mETL dokumentáció

The basics

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

Képességek

Leggyakoribb fájlformátumokra biztosít a program aktuális verziója támogatást **migrációk és migrációs csomagok** kezelésével. Ezek a következőek típusonként.

Forrás típusok:

- CSV, TSV, XLS, Google SpreadSheet, Fix szélességű fájl
- PostgreSQL, MySQL, Oracle, SQLite, Microsoft SQL Server
- JSON, XML, YAML

Cél típusok:

- CSV, TSV, XLS, Google SpreadSheet - fájl folytatással is
- Fix szélességű fájl
- PostgreSQL, MySQL, Oracle, SQLite, Microsoft SQL Server - módosítás céllal is
- JSON, XML, YAML

Fejlesztés folyamán igyekeztünk a leggyakoribb transzformációs lépésekkel, programszerkezetekkel, és manipulációs lépésekkel is ellátni a teljes feldolgozási folyamatot. Ennek fényében a program a következő transzformációkkal rendelkezik alapértelmezetten:

- **Add**: Hozzáad egy tetszőleges számot egy értékhez.
- **Clean**: Eltávolítja a különféle írásjeleket. (pont, vessző, stb.)
- **ConvertType**: Módosítja a mező típusát egy másik típusra.
- **Homogenize**: Az ékezetes karaktereket ékezet nélküliekre alakítja. (NFKD formátum)
- **LowerCase**: Kisbetűssé alakítás.
- **Map**: Kicserél mező értékeket, más értékre.
- **RemoveWordsBySource**: Egy másik forrás állományt felhasználva eltávolít szavakat.
- **ReplaceByRegexp**: Reguláris kifejezés alapján cserét hajt végre.
- **ReplaceWordsBySource**: Egy másik forrás állományt felhasználva lecserél szavakat.
- **Set**: Érték beállítást végez.
- **Split**: Szóközök mentén elválasztja a szavakat és a megadott intervallumot hagyja meg.

- **Stem**: Szótőre hozás.
- **Strip**: Eltávolítja az érték elején és végén levő felesleges szóközöket vagy egyéb karaktereket.
- **Sub**: Kivon egy számot egy értékből.
- **Title**: Minden szót nagy kezdőbetűssé alakít.
- **UpperCase**: Nagybetűssé alakítás.

Manupulációk esetében négy csoportot különböztetünk meg:

1. **Modifier**

   Módosítók azok az objektumok, amelyek egy teljes sort (rekordot) kapnak, és mindig egy teljes sorral térnek vissza. Azonban a folyamataik során érték módosításokat végeznek a különböző mezők összefüggő értékeinek felhasználásával.

   - **JoinByKey**: Két forrást a belső forrás kulcsai alapján összefűz, és kitölti a kért mezőket.
   - **Order**: Sorrendbe rendezi a sorokat a megadott feltételeknek megfelelően.
   - **Set**: Érték beállítást végez fix érték séma, függvény, vagy másik forrás felhasználásával.
   - **SetWithMap**: Érték beállítást végez összetett típus esetén megadott map segítségével.
   - **TransformField**: Hagyományos mező szintű transzformáció hívható általa a manipulációs lépés során.
2. **Filter**

   Szűrést végeznek elsősorban. Olyankor használatosak, amikor a korábbi lépésekben transzformációk segítségével megtisztított értékeket szeretnénk kiértékelni és eldobni, ha a rekordot hiányosnak, vagy nem megfelelőnek ítéljük meg.

   - **DropByCondition**: Feltétel alapján dönthető el a rekord sorsa.
   - **DropBySource**: Másik forrás állományban történő szereplés dönt a rekord sorsáról.
   - **DropField**: Rekord számot ugyan nem csökkent, de mezők törölhetőek a segítségével.
   - **KeepByCondition**: Feltétel alapján dönthető el a rekord sorsa.
3. **Expand**

   Bővítésre használjuk, ha további értékeket szeretnénk a jelenlegi forrás után helyezni.

   - **Append**: Mostanival teljesen megegyező forrásállomány beszúrása a folyamatba az aktuális után.
   - **AppendAll**: Mappán szalad végig és a teljesen megegyező forrásállományokat beszűrja a folyamatba.
   - **AppendBySource**: Másik forrás állomány tartalma szúrható az eredeti forrás után.
   - **Field**: Paraméterül megadott oszlopokat egy másik oszlopba gyűjt össze az oszlop értékeivel.

