data-migrator Documentation

Release 0.4.5

Ilja Heitlager

Apr 14, 2017

Contents:

data-migrator is database transformation in a pipes and filter, declarative kind of way, Python/Django style.

Data transformation is an important process in software development when architectures upgrade and data has to move along. This requires fast and easy restructuring of data in a repetitive way. Quite quickly ad-hoc scripts or transfer data, upgrade it in sql second is not flexible enough. For those cases *data-migrator* is an answer.

Once you 've *installed* *data-migrator*, we recommend reading the

- *Introduction* first,
- Take a look at the *Example*
- after that continue with the *Tutorial*

CHAPTER 1

Contents

Introduction

Data transformation is a classic problem in compute, but underestimated in modern software development. Everybody working with persistent data will be involved somehow in restructuring existing data in databases or files while systems evolve. A wide range of practices exist ranging from ad-hoc scripts to sophisticated ETL processes. When we upgraded an existing modules at Schuberg Philis moving from a mono-lithical application to a microservice architecture, we found ourselves in a position to write some ad-hoc python scripts. Table by table the transformation was done, simply by exporting existing in csv's and with some simple python scripts - a single read/print loop - generate new INSERT statements.

How hard can it be, just some basic processing and statement emitting. But soon we found ourselves cleaning/fixing data, generating multiple records out of single rows. The scripts became unreadable and hard to maintain. That is when we came up with the idea of applying a more declarative approach. And we were pretty charmed by the Django model approach. Soon after a standardized system based on a declarative definitions originated.

This package is a simple alternative to doing ad-hoc scripts. It is easy to learn, easy to extend and highly expressive in building somewhat more complex transformation. Think of for example:

- renaming, reordering columns
- changing types
- lookup data, advanced transformations
- generating permission records in separate tables for main data

Now see the *example* and move on to the *installation*

Example

Core of data-migrator is the unix pipe and filter paradigm to build data transformers. Source data is read from the database or some other source. It is piped to a filter written in data-migrator which emits for example SQL insert statements, after which this can be piped to a target client.

The most simple single datapump in mysql for example is written like:

```
$ mysqldump -u [uname] -p[pass] source_db table  | mysql target_db
```

In this case mysqldump will export the table as SQL statements and the new database will process them. Now if you want to do something extra and repeatable with respect to the data, you could use all kinds of unix filtering with sed, awk, or other favorite poison. Hard to imagine what Pythonista's would do especially if extra columns or something are needed. The basic packages are quite strong and one would setup something like:

```
$ mysql source_db -E 'select * from table' -B  | python my_filter.py | mysql target_db
```

With *my_filter.py* written as something like:

```
import sys, csv

reader = csv.DictReader(sys.stdin)

for row in reader:
        print 'INSERT INTO `table` (a,b) VALUES ("%(a)s", %(b)s)' % row
```

To see the options for manipulation is left as an exercise to the reader, but do accept that as soon things become just a little more complex (think: splitting in two tables, column reverses, renaming of columns, mixing, joining, filtering, transforming), more declarative support is helpful. That is why we came up with *data-migrator*. One could simply replace this with:

```
from data_migrator import models, transform
from data_migrator.emitters import MySQLEmitter

def parse_b(v):
        if v == 'B':
                return 'transformed_B'
        else:
                return v.lower()

class Result(models.Model):
        id   = models.IntField(pos=0) # keep id
        uuid = models.UUIDField()     # generate new uuid4 field
        # replace NULLs and trim
        a    = models.StringField(pos=1, default='NO_NULL', max_length=5, null='NULL',
↪ replace=lambda x:x.upper())
        # parse this field
        b    = models.StringField(pos=2, parse=parse_b, name='my_b')

        class Meta:
                table_name = 'new_table_name'

# django-esc like creating and saving (to a manager)
Result(a='my a', b='my b').save()

if __name__ == "__main__":
        transform.Transformer(models=[Result], emitter=MySQLEmitter).process()

        assert(len(Result.objects) > 1)
```

