are included, the transform considers only the first one and removals will be done based on the value defined there.

An example of a YAML configuration:

```
- transform: RemoveWordsBySource
  source: CSV
  resource: materials/hu_stopword.csv
  fields:
    - name: word
      type: String
      map: 0
```

ReplaceByRegexp

It executes a replacement based on regular expressions in **String** and **Text** field types. Parameters that can be used:

- **regexp**: Regular expression based on which the replacement can be done.
- **to**: The output/target of the replacement. Opposite to the usual Python syntax, **\$** must be used instead of **** to paste the highlighted parameters
- **ignorecase**: Whether to differentiate between lowercase and uppercase during the assessment of regular expressions. Differentiation is in place by default.

An example for YAML configuration where only the year-month pair is kept from a text based date format:

```
- transform: ReplaceByRegexp
  regexp: '^[0-9]{4}-[0-9]{2}-[0-9]{2}$'
  to: '$1'
```

The above results in the following:

```
'2013-04-15'
=> '2013-04'
```

ReplaceWordsBySource

It replaces words in **String** or **Text** field types by using an other source file. Its parameters:

- **join**: How the field is called in the other source which contains the same value based on which we want to join the current source with the other one. The field name must be identical in both sources!

There are other parameters as well since the entire **Source** configuration must be attached to this condition. **Important to note that the source defined here can contain only 2 fields - including the field defined during join!** Thus the value to which the replacement takes places will be a column not part of the join condition!

An example of a YAML configuration:

```
- transform: ReplaceWordsBySource
  join: KEY
  source: CSV
  resource: materials/hu_wordtoenglish.csv
  fields:
    - name: KEY
      type: Integer
      map: 0
    - name: VALUE
      type: String
      map: 1
```

Set

It performs value setting in the case of any field types. Its parameters:

- **value:** New field value

In the case of String and Text fields it is possible to paste the old value into the new one. Let's see a YAML configuration example for this:

```
- transform: Set
  value: '%(self)s or the new string'
```

An example for the result of the above transform:

```
'Myself'
=> 'Myself or the new string'
```

Split

It separates the words by the spaces and leaves the defined interval in the case of **String** and **Text** field types. It is important to note that an exact number can be given (e.g.: 1) meaning that the words with that index will be kept. Intervals can be defined as well separated by a colon (e.g.: 2:-1). Its parameters:

- **idx:** Index of the excerption, numbered from 0.
- **chars:** Based on which character should the split take place. Whitespace is set by default.

An example of a YAML configuration:

```
- transform: Split  
  idx: '1:-1'
```

An example for the result of the above transform:

```
'contains hungarian members attractive sadness killing'  
=> 'hungarian members attractive sadness'
```

Stem

It brings the words in **String** and **Text** fields to their stem. Words are separated by spaces, so in most cases the usage of **Clean** transform is necessary. Its parameters:

- **language**: Language of stemming

An example of a YAML configuration:

```
- transform: Stem  
  language: English
```

An example for the result of the above transform:

```
'contains hungarian members attractive sadness killing'  
=> 'contain hungarian member attract sad kill'
```

For process execution the nltk **SnowballStemmer** package is used.

Strip

Removes the unnecessary spaces or other characters from the beginning and end of the value. Can be used only in the case of **String** and **Text** fields. Its parameters:

- **chars**: What characters to be removed from the beginning and end of the text. Whitespace is set by default.

An example of a YAML configuration:

```
- transform: Strip
```

An example for the result of the above transform:

```
' That is a good sentence, which is contains many english word! '  
=> 'That is a good sentence, which is contains many english word!'
```

Sub

It subtracts a number from the field value. Can be used only in the case of **Integer** and **Float** fields. Its parameters:

- **number**: The number with which we want to decrease the actual value of the field.

An example of a YAML configuration:

```
- transform: Sub  
  number: 4
```

An example for the result of the above transform:

```
12  
=> 8
```

Title

It capitalizes each word. Can be used only in the case of **String** and **Text** fields. No parameter is expected.

An example of a YAML configuration:

```
- transform: Title
```

An example for the result of the above transform:

```
'That is a good sentence, which is contains many english word!'  
=> 'That Is A Good Sentence, Which Is Contains Many English Word!'
```

UpperCase

It changes the field value to upper case in the case of **String** and **Text** fields. No parameter is expected.

An example of a YAML configuration:

```
- transform: UpperCase
```

An example for the result of the above transform:

```
'That is a good sentence, which is contains many english word!'  
=> 'THAT IS A GOOD SENTENCE, WHICH IS CONTAINS MANY ENGLISH WORD!'
```

It has already been mentioned that the source is responsible for the followings:

1. To describe the type and form of the resource containing data (source)
2. To describe the data structure read (fields)
3. To define the mapping between the above items (map)

We have already covered the first 2 points, let's see how the mapping of the processed data lines works for the above defined arbitrary fields.

FieldMap

We have a resource which values we want to read, and we have fields we want to put the values in for each line. The only missing item is to create a mapping which we can define for all sources.

For fields, fieldmap can be defined in two places:

1. In the **Source** record under the **map** parameter.

```
source:  
...  
map:  
  MEZONEV: 0  
  MASIKMEZONEV: 2  
...
```

2. Within the **Field** itself as **map**.

```

source:
  ...
  fields:
    ...
    - name: MEZONEV
      map: 0
    - name: MASIKMEZONEV
      type: Integer
      map: 2
    ...
  ...

```

Both will produce the same result. The first version is the better choice if we want to derieve a configuration file from it, since in this case, the fields do not need be redefined. The second version is better in the sense that it is more transparent since everything is where they belong. **If no map is defined for a field, then values are searched based on the field name by default.**

Each map means a path to the "data". The path can contain words, numbers (indices) and the combinations of them divided by a `/`.