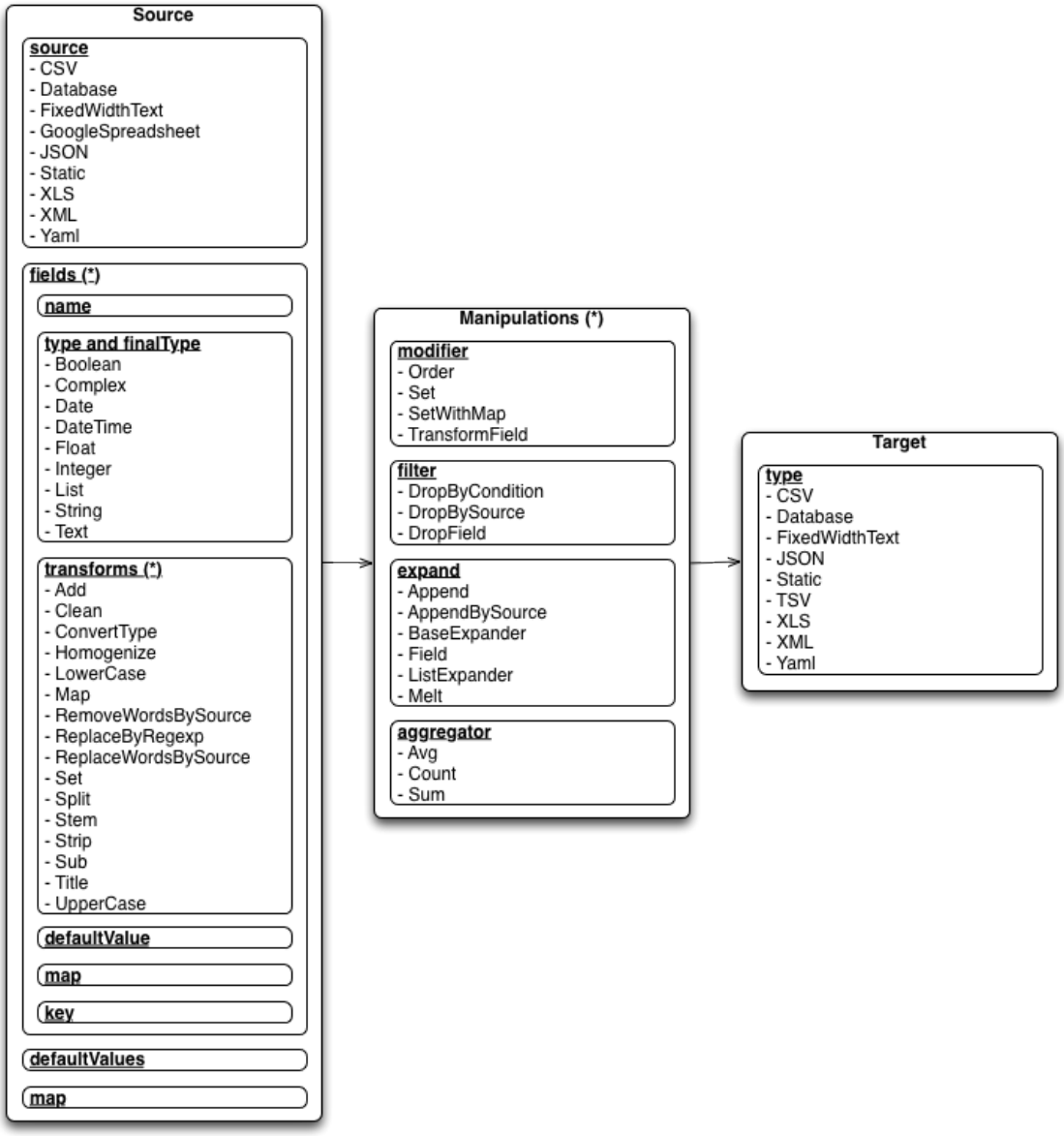
- **BaseExpander**: Kiterjesztésre használható osztály, elsődleges feladata olyan esetben van, ahol egy rekordot többszöröznénk meg.
- **ListExpander**: Lista típusú elemet bont értékei alapján külön sorokba.
- **Melt**: Megadott oszlopokat rögzíti és a többi oszlopot kulcs-érték párok alapján jeleníti meg.

4. **Aggregator**

   Adatok összekapcsolására és csoportokba rendezésére használjuk.

   - **Avg**: Átlag érték meghatározásához használható.
   - **Count**: Számosságok kalkulációjához használjuk.
   - **Sum**: Összérték meghatározásához használható.

Komponens ábra



Telepítés

Hagyományos Python csomagként a telepítést legegyszerűbben a következő parancs kiadásával tehetjük meg a mETL könyvtárában állva: `python setup.py install`

Csomagot ezt követően az alábbi paranccsal tesztelhetjük: `python setup.py test`

Következő függőségekkel rendelkezik: xlrd, gdata, demjson, pyyaml, sqlalchemy, xlwt, tarr, nltk, xlutils, xmlsquash

Mac OSX

Telepítés előtt a következő csomagok feltételére lesz szükség, mely a következő:

- XCode telepítése

- XCode "Command Line Tools" telepítése
- Macports telepítése

Ezt követően minden csomag megfelelően feltelepítésre kerül.

Linux

Telepítés előtt a `python-setuptools` meglétét ellenőrizni kell, illetve hiányzása esetén `apt-get install`-al telepíteni.

Windows

Minden csomag könnyedén feltelepíthető!

Futtatás

Konzol scriptek gyűjteménye a program, amely emiatt bármilyen rendszerbe könnyen beépíthető, és akár cron script-ek segítségével időzíthető is.

Következő script-ekből áll a program:

1. `metl [options] CONFIG.YML`

   Egy teljes folyamat indítható el a segítségével a paraméterül kapott YAML fájl alapján. A konfigurációban megadott folyamatokat a konfigurációs állománynak teljesen le kell írnia, input és output fájlok pontos útvonalával együtt.

   - `-t`: Futtatás során migrációs állomány készítése az aktuális adatok állapotából.
   - `-m`: Korábbi migrációs állomány átadása, amely az előző futtatott verzióé volt.
   - `-p`: Mappa átadása, amit hozzá adunk a PATH változóhoz, hogy a YAML konfigurációban történő hivatkozás megvalósulhasson külső python állományra.
   - `-d`: Debug mód, mindent kiír a stdout-ra.
   - `-l`: Hány elemen történjen a feldolgozás. Nagyszerű lehetőség nagy fájlok tesztelésére kis rekordokon, amíg minden nem úgy működik, ahogy szeretnénk.
   - `-o`: Hányadik elemtől kezdje a feldolgozást.
   - `-s`: Ha a konfigurációs állomány nem tartalmazza a resource útvonalát, itt is megadható.

   Migrációról és a `-p` kapcsolóról később lesz szó részletesebben.