And have a nice self explaining transformer which will generate something like:

```
-- transformation for Result to table new_table_name
-- input headers: id,a,b
```

```
-- stats: in=10,dropped=0,out=10

SET SQL_SAFE_UPDATES = 0; -- you need this to delete without WHERE clause
DELETE FROM `new_table_name`;
ALTER TABLE `new_table_name` AUTO_INCREMENT = 1;

INSERT INTO `new_table_name` (`id`, `uuid`, `a`, `my_b`) VALUES (0, "ac7100b9-c9ad-
↪4069-8ca5-8db1ebd36fa3", "MY A", "my b");
INSERT INTO `new_table_name` (`id`, `uuid`, `a`, `my_b`) VALUES (1, "38211712-0eb2-
↪4433-b28f-e3fe33492e7a", "NO_NULL", "some value");
INSERT INTO `new_table_name` (`id`, `uuid`, `a`, `my_b`) VALUES (2, "a3478903-aed9-
↪462c-8f47-7a89013bc6ea", "CHOPP", "transformed_B");
```

Installation

Using pip (or ...)

Category Stable version

Precondition pip (or setuptools) is installed

Execute the following command to install data-migrator with pip:

```
pip install data-migrator
```

To update an already installed data-migrator version, use:

```
pip install -U data-migrator
```

As an alternative, you can also use easy_install to install data-migrator:

```
easy_install data-migrator          # CASE: New installation.
easy_install -U data-migrator       # CASE: Upgrade existing installation.
```

Hint: See also pip related information for installing Python packages.

Using a Source Distribution

After unpacking the data-migrator source distribution, enter the newly created directory "data-migrator-<version>" and run:

```
python setup.py install
```

Using the Github Repository

Category Bleading edge

Precondition pip is installed

Run the following command to install the newest version from the Github repository:

```
pip install git+https://github.com/schubergphilis/data-migrator
```

To install a tagged version from the Github repository, use:

```
pip install git+https://github.com/schubergphilis/data-migrator@<tag>
```

where <tag> is the placeholder for an existing tag.

When running from the repository it is adviced to run in developer mode:

```
pip install -e .
```

This allows to work on the code while using the library. Be sure to make that pull-request ;-)

Tutorial

Getting Started

First, install *data-migrator*.

Now create a new directory with your migration scripts. Your milage may very, built for here we assume you have client access to source data, spitting out a csv in some form and client access to a target database. To automate and make it repetitive, just use `make`, we add some Makefile-foo here but do not worry:

```
TARGETS = table
OPTIONS ?=-p 2 --debug
OUT_DIR ?= results

TABLE_QRY='SELECT t.* FROM table LIMIT 0,100'

default: clean install all

all: $(TARGETS)

install:
      pip install data-migrator

clean:
      @rm -rf $(OUT_DIR)
      @find . -name *.pyc -delete

$(OUT_DIR)/%.sql: | $(OUT_DIR)
      ssh [SOURCE_HOST] "sudo mysql connect -e $($(@F)) -B" | python transform_$*.py
↪$(OPTIONS) -o $(OUT_DIR)

$(TARGETS):%:$(OUT_DIR)/%.sql

$(OUT_DIR):
      mkdir -p $@

upload:
      ssh [TARGET_HOST] "sudo mysql [TARGET_DB]" < $(OUT_DIR)/table.sql
```

See that we use a simple query and extract the first 100 lines. The rest of the magic of the `Makefile` is to separate the extraction from the loading, and allow to easily extend the script with more tables and source. Note that in this

case we are defining the extract query in the makefile, and we are using sudo rights to extract and upload the data. You might want to have an opinion about that.