In the light of this, let's see a more complex example based on which it will be easier to understand the process. The `XML`, `JSON`, `YAML` resources could contain multidimensional lists, but when the data is coming from `Database`, `GoogleSpreadsheet`, it has to be a one-dimensional list.

```

python_dict = {
  'first': {
    'of': {
      'all': 'dictionary',
      'with': [ 'many', 'list', 'item' ]
    },
    'and': [ (0, 1), (1, 2), (2, 3), (3, 4) ]
  },
  'filtered': [ {
    'name': 'first',
    'value': 'good'
  }, {
    'name': 'second',
    'value': 'normal'
  }, {
    'name': 'third',
    'value': 'bad'
  } ],
  'emptylist': {
    'item': 'itemname'
  },
  'notemptylist': [

```

```

        { 'item': 'itemname' },
        { 'item': 'seconditemname' }
    ],
    'strvalue': 'many',
    'strlist': [ 'many', 'list', 'item' ],
    'root': 'R'
}

print repr( metl.fieldmap.FieldMap({
    'list_first': 'first/of/with/0',
    'list_last': 'first/of/with/-1',
    'tuple_last_first': 'first/and/-1/0',
    'not_existing': 'first/of/here',
    'root': 'root',
    'dict': 'first/of/all',
    'filtered': 'filtered/name=second/value',
    'list': 'filtered/*/value',
    'emptylistref': 'emptylist/~0/item',
    'notemptylistref': 'notemptylist/~0/item',
    'strvalue': 'strvalue',
    'strvalue1': 'strvalue/!/0',
    'strvalue2': 'strvalue/!/1',
    'strlist1': 'strlist/!/0',
    'strlist2': 'strlist/!/1'
}).getValues( python_dict ) )

# {'list_first': 'many', 'not_existing': None, 'dict': 'dictionary',
'tuple_last_first': 3, 'list_last': 'item', 'root': 'R', 'filtered':
'normal', 'list': ['good','normal','bad', 'emptylistref': 'itemname',
'notemptylistref': 'itemname', 'strvalue': 'many', 'strvalue1': 'many',
'strvalue2': None, 'strlist1': 'many', 'strlist2': 'list' ]}

```

If several data sources are used - like **CSV**, **TSV**, **XLS**, lists arrive. Note that in the case of **CSV** and **TSV**, it can be achieved that they receive one-dimensional values in the above format by defining the **headerRow** parameter.

```

python_list = [ 'many', 'list', 'item' ]

print repr( metl.fieldmap.FieldMap({
    'first': 0,
    'last': '-1',
    'not_existing': 4
}).getValues( python_list ) )

# {'last': 'item', 'not_existing': None, 'first': 'many'}

```

More important operators:

- **/**: Defines a next level in the given path/mapping.
- *****: Checks all elements in the case of lists. If we want to save it in this format instead of converted text, the usage of List type and JSON or XML target type is recommended. This operator is used mainly in the case of XML and JSON sources.
- **~**: Test operator (List 2 Dict) if both list and dict can exist on the same level. If list exists, it can be defined what we look for, if dict exists, nothing happens, the process goes on in the given path from the next element. This operator is used in the case of XML and JSON sources.
- **!**: Operator that converts (List 2 Dict). It is used if we want to get a list but it is not known whether we will get that or not. This operator is used in the case of XML sources.

Manipulation

After the whole line is processed, the values are in the fields and the transforms are done on the field level, there is a possibility to manipulate the entire, cleaned values based on their correlations. There are 4 key words that can be used - each of them labels a single type: **modifier**, **filter**, **expand**, **aggregator**. We will mainly use them during our more complex tasks (e.g. API communication) Manipulation steps can follow each other **in any order**, regardless of the type. As soon as one of them finishes, it gives the result to the next one. This process continues until the **Target** object is reached.

```

manipulations:
- filter: DropByCondition
  condition: IsMatchByRegexp
  regexp: '^.*\-.*$'
  fieldNames: name
- modifier: Set
  fieldNames:
    - district_search
    - district_copy
  value: '%(district)s'
- modifier: TransformField
  fieldNames: district_copy
  transforms:
    - statement: IFNot
      condition: IsEmpty
      then:
        - transform: Set
          value: '%(self)s, '
    - statement: ENDIF
- modifier: TransformField
  fieldNames:
    - name_search
    - district_search
  transforms:
    - transform: Clean
    - transform: LowerCase
    - transform: Homogenize
- modifier: Set
  fieldNames: formatted_name
  value: '%(district_roman)s kerület, %(district_copy)s%(name)s'
- filter: tests.test_source.DropIfSameNameAndDistrict
- filter: DropField
  fieldNames:
    - name_search
    - district_copy
    - district_search
    - district_roman
    - district_id
    - region_id

```

The above example will not be explained in details, the main points are to show the key word usage and the format.

Modifier

Modifiers are those objects that are given a whole line (record) and always return with a whole line. However, during their processes they make changes to values with the usage of the related values of different fields. In manipulations they always start with the key word **modifier** and we will use them most of the time during our work.

Before examining what system level modifiers mETL has, let's see how we can add new ones, as this step will be needed most frequently.

```
import urllib, demjson
from metl.utils import *

class MitoAPIPhoneSearch( Modifier ):

    # str
    def getURL( self, firstname, lastname, city ):

        return 'http://mito.api.hu/api/KEY/phone/search/hu/%(firstname)s/%
(lastname)s/%(city)s' % {
            'firstname': urllib.quote( firstname.encode('utf-8') ),
            'lastname': urllib.quote( lastname.encode('utf-8' ) ),
            'city': urllib.quote( city.encode('utf-8' ) )
        }

    # FieldSet
    def modify( self, record ):

        url = self.getURL(
            record.getField('FIRSTNAME').getValue(),
            record.getField('LASTNAME').getValue(),
            record.getField('CITY').getValue()
        )

        fp = urllib.urlopen( url )
        result = demjson.decode( fp.read() )
        phones = list( set([
            r.get('phone',{}).get('format',{}).get('e164') \
            for r in result['result']
        ]))

        record.getField('PHONENUMBERS').setValue( u', '.join( phones ) )

        return record
```

This example shows that by using 3 values included from any source (surname, first name, city) we create a new value (phone number list). But we gather the data through an API request. The above does not have parameters, it can be easily embedded in the process.

```
- modifier: package.path.MitoAPIPhoneSearch
```

We need to pay attention to two things during extension:

1. Needs to be derieved from **Modifier** class

2. `modify` function needs to be rewritten and it should get back with `record`

JoinByKey

It joins two sources by a key defined by the inner source. During the process, the key fields must be highlighted for the inner source so those fields will be searched in the outer source. If there is a match, the outer source is refreshed by the fields listed in the `fieldNames`. **The same name must be given** for both the key and the fields that need to be refreshed. In the case of key based join, only one record can belong to one line.

- **fieldNames:** Which fields are to be updated. Fields must have the same name in both sources!

An example of a YAML configuration:

```

source:
  source: XML
  resource: outer_source.xml
  ...
  itemName: property
  fields:
    - name: originalId
      map: "source-system-id"

    - name: agentId
      map: "agent-id"

    ...

    - name: phone
    - name: email

manipulations:
  ...
  - modifier: JoinByKey
    source: XML
    resource: inner_source.xml
    itemName: agent
    fieldNames:
      - phone
      - email
    fields:
      - name: agentId
        map: "agent-id"
        key: true
      - name: name
        map: "agent-name/text"
      - name: phone
        map: "agent-phone/text"
        transforms:
          - transform: test.convertToE164
      - name: email
        map: "agent-email/text"

target:
  type: JSON
  ...