2. `metl-walk [options] BASECONFIG.YML FOLDER`

   Feladata a paraméterül kapott YAML fájl alkalmazása, minden paraméterül kapott mappában szereplő állományra nézve. A konfigurációnak ebben az esetben nem kell az input fájlok elérhetőségét tartalmazni, a script automatikusan elvégzi ezek behelyettesítését.

   - `-m`: Multiprocessing bekapcsolása több CPU-val rendelkező gépeken. A

feldolgozandó állományok külön thread-ekbe kerülnek. Használata **csak és kizárólag** `Database` cél esetén szabad, mindenhol máshol problémákat okoz!
    - `-p` : Mappa átadása, amit hozzá adunk a PATH változóhoz, hogy a YAML konfigurációban történő hivatkozás megvalósulhasson külső python állományra.
    - `-d` : Debug mód, mindent kiír a stdout-ra.
    - `-l` : Hány elemen történjen a feldolgozás. Nagyszerű lehetőség nagy fájlok tesztelésére kis rekordokon, amíg minden nem úgy működik, ahogy szeretnénk.
    - `-o` : Hányadik elemtől kezdje a feldolgozást.

  A `-p` kapcsolóról később lesz szó részletesebben.

3. `metl-transform [options] CONFIG.YML FIELD VALUE`

   Feladata a YAML fájlban szereplő egyik mező transzformációs lépéseinek tesztelése. Paraméterül várja a mező megnevezését, és azt az értéket, amelyen a tesztelést végeznénk. A script ki fogja írni lépésről-lépésre a mező értékének alakulását.

    - `-p` : Mappa átadása, amit hozzá adunk a PATH változóhoz, hogy a YAML konfigurációban történő hivatkozás megvalósulhasson külső python állományra.
    - `-d` : Debug mód, mindent kiír a stdout-ra.

   A `-p` kapcsolóról később lesz szó részletesebben.

4. `metl-aggregate [options] CONFIG.YML FIELD`

   Feladata kigyűjteni a paraméterül átadott mező összes lehetséges értékét. Ezen értékek alapján utána már könnyen készíthető Map a rekordokhoz.

    - `-p` : Mappa átadása, amit hozzá adunk a PATH változóhoz, hogy a YAML konfigurációban történő hivatkozás megvalósulhasson külső python állományra.
    - `-d` : Debug mód, mindent kiír a stdout-ra.
    - `-l` : Hány elemen történjen a feldolgozás. Nagyszerű lehetőség nagy fájlok tesztelésére kis rekordokon, amíg minden nem úgy működik, ahogy szeretnénk.
    - `-o` : Hányadik elemtől kezdje a feldolgozást.
    - `-s` : Ha a konfigurációs állomány nem tartalmazza a resource útvonalát, itt is megadható.

   A `-p` kapcsolóról később lesz szó részletesebben.

5. `metl-differences [options] CURRENT_MIGRATION LAST_MIGRATION`

   Feladata két különböző migráció összehasonlítása. Első paramétere a friss, és második paramétere a korábbi migráció. A script megmondja, mennyi elem került bele az újba, mennyi elem módosult, mennyi elem maradt változatlan, illetve került törlésre.

    - `-n` : Konfigurációs állomány az új elemek kulcsainak kiírására.

    - `-m` : Konfigurációs állomány a módosult elemek kulcsainak kiírására.
    - `-u` : Konfigurációs állomány a módosulatlan elemek kulcsainak kiírására.
    - `-d` : Konfigurációs állomány a törölt elemek kulcsainak kiírására.
6. `metl-generate [options] SOURCE_TYPE CONFIG_FILE`

   Egy választott forrás állományból készít Yaml kongirációs állományt. Ahhoz hogy a konfiguráció elkészülhessen meg kell adni a forrás alapvető inicializálási és forrás paramétereit.

   Szükséges paraméterek listáját a következőképpen le lehet kérni: `metl-generate SOURCE_TYPE CONFIG_FILE`

   Támogatott forrás típusok listája: CSV, Database, Google Spreadsheet (jelszóval védett), JSON, TSV, XLS, XML, Yaml

    - `-l` : Meghatározza, hány elem átviszgálásával készítse el konfigurációs állományt. Minél több rekord kerül átnézésre, annál pontosabb eredményt tud adni a megfelelő típus használatát illetően, de a konfiguráció készítésének ideje drasztikusan nőhet ennek hatására.

Működés

Az eszköz egy **YAML fájlt használ konfigurációnak**, ami leírja a teljes végrehajtás útját, és az összes elvégzendő transzformációs lépést.