We now have the ground work for extracting a table, transforming it and loading it. Next step is to build the filter and transform the data into something the target database can have. Going back to the example we build a simple transformer:

```
from data_migrator import models, transform
from data_migrator.emitters import MySQLEmitter

def parse_b(v):
        if v == 'B':
                return 'transformed_B'
        else:
                return v.lower()

class Result(models.Model):
        id   = models.IntField(pos=0) # keep id
        uuid = models.UUIDField()     # generate new uuid4 field
        # replace NULLs and trim
        a    = models.StringField(pos=1, default='NO_NULL', max_length=5, null='NULL',
↪ replace=lambda x:x.upper())
        # parse this field
        b    = models.StringField(pos=2, parse=parse_b, name='my_b')

        class Meta:
                table_name = 'new_table_name'

# django-esc like creating and saving (to a manager)
Result(a='my a', b='my b').save()

if __name__ == "__main__":
        transform.Transformer(models=[Result], emitter=MySQLEmitter).process()

        assert(len(Result.objects) > 1)
```

And have a nice self explaining transformer which will generate something like:

```
-- transformation for Result to table new_table_name
-- input headers: id,a,b
-- stats: in=10,dropped=0,out=10

SET SQL_SAFE_UPDATES = 0; -- you need this to delete without WHERE clause
DELETE FROM `new_table_name`;
ALTER TABLE `new_table_name` AUTO_INCREMENT = 1;

INSERT INTO `new_table_name` (`id`, `uuid`, `a`, `my_b`) VALUES (0, "ac7100b9-c9ad-
↪4069-8ca5-8db1ebd36fa3", "MY A", "my b");
INSERT INTO `new_table_name` (`id`, `uuid`, `a`, `my_b`) VALUES (1, "38211712-0eb2-
↪4433-b28f-e3fe33492e7a", "NO_NULL", "some value");
INSERT INTO `new_table_name` (`id`, `uuid`, `a`, `my_b`) VALUES (2, "a3478903-aed9-
↪462c-8f47-7a89013bc6ea", "CHOPP", "transformed_B");
```

There you are, you have setup your first pipeline. Execute this by running:

```
$ make table  # extract the data from the database, transform it
$ make upload # load it into the database
```

You can lookup the intermediate result by viewing the generated sql `results/new_table_name.sql`. data-

migrator does not focus on the database schema (yet!) so the table is expected to exist in the target system. By default the system is wiping the data, not recreating the table. If you have issues with the python libraries, run `make install` do install the library from this makefile.

Now go ahead and add more fields. See *fields reference* for more details about the options of the fields.

Concepts

The Scan Emit Loop

Core in the data-migrator is the declarative definition of the target model. Indeed in a django-esc way. Columns of the target table are defined as fields and each field has many settings. The Field is a definition of what to perform scanning, transforming and emitting the record. Output is abstracted into an extensible set of output writers. The whole is controlled with a standard transformer engine.

The scan-emit loop is the basis the data-migrator. Once everything is setup, by default the transformer will read stdin and send every CSV row to the model for scanning. Out of the box the fields define a scan loop:

1. **select** the specified column from the row.
2. **null** test if not allowed and replace by default.
3. **validate** the input (if validator is provided).
4. **parse** the input (if parser is provided).
5. **store** as native python value (aka NULL=>None).

Once all fields are parsed, the resulting object can be checked for *None* or uniqueness. It can be dropped or the filter can fail because of violations. This are all declarative settings on the Model through the Meta settings. Otherwise the record is saved and (accessible by *Model.objects.all()*) is emitted. This is based on a dedicated emitter, like the MySQL *INSERT* statement generator. Emitting provides some of the following features:

1. **trim** if string and max_length is set (note the full string is stored in the intermediate object!).
2. **validate** the output (if output_validate is provided).
3. **replace** the value with some output string (if provided).
4. **write** in a dedicated format as dictated by the emitter.

API Reference

Model class reference

This document covers features of the `Model` class.

Attributes

objects

`Model.`**`objects`**

Each non-abstract `Model` class must have a `Manager` instance added to it. Data-migrator ensures that in your model class you have at least a default `SimpleManager` specified. If you don't add your own `Manager`,

Django will add an attribute `objects` containing default `SimpleManager` instance. If you add your own `Manager` instance attribute, the default one does not appear.

Methods

scan(row)

`Model.`**`scan`**`(`*row*`)`

Take a row and set values based on the field definitions. All fields in the field definitions are parsed. If field index does not exist an IndexError will be raised.

Returns `self` so it can be chained

save()

`Model.`**`save`**`()`

Save this object and add it to the list.