```

Order

It orders the records based on the defined fields. Its parameters:

- **fieldNamesAndOrder:** On which fields should the ordering occur and in which order. Only **ASC** and **DESC** can be given as an order.

An example of a YAML configuration:

```
- modifier: Order
  fieldNamesAndOrder:
    - year: DESC
    - name: ASC
```

Set

It executes value setting by using fixed value scheme, function or other source. It is the most commonly used modifier, but in order to get a faster and optimal processing, it is worth writing an own modifier. its parameters for initialization:

- **fieldNames:** On which fields should the value setting take place.
- **value:** New field value.

The functioning of Set is complicated. It can be extended with a **fn** parameter as well, where an arbitrary value setting function can be defined for it, and also an entire source description can be given.

Types of usage:

1. Value modification

In short, value modification can be done based on the actual field values by putting the names of the fields into value parameter. Value set will be performed for all fields listed in **fieldNames**.

```
- modifier: Set
  fieldNames: formatted_name
  value: '%(district_roman)s kerület, %(district_copy)s%(name)s'
```

2. Value modification by using function

For performing a complex calculation, it is worth using the **Set** modifier this way. The function needs to be created by our own, therefore the **-p** parameter should be used here as well when running the metl script.

```
- modifier: Set
  fieldNames: age
  fn: utvonal.calculateAge
```

For the above, the following function can be written:

```
def calculateAge( record, field, scope ):

    if record.getField('date_of_birth').getValue() is None:
        return None

    td = datetime.date.today() -
record.getField('date_of_birth').getValue()
    return int( td.days / 365.25 )
```

In the above function, **record** means the whole line, **field** defines the field that needs to be set (this function is carried out for all values listed in **fieldNames**), while **scope** stands for the **SetModifier**.

3. Value modification by using other source

This is the most difficult modifier type, but if the optimal speed is important, it is worth redefining based on the data structure of the known source. **fn** and **source** need to be given as well.

```
- modifier: Set
  fieldNames: EMAILFOUND
  fn: utvonal.setValue
  source: TSV
  resource: utvonal/masikforras.tsv
  fields:
    - name: EMAIL
    - name: FIRSTNAME
    - name: LASTNAME
```

The following function belongs to it:

```
def setValue( record, field, scope ):

    return 'Found same email address' \
        if record.getField('EMAIL').getValue() in \
            [ sr.getField('EMAIL').getValue() for sr in
scope.getSourceRecords() ] \
        else 'Not found same email address'
```

SetWithMap

Sets field's values based on the mapping of a Complex field.

- **fieldNamesWithMap**: Field names and map paths on which the setting must be performed.

- **complexFieldName:** Name of a complex field from which we want to derieve the value. Types can be `List` or `Complex`.

Transforms must be done one by one for each field listed among `fieldNames`.

An example of a YAML configuration:

```
source:
  source: JSON
  fields:
    - name: LISTITEMS
      map: response/tips/items/*
      type: List
    - name: LISTELEMENT
      type: Complex
    - name: CREATEDAT
      type: Integer
    - name: TEXT
    - name: CATEGORIES
      type: List

manipulations:
  - expand: ListExpander
    listFieldName: LISTITEMS
    expandedFieldName: LISTELEMENT
  - modifier: SetWithMap
    fieldNamesWithMap:
      CREATEDAT: createdAt
      TEXT: text
      CATEGORIES: venue/categories/*/id
    complexFieldName: LISTELEMENT
  - filter: DropField
    fieldNames:
      - LISTELEMENT
      - LISTITEMS
```

TransformField

It performs a regular field level transormation during the manipulation step. Parameters of its initialization:

- **fieldNames:** Name of fields on which transforms must be performed.
- **transforms:** List of field level transforms.

Transforms must be done one by one for each field listed among `fieldNames`.

An example of a YAML configuration:


```
- modifier: TransformField
  fieldNames: district_copy
  transforms:
    - statement: IFNot
      condition: IsEmpty
      then:
        - transform: Set
          value: '%(self)s, '
    - statement: ENDIF
```

Filter

Their function is primarily filtering. It is used when we would like to evaluate or get rid of incomplete or faulty records as a result of an earlier transformation.

If we want to put a new filter in the system, the following can help:

```
from metl.utils import *

class MyFilter( Filter ):

    # bool
    def isFiltered( self, record ):

        return not record.getField('MEGMARADJON').getValue()
```

DropByCondition

The fate of the record is decided by condition. Parameters of its initialization:

- **condition:** Condition shown before as well with all its parameters.
- **fieldNames:** Fields on which the examination must be performed.
- **operation:** What condition exists between the assessments of the fields. **AND** condition is in place by default.

Let's see three examples. In the first example, we want to leave out from the results those cases when the value of the **NAME** field matches with a pattern. During the evaluation a field, the **operation** parameter is not important.

```
- filter: DropByCondition
  condition: IsMatchByRegex
  regexp: '^.*\-.*$'
  fieldNames: NAME
```

In the second example, let's see an other type of assessment. We want to delete the line if

both **EMAIL** and **NAME** values are empty.

```
- filter: DropByCondition
  condition: isEmpty
  fieldNames:
    - NAME
    - EMAIL
  operation: AND
```

In the third example, let's examine the previous one with **OR** operation. In this case, the line will be deleted from the results if either the **NAME** or the **EMAIL** fields are empty.

```
- filter: DropByCondition
  condition: isEmpty
  fieldNames:
    - NAME
    - EMAIL
  operation: OR
```

DropBySource

Inclusion in an other source file decides on the fate of the record. All rules are identical with the ones applicable for DropByCondition, so here the cases will not be described again. Its initialization:

- **condition:** Condition described before with all its parameters. Only conditions with 1 parameter can be used!
- **join:** Name of fields that are joined. They must have the same name in both sources.
- **operation:** What condition exists between the assessments of the fields. **AND** condition is in place by default.

and **source** with all its parameters. The only parameter that belongs to the condition (which is usually **value**) must contain that in which column of the other source the comparison value can be found.

An example of a YAML configuration:

```
- filter: DropBySource
  join: PID
  condition: IsEqual
  value: NAME
  source: Database
  url: 'postgresql://felhasznalonev:jelszo@localhost:5432/adatbazis'
  table: adattabla
  fields:
    - name: PID
      type: Integer
      map: id
    - name: NAME
      type: String
```

DropField

Fields can be dropped from a record with its help. It can happen that a value from a source is pasted into several fields in order to perform different transformations on them, and at the end of the process we want to delete those fields that are not needed anymore. This filter makes this possible.

- **fieldNames:** List of fields to be dropped.