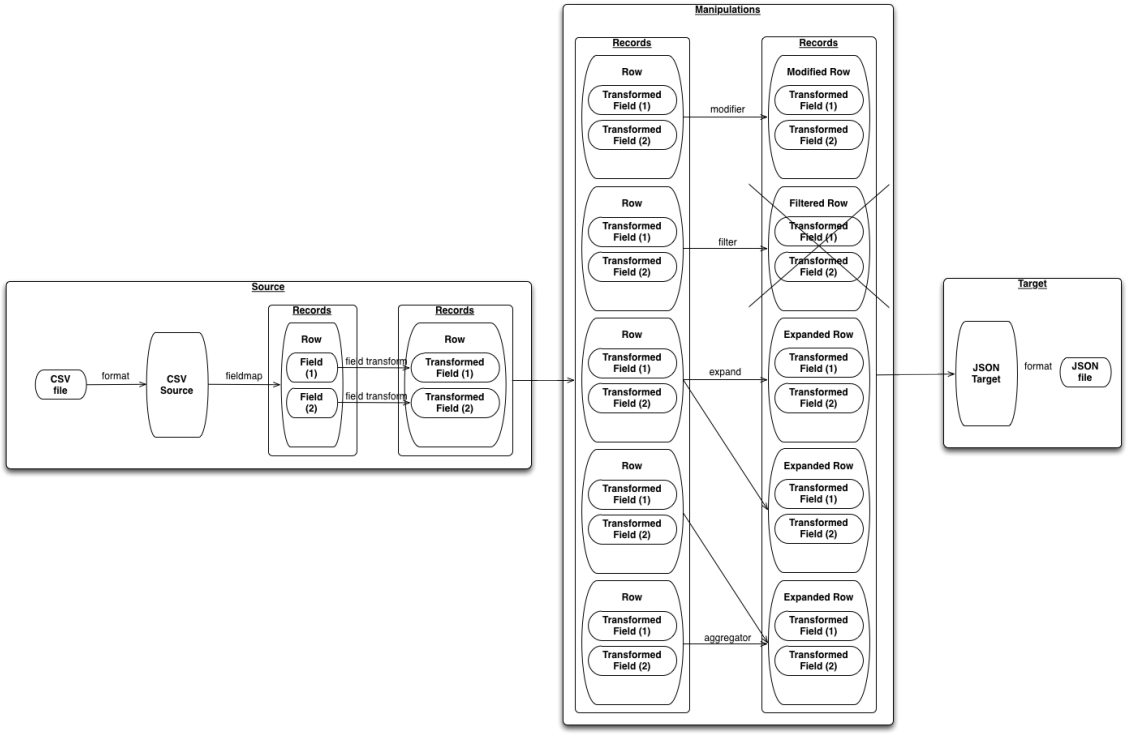
Rövid működése egy átlagos programnak a következőképpen néz ki:

1. A program beolvassa a megadott forrás állományt.
2. Soronként egy illesztés felhasználásával betölti megfelelő mezőkbe a sor értékeit.
3. Mezőkön egyesével hívódnak meg a tetszőleges bonyolultságú transzformációk.
4. Végleges, transzformációkon átjutott sor kerül az első manipulációhoz, ahol a további szűrések, módosítások már a teljes sor értékeire érvényben lehetnek. Minden manipuláció a következő manipulációs lépés számára adja át a már konvertált és feldolgozott sort.
5. Cél típushoz kerülés után, megtörténik a végleges sor kiírsa a megadott típusú állományba.

Nézzük meg a működés során használt összes komponenset részletesen, majd pedig nézzük meg a felsorolt lépésekből, hogyan tudunk konfigurációs YAML állományokat készíteni.

Az alábbi dokumentáció két témakört igyekszik lefedni, egyrészt definiálja, hogy a YAML konfigurációban hogyan tudjuk leírni a szükséges feladatokat, illetve egy rövid betekintést ad példákon keresztül a Python oldali kódba és legfontosabb metódusokba, hogy ha az alap eszköz funkció készlete kevésnek bizonyulna, hogyan tudunk gyorsan és egyszerűen kiegészítő feltételeket, módosítókat készíteni.

Működési ábra



YAML konfigurációs állomány

Minden mETL konfigurációs folyamat állománya a következő formában néz ki:

```
source:
  source: <forras_tipusa>
  …

manipulations:
  - <manipulacios_lepesek_felsorolasa>
  …

target:
  type: <target_tipusa>
  …
```

Ebből a manipulációs lépések megadása nem kötelező, hiszen egyszerűbb folyamatok esetében nincs szükség rájuk. Az első érdekesség, hogy a konfigurációs állományok korlátlan szintig örköltethetőek, így nagyon hasonló kongfigurációs állományokat lehet ugyanabból az "absztakt" konfigurációból származtatni. Erre nézzük a következő példát.

Legyen a fájl neve **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
```

A következőé, pedig **fromcsv.yml**:

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

Utóbbi állomány az elsőből származott, és csak azt mondta meg hogy ha CSV állományból kell feldolgoznia az adatokat, akkor az állomány hol található, és mi az illesztés a mezőkre. Ha lenne egy TSV állományunk is, amely ugyanez a formátumú, csak az elválasztó jel a CSV hagyományos `,`-vel szemben `\t`, akkor arra a következő konfiguráció adható:

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

Forrás (Source)

Minden folyamat egy forrás állománnyal kezdődik, ahonnan az adatokat beolvassuk. Típusok egyediek, saját beállításokkal rendelkezik. Források szerepe összetett, mivel a teljes ETL procedúrában bármilyen transzformáció, manipulációs lépés képes behívni további

forrásokat a műveleteik elvégzésére, így helyes példányosításuk és illesztésük kritikus. Utóbbiról sokkal később. Az **összes forrás típus** esetén a következő adatokkal rendelkezhet:

- **source**: Forrás típusa
- **fields**: Mezők listája
- **map**: Mezők illesztése. Megadása nem kötelező, mezők szintjén is megadható.
- **defaultValues**: Alapértelmezett értékek mezőkhöz. Megadása nem kötelező, mezők szintjén is megadható.

Példa egy Static forrás YAML konfigurációjára:

```
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 Craw', 'The Disappointed Craw@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: ReplaceByRegexp
          regexp: '^([0-9]{4}-[0-9]{2})-[0-9]{2}$'
          to: '$1'
    - name: year
      type: Integer
```

Példa sok, még nem ismert adatot és szerkezetet tartalmazhat, ezeket a részeket később elemezzük részletesen.

Forrás - ennek megfelelően - a következő dologért felelős:

1. Adatokat tartalmazó állomány típusának és formátumának leírása (source)
2. Beolvasott adatstruktúra leírása (fields)
3. Fentiek közötti illesztések definiálása (map)

Nézzük meg, hogy tudjuk az adatokat tartalmazó állományok típusát leírni.

CSV

CSV állományok esetén használatos forrás típus. Inicializálásának paraméterei:

- **delimiter**: CSV állományban használt elválasztó jel. Alapértelmezetten `,`-t használunk.
- **quote**: Milyen karaktert használunk adatok levédésére, ha a szöveg tartalmazza a korábban említett delimiter-t. Alapértelmezetten `"`-t használunk.
- **skipRows**: Megmondja, hogy a CSV fájl elejéből ennyi sort hagyjunk ki. Alapértelmezetten nem hagyunk ki egyetlen sort sem a feldolgozás során.
- **headerRow**: Megmondható, hogy hányadik sorban találhat a CSV fájl header-je. Amennyiben megadjuk, akkor a mezőnkénti illesztés nem index (sorszám), hanem oszlop név alapján történhet csak.

Forrás adatokhoz szükséges további paraméterek:

- **resource**: CSV állomány útvonala, amely akár URL is lehet.
- **encoding**: CSV állomány kódolása. Alapértelmezetten `UTF-8`-at várunk.

Kivonatolt, példa YAML konfiguráció CSV forrásra:

```
source: CSV
resource: utvonal/fajl/neve.csv
delimiter: "|"
headerRow: 0
skipRows: 1
```

Database

Adatbázisból történő adatolvasásra használatos forrás típus. Több féle működésre képes, de tekintsük meg az forrás adatok beolvasásához szükséges paramétereket:

- **url**: Adatbázis csatlakozási URL-je.
- **schema**: Adatbázis sémája, amihez csatlakozni kell. Nem kötelező!
- **table**: Adatbázis táblája, amiből az adatokat olvassuk be.
- **statement**: Egyedi lekérdezés adható adat beolvasásra. Ha van megadva, akkor a `table` paraméter megadása nem kötelező.

Ennek fényében, nézzünk két példa YAML konfigurációt. Az első legyen egy `SQLite` adatbázis `test` táblája.

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

A második pedig egy `PostgreSQL` adatbázisból történő egyedi lekérdezés:

```
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

Fix szélességű fájl esetén használatos forrás típus. Inicializálásának paramétere:

- **skipRows**: TXT fájl elejéből ennyi sort hagy ki. Alapértelmezetten nem hagy ki

egyetlen sort sem a feldolgozás során.

Forrás adatokhoz szükséges további paraméterek:

- **resource**: TXT állomány útvonala, amely akár URL is lehet.
- **encoding**: TXT állomány kódolása. Alapértelmezetten `UTF-8`-at várunk.

Kivonatolt, példa XLS konfigurációra:

```
source: FixedWidthText
resource: utvonal/fajl/neve.txt
skipRows: 1
```

GoogleSpreadsheet

Van lehetőség Google Spreadsheet forrás használatára is. Inicializálásához nem szükséges paraméterek, azonban a forrás adatokhoz a korábbiakhoz képest nagyon is sokra van szükség:

- **username**: Felhasználó név.
- **password**: Jelszó
- **spreadsheetKey**: SpreadSheet kulcsa.
- **spreadsheetName**: SpreadSheet neve.
- **worksheetId**: Munkalap azonosítója.
- **worksheetName**: Munkalap neve.

Fenti paraméterek közül egyik sem kötelező, de a forrás működésképtelen ha nem kap megfelelő adatokat. Kitöltésükkor a következő szabályok érvényesek.

1. **Publikus Google SpreadSheet** esetén, csak a `spreadsheetKey` megadása szükséges. Publikus Spreadsheet-ek kezelése nem tökéles, ha az állomány tartalmaz `:` és `,` karaktereket, problémás eredmények születhetnek. A hiba sajnos a Google hatásköre, mivel publikus dokumentumok esetén nem cellánként ad vissza értékeket, hanem ömlesztve, szövegesen, levédő karakterek nélkül.
2. **Nem publikus SpreadSheet esetén** a `username`, `password` mezők megadása kötelező, továbbá a `spreadsheetKey` vagy `spreadsheetName` valamelyike. Ha pontos munkalapra akarunk hivatkozni, úgy szintén elegendő `worksheetId` és `worksheetName` közül az egyik megadása.

Példa publikus Google SpeadSheet YAML konfigurációra:

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

JSON

JSON állományok esetén használatos forrás típus. Inicializálásának paramétere:

- **rootIterator**: Mi a gyökérelem neve, ami az adatok listáját tartalmazza. Megadása nem kötelező, de ha nincs megadva, akkor az egész JSON állomány adatait tekintjük egyetlen-egy rekordnak. Tömegesen azokat az adatokat a `metl-walk`-al dolgozhatjuk fel.

Példa a fenti rootIterator-ra, ahol a `rootIterator` értéke az `items`:

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

Forrás adatokhoz szükséges további paraméterek:

- **resource**: JSON állomány útvonala, amely akár URL is lehet.
- **encoding**: JSON állomány kódolása. Alapértelmezetten `UTF-8`-at várunk.

Kivonatolt, példa YAML konfigurációra:

```
source: JSON
resource: utvonal/fajl/neve.json
rootIterator: items
```

Static

Főleg tesztelésre használatos forrás típus, ahol maga a konfigurációs állomány tartalmazza a rekordokat. Egyetlen paraméterrel rendelkezik:

- **sourceRecords**: Adatok listája, tetszőleges formátumban.