Returns `self` so it can be chained

emit(escaper)

`Model.`**`emit`**`(`*escaper=None*`)`

Emit the existing object, apply all field translations. Might raise exceptions due to validations.

Returns a dict with the translated values

Meta class reference

This document covers features of the `Meta` class. The meta class defines model specific settings.

Note: Technically, Meta is just a container and forwarded to `data-migrator.models.options.Options`

Field options

The following arguments are available to all field types. All are optional.

drop_if_none

`Meta.`**`drop_if_none`**

Is a list of field names as defined. If set *data-migrator* will check if fields are not None and drop if one of the columns is.

Any field listed in this attribute is checked after scanning and just before save-ing.

Note: Note that only NullXXXFields actually can be `None` after scanning and parsing. Non Null fields are set to their default value.

drop_non_unique

`Meta.`**drop_non_unique**

If `True`, *data-migrator* will drop values if the column uniqueness check fails (after parsing). Default is `False`.

Any field can be defined as a unique column. Any field set so, is checked after scanning and just before save-ing.

emitter

`Meta.`**emitter**

If set, *data-migrator* will use this emitter instead of the default emitter.

fail_non_unique

`Meta.`**fail_non_unique**

If `True`, *data-migrator* will fail as a whole if the column uniqueness check fails (after parsing). Default is `False`.

Any field can be defined as a unique column. Any field set so, is checked after scanning and just before save-ing.

fail_non_validated

`Meta.`**fail_non_validated**

If `True`, *data-migrator* will fail as a whole if the column validation check fails (after parsing). Default is `False`.

Any field can have its own validator, this is a rough method to prevent bad data from being transformed and loaded.

file_name

`Meta.`**file_name**

If set, *data-migrator* will use this as file_name for the emitter instead of the default filename based on table_name.

table_name

`Meta.`**table_name**

If set, *data-migrator* will use this as table_name for the emitter instead of the default tablename based on model_name.

prefix

Meta.**prefix**

If set, *data-migrator* will use this list of statements as a preamble in the generation of the output statements. By default an emitter uses this to clear the old records.

remark

Meta.**remark**

If set, *data-migrator* will use this as the remark attribute in the Model, default='remark'. Use this for example if you have a `remark` field in your model and need to free the keyword.

manager

Meta.**manager**

If set, *data-migrator* will use this as the manager for this model. This is useful if the `transform` method needs to be overridden.

Model field reference

This document contains all API references of :class: *Field* including the *field options* and *field types* data-migrator offers.

Note: Technically, these models are defined in `data-migrator.models.fields`, but for convenience they're imported into `data-migrator.models`; the standard convention is to use `from data-migrator import models` and refer to fields as `models.<Foo>Field`

Field options

The following arguments are available to all field types. All are optional.

pos

Field.**pos**

If positive or zero this denotes the column in the source data to select and store in this field. If not set (or negative) the fields is interpreted as not selecting just a column from the source but to take the full row in the parse function

name

Field.**name**

The name of this field. By default this is the name provided in the model declaration. This attribute is to replace that name by the final column name.

default

Field.**default**

The default value to use if the source column is found to be a `null` field or if the parse function returns None. This attribute has default values for Fields that are not Null<xxx>Fields. For example NullStringField has both NULL and empty string as empty value. StringField only has empty string as empty value. With this field it can be changed to some other standard value. Consider a Country field as string and setting it to the home country by default.

null

Field.**null**

If set it will match the source column value and consider this a `None` value. By default this attribute is set to `None`. Note that for none Null fields `None` will be translated to *`default`*

replace

Field.**replace**

If set this is a pre-emit replacement function. This for example could be used to insert replacement lookup select queries. Adding more indirection into the data generation.

replace

Field.**replace**

If set this is a pre-emit replacement function. This for example could be used to insert replacement lookup select queries. Adding more indirection into the data generation.

parse

Field.**parse**

If set this is the parsing function to replace the read value into something to use further down the data migration. Use this for example to clean phonenumbers, translate country definitions into alpha3 codes, or to translate ID's into values based on a separately loaded lookup table.

validate

Field.**validate**

Expects a function that returns a boolean, and used to validate the input data. Expecting data within a range or a specific format, add a column validator here. Raises *ValidationException* if set and false.

max_length

Field.**max_length**

In case of StringFields use this to trim string values to maximum length.

unique

Field.**unique**

If True, *data-migrator* will check uniqueness of intermediate values (after parsing). Default is False.