An example of a YAML configuration:

```
- filter: DropField
  fieldNames:
    - name_search
    - district_copy
    - district_search
    - district_roman
    - district_id
    - region_id
```

KeepByCondition

The fate of the record can be decided by condition. Parameters for its initialization as follows:

- **condition:** Condition described before with all its parameters.
- **fieldNames:** On which fields should the examination take place.
- **operation:** What condition exists between the assessments of the fields. **AND** condition is in place by default.

It is almost identical with the **DropByCondition** function, the only difference is that in this case, the record will not be filtered if the condition is met!

Expand

It is used for expansion if we want to add additional values after the current source.

Append

It gives the possibility to read a resource with the same format as of the actual source, and paste it in the actual process. Its parameters:

- **skipIfFails:** If the source is incorrect, the process will not be stopped only the step will be skipped. The list of incorrect sources can be saved with `logFile` and `appendLog`.

Important to note that **it does not extend the previous source with new fields**, everything continues the same way as if the current file would get an other `resource` without `modifier` and `target`. All **resource** attributes can be rewritten, even `username`, `password`, or `encoding` data connected to htaccess!

An example of a YAML configuration:

```
- expand: Append
  resource: target/otherfile.json
  encoding: iso-8859-2
  skipIfFails: true
  logFile: log/otherfile.txt
  appendLog: true
```

AppendAll

It gives the possibility to read a resource with the same format as of the actual source, and paste it in the actual process. Its parameters:

- **folder:** The folder which contains the files that need to be appended to the ending. If the source file is part of the folder as well, it will be ignored.
- **extension:** Only files with extensions will be processed.
- **skipIfFails:** If the source file is incorrect, the process will not be stopped only the step will be skipped. The list of incorrect sources can be saved with `logFile` and `appendLog`
- **skipSubfolders:** It skips the subfolders. They are included by default.

Important to note that **it does not extend the previous source with new fields**, everything continues the same way as if the current file would get an other `resource` without `modifier` and `target`. It creates `Append` for each file and the process will be executed this way.

An example of a YAML configuration:

```
- expand: AppendAll
  folder: source/oc
  extension: xml
```

AppendBySource

The content of an other source can be appended after the original source. Only **source** is needed for initialization with all its parameters.

Important to note that **it does not extend the previous source with new fields**, it pairs everything by name to the data and columns of the original source. The same fields in must have identical name in both sources. Those fields of the current source that do not exist in the original source will not be included among the results!

An example of a YAML configuration:

```
- filter: AppendBySource
  source: Database
  url: 'postgresql://felhasznalonev:jelszo@localhost:5432/adatbazis'
  table: adattabla
  fields:
    - name: PID
      type: Integer
      map: id
    - name: NAME
      type: String
```

Field

It collects columns defined as parameters to an other column including the column values. It can be used if we want to list a few lines of a statistics in key-value form below each other and we want to keep all original columns. Its initialization:

- **fieldNamesAndLabels:** Those fields and their names which we want to contract in two columns.
- **valueFieldName:** The name of the field where the value of the column will be written.
- **labelFieldName:** The name of the field where the name of the value column will be written.

An example of a YAML configuration:

```
- expand: Field
  fieldNamesAndLabels:
    cz: Czech
    hu: Hungary
    sk: Slovak
    pl: Poland
  valueFieldName: count
  labelFieldName: country
```

BaseExpander

A class that can be used for expansion, but it cannot work on its own. It has a task if we want to create more lines from one line in the process. Don't forget the `clone()` method during the prototype query!

```
class ResultExpand( BaseExpanderExpand ):

    def expand( self, record ):

        for phone in record.getField('PHONES').getValue().split(', '):
            fs = self.getFieldSetPrototypeCopy().clone()
            fs.setValues( record.getValues() )
            fs.getField('PHONES').setValue( phone )

        yield fs
```

ListExpander

It breaks up list type elements to separate lines based on their values. It derives from `BaseExpander`, therefore their functioning is quite similar. Its parameters:

- **listFieldName**: The name of the list type element which values need to be broken into separate lines.
- **expandedFieldName**: Where to write the actual value of the list element. Type can be given to it for further type conversion.
- **expanderMap**: It is used when we want to write the list value into several fields. In this case, each field can have a map added to define from where the values should be gathered within the list.

It is important to note, that the two fields can never be identical. If the list element is not needed later, the unnecessary field can be dropped by a `filter` step. Either the `expandedFieldName` or the `expanderMap` is mandatory.

```

source:
  resource: 589739.json
  source: JSON
  fields:
    - name: ID
      type: Integer
      map: response/user/id
    - name: FIRST
      map: response/user/firstName
    - name: LAST
      map: response/user/lastName
    - name: FRIENDS
      map: response/user/friends/groups/0/items/*/id
      type: List
    - name: FRIEND
      type: Integer

manipulations:
  - expand: ListExpander
    listFieldName: FRIENDS
    expandedFieldName: FRIEND
  - filter: DropField
    fieldNames: FRIENDS

target:
  type: JSON
  resource: result.json
  compact: false

```

Melt

It fixes the given columns while the other columns will be shown by key-value pairs. **During the process, all fields that are neither fixed nor contain key-value pairs will be deleted.**

If we do not want to remove the fields just a few fields should be melted, the **Field** expander must be used. Its initialization:

An example of a YAML configuration:

```

- expand: Melt
  fieldNames:
    - first
    - last
  valueFieldName: value
  labelFieldName: quantity

```

Aggregator

It is used to create groups and calculate information from them. Aggregators act many times as Filters or Modifiers as well, since in several cases they delete lines or columns, modify and collect given values. All procedures like this starts with the key word **aggregator**.

Avg

It is used to determine the mean average. Its initialization:

- **fieldNames**: Which fields belong to the group. These values will appear in distinct form in the future!
- **targetFieldName**: Name of the field which will contain the distinct count of the records.
- **valueFieldName**: The value of which fields must be added.
- **listFieldName**: Name of a List field. Matched records will be saved here. It is not mandatory to give.

The aggregator deletes all columns except for **fieldNames**, **targetFieldName**, **listFieldName**.

An example of a YAML configuration:

```
- aggregator: Avg
  fieldNames: author
  targetFieldName: avgprice
  valueFieldName: price
```

Count

Used to calculate figures. Its initialization:

- **fieldNames**: Which fields belong to the group. These values will appear in distinct form in the future!
- **targetFieldName**: Name of the field which will contain the distinct count of the records.
- **listFieldName**: Name of a List field. Matched records will be saved here. It is not mandatory to give.

The aggregator deletes all columns except for **fieldNames**, **targetFieldName**, **listFieldName**.

An example of a YAML configuration:

```
- aggregator: Count
  fieldNames: word
  targetFieldName: count
  listFieldName: matches
```


Sum

It is used to sum values. Its initialization:

- **fieldNames:** Which fields belong to the group. These values will appear in distinct form in the future!
- **targetFieldName:** Name of the field which will contain the distinct count of the records.
- **listFieldName:** Name of a List field. Matched records will be saved here. It is not mandatory to give.

The aggregator deletes all columns except for `fieldNames`, `targetFieldName`, `listFieldName`.

An example of a YAML configuration:

```
- aggregator: Sum
  fieldNames: author
  targetFieldName: sumprice
  valueFieldName: price
```

Target

After the data is read from the source, and the transform and manipulation steps are over, the finalized record gets to the `Target`. This will write and create the file with the final data.

```
target:
  type: <target_type>
...
```

Target is required for every process, and **only one instance of it could exist**. You can continue previous actions when you have used `CSV`, `TSV`, `Database` targets before.

CSV

Target type used in the case of CSV resource. Parameters of its initialization:

- **delimiter:** The sign used for separation in CSV resource. By default `,` is used.
- **quote:** The character used to protect data if the text contains the previously mentioned delimiter. `"` is used by default.
- **addHeader:** It puts the field names in the first line as header.
- **appendFile:** If the target files already exists, should the writing continue or start from the beginning. It always rewrites the files by default.

Further parameters to define the target place:

- **resource:** Target path of the CSV resource, it can be URL as well.
- **encoding:** Coding of the CSV resource. Coding occurs in **UTF-8** by default.

An example of a YAML configuration:

```
target:
  type: CSV
  resource: path/to/the/output.csv
  delimiter: "|"
  addHeader: true
  appendFile: true
```

Database

This target type is used if we want to write our records to a database. Several parameters exist for its initialization:

- **createTable:** If the table does not exist in the database, should it be created or not. It is not created by default, the assumption is that the table already exists with the correct scheme.
- **replaceTable:** Should the table be deleted and re-created either if it exists already in the database or not. It does not delete and replace by default. The usage of this is needed if the table exists already but the new process would expand it with additional columns.
- **truncateTable:** Should the already existing table be cleared in order to write the records to an empty state. It does not clear by default. It is important to note that if the value of the **replaceTable** is true, the table will become empty anyway, so this parameter does not need to be defined in that case.
- **addIDKey:** Should an univoke key with autoincrement sequence be added to the table at the moment of creation. It adds by default.
- **idKeyName:** If the value of the **addIDKey** is true, what the name of that column should be. No columns with this name be among the values to be written.
- **continueOnError:** The line can be skipped if error occurs during the writing or modification (e.g. Foreign Key is not listed). It does not continue by default.

Further parameters to define the target place:

- **url:** Connection link of the database.
- **table:** The name of the table in which we want to write. If the writing/modification is given, the system performs automatically.
- **fn:** The name of the function with which we will write/modify. It must be given in the case when no table is defined. It is worth using if we want to write in several tables in Foreign Key environment. If both **table** and **fn** are defined, then the automatic writing and the function load happen as well.
- **schema:** The name of the scheme in which the table is found. It is not necessary to

define.

An example of a YAML configuration:

```
target:
  type: Database
  url: sqlite:///tests/target
  table: result
  addIDKey: false
  createTable: true
  replaceTable: true
  truncateTable: true
```

or

```
target:
  type: Database
  url: sqlite:///tests/target
  fn: mgmt.RunFunctionQuery
```

with the following Python resource:

```
def RunFunctionQuery( connection, insert_buffer, update_buffer ):

    for item in insert_buffer:
        connection.execute(
            """
                INSERT INTO result ( lat, lng ) VALUES ( :lat, :lng );
            """,
            item
        )

    ...
```

To define **fn** is most useful when creating migrations. Example will be added later.

FixedWidthText

Target type used in the case of fixed width resources. Parameters of its initialization:

- **addHeader**: Should the field names be put in the first line as header. It puts by default.

Further parameters to define the target place:

- **resource**: Target path of the TXT resource, it can be URL as well.

- **encoding:** Coding of the TXT resource. Coding occurs in **UTF-8** by default.

An example of a YAML configuration:

```
target:
  type: FixedWidthText
  resource: utvonal/output.txt
```

GoogleSpreadsheet

Target type used in the case of spreadsheet resources. Parameters of its initialization:

- **addHeader:** Should the field names be put in the first line as header. It puts by default.

Further parameters to define the target place:

- **username:** Name of the user
- **password:** Password of the user
- **spreadsheetKey:** Identifier of the spreadsheet to be written.
- **spreadsheetName:** Name of the spreadsheet to be written. Either spreadsheetKey or spreadsheetName must be given!
- **worksheetName:** Name of the worksheet. If it does not exist, it will be created automatically.
- **truncateSheet:** Should the content of the spreadsheet be cleared. It does not clear by default.

An example of a YAML configuration:

```
target:
  type: GoogleSpreadsheet
  username: ***
  password: ***
  spreadsheetKey: 0ApA_54tZDwKTdDlibXppSkd1MExxb3Y5WmJrZjFxFxR1E
  worksheetName: Sheet1
```

JSON

Target type used in the case of JSON resources. Parameters of its initialization:

- **rootIterator:** The name of the variable in which we want to collect the records. It is not mandatory to be given, the JSON resource will contain only a list if this parameter is left empty.
- **flat:** If only one field exists, this option makes possible to only list the values without field name. It is not used by default.

- **compact:** It generates formatted JSON. Its value is false by default, it generates JSON into one line.

Left empty:

```
[ { ... }, { ... }, ..., { ... } ]
```

Filled in, e.g. with **items**:

```
{ "items": [ { ... }, { ... }, ..., { ... } ] }
```

Further parameters to define the target place:

- **resource:** Target path of the JSON resource. It can be URL as well.
- **encoding:** Coding of the JSON resource. Coding occurs in **UTF-8** by default.

An example of a YAML configuration:

```
target:
  type: JSON
  resource: utvonal/output.json
  rootIterator: items
```

Neo4j

Target type used to write into graph database. Parameters of its initialization:

- **bufferSize:** Size of record to be written at the same time.

Further parameters to define the target place:

- **url:** The address of the Neo4j database
- **resourceType:** The type of data we want to write. Can have **Node** and **Relation** values.
- **label:** Label to be used for the loaded data. It is always mandatory to give, even if for Neo4j it is not necessary.
- **truncateLabel:** To delete the already existing records with the same labels at the beginning of the load. It does not delete by default.

If we choose **Relation** resourceType, the following parameters are mandatory as well:

- **fieldNameLeft:** From the data to be loaded, in which field the identifier describing the left side of the relation is included.
- **fieldNameRight:** From the data to be loaded, in which field the identifier describing

the right side of the relation is included.

- **keyNameLeft**: Which field of the object on the left hand side of the relation includes the key.
- **keyNameRight**: Which field of the object on the right hand side of the relation includes the key.
- **labelLeft**: The label the left hand side element has. Not mandatory.
- **labelRight**: The label the right hand side element has. Not mandatory.