In relationship with the default manager this will keep track of values for this field. The manager can raise exceptions if uniqueness is violated. Note that it is up to the manager to either fail or drop the record if the exception is raised.

Note: Use this with HiddenField and a row parse function if some combination of fields (aka a compound key) is expected to be unique and not to be violated.

---

validate_output

Field.**validate_output**

A pre-emit validator used to scan the bare output and raise exceptions if output is not as expected

creation_order

Field.**creation_order**

An automatically generated attribute used to determine order of specification, and used in the emitting of dataset

Field types

BooleanField

class data-migrator.models.**BooleanField**(***options*)

a bool that takes any cased permutation of true, yes, 1 and translates this into True or False otherwise.

IntField

class data-migrator.models.**IntField**(***options*)

a field that accepts the column to be integer.

NullIntField

class data-migrator.models.**NullIntField**(***options*)

a field that accepts the column to be integer and can also be None, which is not the same as 0 (zero).

HiddenField

class data-migrator.models.**HiddenField**(***options*)

a field that accepts, but does not emit. It is useful for uniqueness checked and more. Combine this with a row parse and check the complete row.

StringField

class data-migrator.models.**StringField**(***options*)

a field that accepts the column to be string.

NullStringField

class data-migrator.models.**NullStringField**(***options*)

a field that accepts the column to be string and can also be None, which is not the same as empty string (“”).

UUIDField

class data-migrator.models.**UUIDField**(***options*)

a field that generates a `str(uuid.uuid4())`.

NullField

class data-migrator.models.**NullField**(***options*)

a field that generates None.

JSONField

class data-migrator.models.**JSONField**(***options*)

a field that takes the values and spits out a JSON encoding string. Great for maps and lists to be stored in a string like field.

MappingField

class data-migrator.models.**MappingField**(*data_map={}*, *as_json=False*, ***options*)

a field that takes the values translates these according to a map. Great for identity column replacements. If needed output can be translated as `json`, for example if the map returns lists.

MappingField has two extra arguments:

MappingField.**data_map**
: The data_map needed to translate. Note the fields returns *default* if it is not able to map the key

MappingField.**as_json**
: If `True`, the field will be output as json encoded. Default is `False`.

Contributor Code of Conduct

As contributors and maintainers of these projects, and in the interest of fostering an open and welcoming community, we pledge to respect all people who contribute through reporting issues, posting feature requests, updating documentation, submitting pull requests or patches, and other activities.

We are committed to making participation in these projects a harassment-free experience for everyone, regardless of level of experience, gender, gender identity and expression, sexual orientation, disability, personal appearance, body size, race, ethnicity, age, religion, or nationality.

Examples of unacceptable behavior by participants include:

- The use of sexualized language or imagery
- Personal attacks
- Trolling or insulting/derogatory comments
- Public or private harassment
- Publishing other's private information, such as physical or electronic addresses, without explicit permission
- Other unethical or unprofessional conduct.

Project maintainers have the right and responsibility to remove, edit, or reject comments, commits, code, wiki edits, issues, and other contributions that are not aligned to this Code of Conduct. By adopting this Code of Conduct, project maintainers commit themselves to fairly and consistently applying these principles to every aspect of managing this project. Project maintainers who do not follow or enforce the Code of Conduct may be permanently removed from the project team.

This code of conduct applies both within project spaces and in public spaces when an individual is representing the project or its community.

Instances of abusive, harassing, or otherwise unacceptable behavior may be reported by opening an issue or contacting one or more of the project maintainers.

This Code of Conduct is copied from PyPA. and is adapted from the Contributor Covenant, version 1.2.0 available at http://contributor-covenant.org/version/1/2/0/.

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Index