The system automatically places an index to the fields with **key**.

An example for YAML configuration in the case of **Node** :

```
source:
  source: TSV
  resource: Artist.txt
  quote: ""
  skipRows: 1
  fields:
    - name: uid
      map: 0
      key: true
    - name: name
      map: 1
    - name: nationality
      map: 2

target:
  type: Neo4j
  url: http://localhost:7474
  label: Artist
  truncateLabel: true
  resourceType: Node
```

And in the case of **Relation** :

```

source:
  source: TSV
  resource: AlbumArtist.txt
  quote: ""
  skipRows: 1
  fields:
    - name: album_uid
      map: 0
    - name: artist_uid
      map: 1

target:
  type: Neo4j
  url: http://localhost:7474
  label: Contains
  truncateLabel: true
  resourceType: Relation
  fieldNameLeft: album_uid
  fieldNameRight: artist_uid
  keyNameLeft: uid
  keyNameRight: uid
  labelLeft: Album
  labelRight: Artist

```

Static

Type created for testing purposes, it works for `stdout` in TSV format. Parameters of its initialization:

- **silence:** Whether to write to stdout.

An example of a YAML configuration:

```

target:
  type: Static
  silence: false

```

TSV

Target type used in the case of TSV resources. Parameters of its initialization:

- **delimiter:** The sign used for separation in TSV resources. By default `,` is used.
- **quote:** The character used to protect data if the text contains the previously mentioned delimiter. `"` is used by default.
- **addHeader:** Whether to put the field names in the first line as header. It puts by default.

- **appendFile:** If the target files already exists, should the writing continue or start from the beginning. It always rewrites the files by default.

Further parameters to define the target place:

- **resource:** Target path of the TSV resource, it can be URL as well.
- **encoding:** Coding of the TSV resource. Coding occurs in **UTF-8** by default.

An example of a YAML configuration:

```
target:
  type: CSV
  resource: path/to/output.csv
  delimiter: "|"
  addHeader: true
  appendFile: true
```

XLS

Target type used in the case of XLS resources. Parameters of its initialization:

- **addHeader:** Whether to put the field names in the first line as header. It puts by default.

Further parameters to define the target place:

- **resource:** Target path of the XLS resource, it can be URL as well.
- **encoding:** Coding of the XLS resource. Coding occurs in **UTF-8** by default.
- **sheetName:** Name of the worksheet in which we want to write
- **replaceFile:** Should the whole XLS be replaced. If we had an XLS file before with even several worksheets, their values will be lost. It replaces by default.
- **truncateSheet:** Should the worksheet be empty. If the value of the **replaceFile** is true, then this parameter has no importance. But in the other case, it is important to define whether we want to continue writing to the end of worksheet or replace the worksheet to the new data. The worksheet is replaced by default.
- **dynamicSheetField:** If we want to create more worksheets other than the main field by data scattering, then here the field name must be given on the basis of which we want to group the data. The value of the defined field here will be the name of the worksheet. Not necessary.

If we give a non-existing worksheet, the process automatically creates one together with the entire XLS file.

An example of a YAML configuration:


```
target:
  type: XLS
  resource: path/to/output.xls
  sheetName: NotExisting
  replaceFile: false
  truncateSheet: false
```

XML

Target type used in the case of XML resources. Parameters of its initialization:

- **itemName:** Name of an XML element (or the path to that element) which contains one record we need to process. It is mandatory to define!
- **rootIterator:** The name of the root element to which the above XML data should be collected. `root` is used by default. It can't be left empty.
- **flat:** If only one field exists, this option makes possible to only list the values without field name. It is not used by default.

Further parameters to define the target place:

- **resource:** Target path of the XML file, it can be URL as well.
- **encoding:** Coding of the XML file. Coding occurs in `UTF-8` by default.

An example of a YAML configuration:

```
target:
  type: XML
  resource: utvonal/output.xml
  itemName: estate
  rootIterator: estates
```

Yaml

Target type used in the case of Yaml resources. Parameters of its initialization:

- **rootIterator:** The name of the root element to which the data should be collected. It can't be left empty.
- **safe:** Should the indications generated by Python be removed. (e.g. unicode character coding) They are kept by default.
- **flat:** If only one field exists, this option makes possible to only list the values without field name. It is not used by default.

Further parameters to define the target place:

- **resource:** Target path of the YML resource, it can be URL as well.

- **encoding:** Coding of the YML resource. Coding occurs in **UTF-8** by default.

An example of a YAML configuration:

```
target:
  type: Yaml
  resource: utvonal/output.yml
  rootIterator: estates
```

Migrate

During the running of the mETL script, there is a possibility to define a migration file (**-m** parameter) and to generate a new migration file (**-t** parameter). There are two types of migrations, **keyless** and **with key**. If a modification occurs in the configuration file, the earlier migration cannot be used in the future. The two different types of migrations cannot be mixed with each other in any way.

Key, Hash, ID

Each line has the above parameters.

- **Key:** The values of the fields identified as key field separated by **-**. These values clearly identify the record (line). The line does not have a **key** if no fields are labelled.
- **Hash:** Long number and letter row created from the value of an entire record (line) which cannot be decoded.
- **ID:** Merge of the above ones separated by **:**. It clearly identifies the actual status of all values of a line.

Logging

Log variable can be given to a start-up source, manipulation and target file. To activate the logging to a given step, the path of the file must be defined. Each step creates a different logging format, but in general the following applies:

1. **Source log:** It contains the **ID** of the processed line, the dictionary of the processed line as JSON and the value after the transformation in JSON.
2. **Filter modifier log:** It contains the **ID** of the processed line, and the transform value of the dropped line in JSON.
3. **Target file log:** It contains the **ID** of the processed line, the operation (writing or modification) and the value of the line to be written out in JSON.

Modifiers, Expanders, and the transformation steps will not be logged one by one separately.

Migration without key fields

In this case, we can't identify the lines clearly, thus we can't determine whether the value of

the line has changed or not compared to previous status. In this version, none of the fields contain `key` in their configuration, so the only item the migration can define is whether in the previous version (`-m`) there was any identical values based on the `hash`.

Only the already non-existing fields will be written into Target. If a new migration is asked to be generated (`-t`), then the migration file will have all old and new values. Those that are not included in the new file will not be part of the new migration, they will be considered as deleted elements though we do not mark them for deletion anywhere.

Migration with the use of key fields

It is more commonly used, since for almost all sources we can find a combination with which a line can be clearly identified based on several data. During the migration process, for all `ID`s it stores the `hash` belonging to the line. This way, if the hash that belongs to the same `ID` changes, we know exactly which record (line) value has been modified. In the case of text source, all new and modified records get into the target file. But in the case of database target, `UPDATE` commands must be used. Migration to be generated (`-t`) will contain the final status.

List of new/modified/unaltered/deleted element keys

The `metl-differences` script is able to compare migrations. Example can be as follows:

```
metl-differences -d delete.yml migration/migration.pickle  
migration/historical.pickle
```

As it can be seen, it gets a `-d` parameter with the configuration file. It defines where to write the keys of those elements that are to be deleted during the new migration. An example for the `delete.yml` configuration:

```
target:  
  type: JSON  
  resource: migration/migration_delete.json  
  rootIterator: deleted  
  flat: true
```

Only target must be defined, the others are handled by the script. The above generates a list similar to this one:

```
{"deleted":["23105283","23099212","23101411"]}
```

In the original configuration file, one single `id` field contained the `key` setting. **The above script can be used only in the case of identical types of migrations!!!**

But in most cases, the modifications and the list of deleted records are needed due to other reasons. It is common that the new records of the whole migration are written to a database, while the deleted records are to be inactivated. The `fn` attribute of the `DatabaseTarget` is

used for this.

In the case of

```
metl-differences -d delete.yml migration/current.pickle  
migration/prev.pickle
```

the content of delete.yml is:

```
target:  
  type: Database  
  url: sqlite:///database.db  
  fn: mgmt.inactivateRecords
```

while the content of the mgmt.py is:

```
def inactivateRecords( connection, delete_buffer, other_buffer ):  
  
    connection.execute(  
        """  
        UPDATE  
            t_result  
        SET  
            active = FALSE  
        WHERE  
            id = ANY( VALUES %s )  
        """ % ( ', '.join( [ "%(key)s'" % b for b in delete_buffer ] ) )  
    )
```

Example

As we are already familiar with the possibilities the tool can give, let's see what it can be used for.

Loading Spanish data

It is a quite simple load, but can be interesting due to the amount of data. The data is several GBs, it covers almost a decade. On a monthly basis, it has about 10 pieces of 80 MB resources with 250 000 lines for each. The goal is to load the data in a database table in the fastest way.

The following configuration was created for it:

```
source:  
  source: FixedWidthText  
  map:  
    FLOW: '0:1'  
    YEAR: '1:3'
```

MONTH: '3:5'
CUSTOM_ENCLOSURE_RPROVINCE: '5:7'
DATE_OF_ADMISSION_DOCUMENT: '19:25'
POSITION_STATISTICS: '25:37'
DECLARATION_TYPE: '37:38'
ADDITIONAL_CODES: '38:46'
COUNTRY_ORIGIN_DESTINATION: '66:69'
COUNTRY_OF_ORIGIN_ISSUE: '69:72'
PROVINCE_OF_ORIGIN_DESTINATION: '75:77'
CUSTOMS_REGIME_REQUESTED: '82:84'
PRECEDING_CUSTOMES_PROCEDURE: '84:86'
WEIGHT: '89:104'
UNITS: '104:119'
STATISTICAL_VALUE: '119:131'
INVOICE_VALUE: '131:143'
COUNTRY_CURRENCY: '143:146'
CONTAINER: '158:159'
TRANSPORT_SYSTEM: '159:164'
BORDER_TRANSPORT_MODE: '164:165'
INLAND_TRANSPORT_MODE: '165:166'
NATIONALITY_THROUGH_TRANSPORT: '166:169'
ZONE_EXCHANGE: '170:171'
NATURE_OF_TRANSACTION: '172:174'
TERMS_OF_DELIVERY: '174:177'
CONTINGENT: '177:183'
TARIFF_PREFERENCE: '183:189'
FREIGHT: '189:201'
TAX_ADDRESS_PROVINCE: '224:226'

fields:

- name: FLOW
- name: YEAR
type: Integer
- name: MONTH
type: Integer
- name: CUSTOM_ENCLOSURE_RPROVINCE
- name: DATE_OF_ADMISSION_DOCUMENT
type: Date
- name: POSITION_STATISTICS
- name: DECLARATION_TYPE
- name: ADDITIONAL_CODES
- name: COUNTRY_ORIGIN_DESTINATION
- name: COUNTRY_OF_ORIGIN_ISSUE
- name: PROVINCE_OF_ORIGIN_DESTINATION
- name: CUSTOMS_REGIME_REQUESTED
- name: PRECEDING_CUSTOMES_PROCEDURE
- name: WEIGHT
- name: UNITS
- name: STATISTICAL_VALUE
- name: INVOICE_VALUE
- name: COUNTRY_CURRENCY
- name: CONTAINER

- name: TRANSPORT_SYSTEM
- name: BORDER_TRANSPORT_MODE
- name: INLAND_TRANSPORT_MODE
- name: NATURE_OF_TRANSACTION
- name: ZONE_EXCHANGE
- name: NATIONALITY_THROUGH_TRANSPORT
- name: TERMS_OF_DELIVERY
- name: CONTINGENT
- name: TARIFF_PREFERENCE
- name: FREIGHT
- name: TAX_ADDRESS_PROVINCE

target:

type: Database
url: postgresql://metl:metl@localhost:5432/metl
table: spanish_trade
createTable: true
replaceTable: false
truncateTable: false

As it can be seen, there is no `resource` parameter defined in the case of `Source`, since we do not want to create separate configurations for similar file formats. For running, `metl-walk` is used with `multiprocess (-m)` setting, to process the monthly 10 files as soon as possible simultaneously.

```
metl-walk -m spanishtrade.yml data/spanish_trade/2013/jan
```

Aggregated data conversion and collection

Many cases it is needed to create meaningful, clear data from non-reasonable data sources. The below resource formats arrived for each county:

```

{
  "data":[
    {
      "category":"Local business",
      "category_list":[
        {
          "id":"115725465228008",
          "name":"Region"
        },
        {
          "id":"192803624072087",
          "name":"Fast Food Restaurant"
        }
      ],
      "location":{
        "street":"Sz\u00e9chenyi t\u00e9r 1.",
        "city":"P\u00e9cs",
        "state":"",
        "country":"Hungary",
        "zip":"7621",
        "latitude":46.07609661278,
        "longitude":18.228635482364
      },
      "name":"McDonald's P\u00e9cs Sz\u00e9chenyi t\u00e9r",
      "id":"201944486491918"
    },
    ...
  ]
}

```

The goal is to generate a TSV resource that contains all data included in these files.
Configuration used to achieve this:

```

source:
  source: JSON
  fields:
    - name: category
    - name: category_list_id
      map: category_list/0/id
    - name: category_list_name
      map: category_list/0/name
    - name: location_street
      map: location/street
    - name: location_city
      map: location/city
    - name: location_state
      map: location/state
    - name: location_country
      map: location/country
    - name: location_zip
      map: location/zip
      type: Integer
    - name: location_latitude
      map: location/latitude
      type: Float
    - name: location_longitude
      map: location/longitude
      type: Float
    - name: name
    - name: id
  rootIterator: data

target:
  type: TSV
  resource: common.tsv
  appendFile: true

```

The program was running with the below format: `metl-walk config.yml data/`

Long format conversion from table form

By using Field expander

We have a TSV resource with the following format:

```

Year    CZ  HU  SK  PL
1999    32  694 129 230
1999    395 392 297 453
1999    635 812 115 97
...

```


To which we create the below configuration:

```
source:
  source: TSV
  resource: input1.csv
  skipRows: 1
  fields:
    - name: year
      type: Integer
      map: 0
    - name: country
    - name: count
      type: Integer
    - name: cz
      type: Integer
      map: 1
    - name: hu
      type: Integer
      map: 2
    - name: sk
      type: Integer
      map: 3
    - name: pl
      type: Integer
      map: 4

manipulations:
  - expand: Field
    fieldNamesAndLabels:
      cz: Czech
      hu: Hungary
      sk: Slovak
      pl: Poland
    valueFieldName: count
    labelFieldName: country
  - filter: DropField
    fieldNames:
      - cz
      - hu
      - sk
      - pl

target:
  type: TSV
  resource: output1.csv
```

Thus we get the following result:

year	country	count
1999	Slovak	129
1999	Czech	32
1999	Poland	230
1999	Hungary	694
1999	Slovak	297
1999	Czech	395

By using Melt expander

Let's see the following input file:

first	height	last	weight	iq
John	5.5	Doe	130	102
Mary	6.0	Bo	150	98

An example configuration file to get the data to long value based on the key-value pairs:

```

source:
  source: TSV
  resource: input2.csv
  skipRows: 1
  fields:
    - name: first
      map: 0
    - name: height
      type: Float
      map: 1
    - name: last
      map: 2
    - name: weight
      type: Integer
      map: 3
    - name: iq
      type: Integer
      map: 4
    - name: quantity
    - name: value

manipulations:
  - expand: Melt
    fieldNames:
      - first
      - last
    valueFieldName: value
    labelFieldName: quantity

target:
  type: TSV
  resource: output2.csv

```

As a result, the below will be created:

first	last	quantity	value
John	Doe	iq	102
John	Doe	weight	130
John	Doe	height	5.5
Mary	Bo	iq	98
Mary	Bo	weight	150
Mary	Bo	height	6.0

Data load to two tables of a database

Let's see a complex example which is based on the usage of the ListExpander.

```
source:
  source: JSON
  rootIterator: features
  resource: hucitystreet.geojson
  fields:
    - name: id
      type: Integer
      map: id
      key: true
    - name: osm_id
      type: Float
      map: properties/osm_id
    - name: name
      map: properties/name
    - name: ref
      map: properties/ref
    - name: type
      map: properties/type
    - name: oneway
      type: Boolean
      map: properties/oneway
    - name: bridge
      type: Boolean
      map: properties/bridge
    - name: tunnel
      type: Boolean
      map: properties/tunnel
    - name: maxspeed
      map: properties/maxspeed
    - name: telkod
      map: properties/TEL_KOD
    - name: telnev
      map: properties/TEL_NEV
    - name: kistkod
      map: properties/KIST_KOD
    - name: kistnev
      map: properties/KIST_NEV
    - name: megynev
      map: properties/MEGY_NEV
    - name: regnev
      map: properties/REG_NEV
    - name: regkod
      map: properties/REG_KOD
    - name: geometry
      type: List
      map: geometry/coordinates

target:
  type: Database
  url: postgresql://metl:metl@localhost:5432/metl
```

```
table: osm_streets
createTable: true
replaceTable: true
truncateTable: true
addIDKey: false
```

In the database, the value of the `geometry` field will be `JSON`. We want to break up this list to an other table as `latitude` and `longitude` coordinates. Currently, the following values are in the `geometry` field:

```
[[17.6874552,46.7871465],[17.6865955,46.7870049],[17.6846158,46.7866786],
[17.6834977,46.7864944],[17.6822251,46.7862847],[17.6815319,46.7861705],
[17.6811473,46.7861071],[17.6795989,46.785852],[17.6774482,46.7854976],
[17.6739061,46.7849139],[17.6729351,46.7847539],[17.6720789,46.7846318]]
```

We want to achieve this by the below configuration:

```

source:
  source: Database
  url: postgresql://metl:metl@localhost:5432/metl
  table: osm_streets
  fields:
    - name: street_id
      type: Integer
      map: id
    - name: latitude
      type: Float
    - name: longitude
      type: Float
    - name: geometry
      type: List
      map: geometry

manipulations:
  - expand: ListExpander
    listFieldName: geometry
    expanderMap:
      latitude: 0
      longitude: 1
  - filter: DropField
    fieldNames: geometry

target:
  type: Database
  url: postgresql://metl:metl@localhost:5432/metl
  table: osm_coords
  createTable: true
  replaceTable: true
  truncateTable: true

```

We read out here the previously loaded value of the `geometry` field into a list, then with the help of the `ListExpander` we define what to write exactly in the `latitude` and `longitude` fields. With this, we created a table and a connection table belonging to it.

Database transfer

We have a MySQL and a PostgreSQL database and we want to switch between the two. Data can be transferred easily through a command:

```
metl-transfer config.yml
```

The configuration file is the following:

```
sourceURI: mysql:mysqlconnector://xyz:xyz@localhost/database
targetURI: postgresql://xyz:xyz@localhost:5432/database
```

```
tables:
```

- ['Message', 'message']
- ['SourceMessage', 'sourcemessage']
- related_content
- poi
- shorturl
- ident_data
- user
- estate_agency
- time_series
- auth_item
- property_migration
- property_group
- cemp_id_daily
- cluster
- auth_assignment
- property
- lead
- pic
- lead_comment
- similarity
- property_cluster

```
truncate:
```

- auth_item
- estate_agency

```
runAfter: |
```

```
UPDATE
```

```
property
```

```
SET
```

```
status = status + 1,
condition = condition + 1,
estatetype = estatetype + 1,
heating = heating + 1,
conveniences = conveniences + 1,
parking = parking + 1,
view = view + 1,
material = material + 1;
```

`sourceURI` contains the address of the source database, while `targetURI` contains the address of the target database. Listing of `tables` is not mandatory, if they are not listed, then all of the tables from the source database will be copied to the target database. With the truncate option, given tables can be cleared in the target database before loading, while SQL ccommands can be run with `runAfter` and `runBefore`

Important to note that the tables must exist in the target database, the transfer does not create them.