Példa lehet a feljebb bemutatott kódrészlet:

```
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

TSV állományok esetén használatos forrás típus. Inicializálásának paraméterei:

- **delimiter**: TSV állományban használt elválasztó jel. Alapértelmezetten `\t`-t használunk.
- **quote**: Milyen karaktert használunk adatok levédésére, ha a szöveg tartalmazza a korábban említett delimiter-t. Alapértelmezetten `"`-t használunk.
- **skipRows**: Megmondja, hogy a TSV fájl elejéből ennyi sort hagyjunk ki. Alapértelmezetten nem hagyunk ki egyetlen sort sem a feldolgozás során.
- **headerRow**: Megmondható, hogy hányadik sorban található a TSV fájl header-je. Amennyiben megadjuk, akkor a mezőnkénti illesztés nem index (sorszám), hanem oszlop név alapján történhet csak.

Forrás adatokhoz szükséges további paraméterek:

- **resource**: TSV állomány útvonala, amely akár URL is lehet.
- **encoding**: TSV állomány kódolása. Alapértelmezetten `UTF-8`-at várunk.

Kivonatolt, példa YAML konfiguráció CSV forrásra:

```
source: TSV
resource: utvonal/fajl/neve.tsv
headerRow: 0
skipRows: 1
```

XLS

XLS állományok esetén használatos forrás típus. Inicializálásának paramétere:

- **skipRows**: XLS fájl elejéből ennyi sort hagy ki. Alapértelmezetten nem hagy ki egyetlen sort sem a feldolgozás során.

Forrás adatokhoz szükséges további paraméterek:

- **resource**: XLS állomány útvonala, amely akár URL is lehet.

- **encoding**: XLS állomány kódolása. Alapértelmezetten `UTF-8`-at várunk.
- **sheetName**: XLS állomány munkalapjának neve, vagy sorszáma.

Kivonatolt, példa XLS konfigurációra:

```
source: XLS
resource: utvonal/fajl/neve.xls
skipRows: 1
sheetName: Sheet1
```

XML

XML állományok esetén használatos forrás típus. Inicializálásának paramétere:

- **itemName**: Az adatot tartalmazó blokk neve. Megadása nem kötelező, amennyiben nem kerül megadásra, maga az egész állomány minősül egyetlen rekordnak! Tömegesen azokat az adatokat a `metl-walk`-al dolgozhatjuk fel.

Példa az **itemName** megadására, ha a fájl több rekordot tartalmaz. Ebben az esetben az `itemName` értéke `item` kell hogy legyen.

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

Forrás adatokhoz szükséges további paraméterek:

- **resource**: XML állomány útvonala, amely akár URL is lehet.
- **encoding**: XML állomány kódolása. Alapértelmezetten `UTF-8`-at várunk. Az XML fájl header-je ha tartalmaz encoding paramétert, annak kódolásának is meg kell egyeznie a fájl kódolásával.

Kivonatolt, példa XML konfigurációra:

```
source: XML
resource: utvonal/fajl/neve.xml
itemName: item
```

Később az illesztés (Map) során fontos figyelni arra, hogy az XML-ek bejárására és útvonalaikhoz az xml2dict csomag kerül felhasználásra, így az útvonal megadásánál a tényleges érték a `text` attribútumba fog kerülni. Példa illesztésre `latitude`, `longitude` és `name` mezők esetén a fenti XML-re:

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

Yaml

YAML állományok esetén használatos forrás típus. Inicializálásának paramétere:

- **rootIterator**: Mi a gyökérelem neve, ami az adatok listáját tartalmazza.

Példa a fenti rootIterator-ra, ahol a `rootIterator` értéke az `items`:

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

Forrás adatokhoz szükséges további paraméterek:

- **resource**: YAML állomány útvonala, amely akár URL is lehet.
- **encoding**: YAML állomány kódolása. Alapértelmezetten `UTF-8`-at várunk.

Kivonatolt, példa YAML konfigurációra:

```
source: Yaml
resource: utvonal/fajl/neve.yml
rootIterator: items
```

Pár blokkal feljebb, említésre került, hogy a forrás a következő dolgokért felelős:

1. Adatokat tartalmazó állomány típusának és formátumának leírása (source)
2. Beolvasott adatstruktúra leírása (fields)

3. Fentiek közötti illesztések definiálása (map)

Az első pontot átnéztük, jöjjön a második, azaz hogy hogyan történik a beolvasott adatsturktúra leírása.

Mező (Field)

Minden forrás állomány esetén kötelező megadni a forrásban szereplő mezőket. Természetesen, ha valamelyik mezőre a folyamathoz nincs szükség, akkor annak szerepeltetése nem szükséges, csak akkor, ha a kimenetben is szeretnénk feltüntetni. Azokat a mezőket azonban feltétlenül fel kell sorolni, amelyekbe később értéket szeretnénk írni, hiszen a folyamat közben nincs lehetőség újabb mezők hozzáadására. Minden mező a következő értékkel rendelkezhet:

- **name**: Mező megnevezése, melynek egyedinek kell lennie.
- **type**: Mező típusa, ha nem definiáljuk alapértelmezetten String.
- **map**: Illesztés leírása. Nem kötelező, illetve megadható a forrás szintjén is.
- **finalType**: Mező végső típusa, ha a transzformációk során megváltozik a beolvasotthoz képest.
- **key**: Kulcs mező-e. Minden esetben állítsunk minden mező értéket igaz-ra, amely egyértelműen képes azonosítani egy sort, hogy használhassuk majd a migrációs tulajdonságokat.
- **defaultValue**: Alapértelmezett kezdőérték. Csak akkor használható ha nincs a mezőhöz illesztés definiálva.
- **transforms**: Transzformációs lépések.
- **limit**: Adatbázisban használt mezőhossz.
- **nullable**: Üresen hagyható-e az érték, ha nem abban esetben üres szövegként fogjuk eltárolni.

Példa YAML konfigurációs szerepeltetésre:

```
- name: uniquename
  type: Float
```

Két legfontosabb Python oldali metódusa a következő:

- **setValue(value):** Érték beállítást végez a mezőn.
- **getValue():** Lekérdezi a mező aktuális értékét.

Példa Python oldalról:

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

Mező típus (FieldType)

Minden mező kötelezően rendelkezik egy típussal. A következő típusokat kezeli jelenleg az mETL:

- **Boolean**: Igaz-Hamis mező.
- **Complex**: Összetett típus, bármilyen adat tárolására. Dict/List esetén érdemes használni, ha később még teendőnk van az adott értékkel.
- **Date**: Dátum típus.
- **Datetime**: Dátim és idő típus.
- **Float**: Tört szám mező típus.
- **Integer**: Egész szám mező típus.
- **List**: Lista típus bármilyen adat tárolására.
- **String**: Szöveges mező típus.
- **Text**: Hosszú szöveges mező típus.

A típus érték konverzióra használható. Alap feladata, egy beérkező értéket a megadott típusú elemmé konvertálni. Amennyiben a konverzió nem sikerül, vagy üres értékkel rendelkezik (pl.: üres szöveg), akkor None értéket fog felvenni. `None` értékkel minden mező típus rendelkezhet, attól a típus értéke megfelelő.

Példa Python oldalról:

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

Szerepeltetése YAML konfigurációs állományban:

```
type: Date
```

A legérdekesebb típus a `List`, mivel nehéz elképzelni bizonyos állományok esetében. `XML`-ben, `JSON`-ban eredeti formájában fog tárolódni, `CSV`, `TSV`-ben a Python list lesz szöveggé konvertálva, `Database` cél esetén pedig az adatbázisban `VARCHAR` mezőben `JSON`-ként kerül tárolásra.

Mező transzformációk (Transforms)

TARR csomag felhasználásval történik a mező transzformációk kezelése az mETL-en belül. Hagyományos listát vár, amely tartalmazhat transzformációkat és programszerkezeteket is egyaránt. Konfigurációs oldalon azonban van lehetőség a then szerkeszet felhasználásával picit rendszerezni a lépéseket, hogy könnyebben olvashatóbb maradhasson.

Működése egyszerű, programszerkezetek szerint bejárja a transzformációkat, majd a legvégén ha a mező finalType értéke különbözik a mező akkori típusától, megpróbálja átkonvertálni az értéket.

Nézzük a következő YAML konfigurációt egy mezőre:

```yaml
- 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
```

Amit elsőre észre kell venni, hogy az állományból `Integer` típussal történt a beolvasás, így biztosak lehetünk, hogy minden olyan érték, amely nem értelmezhető számként `None`-ként került tárolásra. Mivel `finalType` értéke `String`, így egy típus váltásra is fel kell készülni a mező transzformációja során. Az első transzformáció egy ConvertType, amely elvégzi a fenti típus módosítást, míg az azt követő a `Map`, amely egy-egy értékhez különböző értékeket rendel. Így egy szám mezőből végül szöveges értékes mezőt hoztunk létre, ahol immáron a kerületek eredeti nevei szerepelnek. Minden transzformációt a `transform` kulcsszóval kell megnevezni.

Ilyen jellegű transzformációkat minden egyes mezőre külön-külön lehet definiálni.

Nézzünk még egy hasonlóan egyszerű példát, mielőtt belemennénk a bonyolultabb transzformációk leírásába:

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

Ennél a mezőnél a célunk, hogy egy hasonlóan egész szám értékből római számot generáljunk. Mivel ilyen alapértelmezett transzformációval az mETL nem rendelkezik, így a egy külső hívást végzünk más csomag tartalmával. Ha nem telepített csomagról van szó, akkor az mETL számára a `-p` **paraméter** segítségével tudjuk kiegészíteni a `PATH` környezeti változót, hogy töltse be a kért Python package-et.

```
@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
```

Ezzel a módszerrel könnyedén tudunk egyedi transzformációkat adni a projektünkhöz. Ennek a módszernek összesen egy hiányossága van, az, hogy az így definiált transzformációk számára nem adható további paraméter, így átlalánosabb feladatok elvégzésére nem biztos hogy felhasználható.

Példa képpen magát a StripTransform kódját láthatjátok:

```
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
```

Ezzel a módszerrel tudunk olyan transzformációt adni a rendszerhez, amely további paramétereket fogadhat a következő formában:

```
transforms:
  ...
  - transform: package.utvonal.StripTransform
    chars: -
  ...
```

Fenti példa annyiban sántít, hogy az mETL által alapértelmezett transzformációk neve mögé soha nem kell odatenni a Transform szót, illetve útvonalat sem kell számukra adni.

A transzformációkkal kapcsolatban volt szó arról, hogy támogatja a programszerkezeteket is. Nézzünk erre is egy példa YAML konfigurációt:

```
- 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
```

A fenti beolvas egy dátum mezőt, amelyből a folyamat végére igaz-hamis értéket generál. Ehhez programszerkezeteket használ. Mint minden számítógép közeli programozási nyelvben, az `IF`-et `ENDIF`-el kell lezárni. A fenti példa, megnézi hogy a beolvasott dátum két intervallum között helyezkedik-e el, és ha igen akkor felveszi az igaz értéket, különben pedig a hamis értéket. Az értékek beállítására a fenti példa mutat több lehetőséget egyaránt.

Programszerkezetek esetén a `transform` kulcsszó helyett a `statement` kulcsszót kell használni. Feltételekhez pedig a `condition` kulcsszóra van szükség, viszont mielőtt belemerülnénk a előbbibe, nézzük meg milyen feltételek vannak és hogy történik azok paraméterezése.

Feltétel (Condition)

Minden feltétel a `condition` kulcsszóra példányosítható, azonban önmagában nincsen jelentősége, csak programszerkezetek és bizonyos manipulációs objektumok használják döntéshozásra. Egy feltétel, pontosan **egy mezőre hoz igaz-hamis döntést**, teljes sorokra és mezők közötti összefüggésekre nem használható!

Python oldalról a következő formában működnek a feltételek:

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

Ahogy a hagyományos transzformációk esetén volt, itt is definiálhatóak egyedi feltételek a következő módon:

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

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

illetve, természetesen akár paraméterezett formában is elkészíthető a fenti:

```
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 )
```

Jelen esetben itt is az egyik beépített feltétel kódja látható. A fenti verzióval szemben az alsó annyival bővebb, hogy itt paraméterként átadható az a szám, aminél nagyobb értéket szeretnénk kapni, illetve a kapott számra egy típus konverzió is megtörténik, hogy azonos típuson történjen a kiértékelés.

IsBetween

Mező értékbe benne van-e egy megadott intervallumban. Csak `Integer`, `Float`, `Date`, `DateTime` típusok esetén van értelme használni. Paraméterei:

- **fromValue**: Intervallum minimum értéke
- **toValue**: Intervallum maximum értéke

Példa YAML konfigurációra:

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

IsEmpty

Megvizsgálja, hogy a kapott mező üres-e. Nem vár paramétert, és bármilyen típusra alkalmazható.

Példa YAML konfigurációra:

```
condition: IsEmpty
```

IsEqual

Mező értéke megegyezik a paraméterül kapott értékkel. A feltétel bármilyen típus esetén használható. Paramétere:

- **value**: Vizsgált érték az összehasonlításkor

Példa YAML konfigurációra:

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

IsGreaterAndEqual

Mező értéke nagyobb vagy egyenlő a paraméterül kapott értékkel. A feltétel bármilyen típus esetén használható. Paramétere:

- **value**: Vizsgált érték az összehasonlításkor

Példa YAML konfigurációra:

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

IsGreater

Mező értéke nagyobb a paraméterül kapott értékkel. A feltétel bármilyen típus esetén használható. Paramétere:

- **value**: Vizsgált érték az összehasonlításkor

Példa YAML konfigurációra:

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

IsLessAndEqual

Mező értéke kisebb vagy egyenlő a paraméterül kapott értékkel. A feltétel bármilyen típus esetén használható. Paramétere:

- **value**: Vizsgált érték az összehasonlításkor

Példa YAML konfigurációra:

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

IsLess

Mező értéke kisebb a paraméterül kapott értékkel. A feltétel bármilyen típus esetén használható. Paramétere:

- **value**: Vizsgált érték az összehasonlításkor

Példa YAML konfigurációra:

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

IsIn

Mező értéke egyike a paraméterül kapott értékeknek. A feltétel bármilyen típus esetén használható, melynek paramétere:

- **values**: Összehasonlításkor vizsgált értékek listája

Példa YAML konfigurációra:

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

IsInSource

Az IsIn feltételhez nagyon hasonló működésű, azonban a vizsgálathoz használt értékeket egy másik forrás állományból tölti be és onnan ellenőrzi, hogy szerepel-e a mező értéke az állományban. Paraméter:

- **join**: Másik forrásban hogy hívják azt a mezőt, amely azt az értéket tartalmazza, amelyre az IsIn feltételt alkalmazni akarjuk.

Természetesen vannak további paraméterek is, hiszen a feltételhez csatolni kell a teljes `Source` konfigurációját is.

Példa YAML konfigurációra:

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

IsMatchByRegexp

Reguláris kifejezést használ a mező értékének kiértékelésére. Akkor tekinti sikeresenek az adott mezőt, ha illeszkedik rá a reguláris kifejezés. Paraméterei:

- **regexp**: Vizsgálandó reguláris kifejezés.
- **ignorecase**: Hagyja figyelem kívül a kis- és nagybetűket a reguláris kifejezés kiértékelése során. Alapértelmezetten megkülönbözteti őket!

Példa YAML konfigurációra:

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

Programszerkezet (Statement)

A **transform** kulcsszó helyett a **statement** kulcsszót kell használni, természetesen programszerkezeteket továbbra is a Mezők transzformációs lépéseiben használhatunk csak, hogy a mező értékét végleges formába hozzuk és sikeres adattisztítást végezhessünk rajta. Programszerkezetek **korlátlan mennyiségig egymásba ágyazhatóak**, viszont mind lezárása szükséges.

IF

"Ha, akkor" feltételként szolgál, akárcsak a hagyományos programnyelvekben. Minden **IF**-et kötelező valamikor követnie egy **ENDIF**-nek. Paramére:

- **condition**: Feltétel, minden szükséges paraméterével.

Gyakorlatilag nem kötelező, de adható számára a YAML konfigurációban egy **then** is, amely az alá tartozó transzformációkat tartalmazza.

IFNOT

"Ha nem, akkor" feltételként szolgál. Ugyanazok a szabályok és paraméterek vontakozának rá, mint az **IF**-re. Paramétere:

- **condition**: Feltétel, minden szükséges paraméterével.

ELIF

Ha a feltétel nem teljesül, azonban újabb feltételt definiálnánk, akkor erre a programszerkezetre lesz szükségünk. **IF** és **ENDIF** között használható, szigorúan **ELSE** előtt. Ugyanazok a szabályok és paraméterek vontakozának rá, mint az **IF**-re. Paramétere:

- **condition**: Feltétel, minden szükséges paraméterével.

ELIFNOT

Ugyanazok a szabályok és paraméterek vontakozának rá, mint az **ELIF**-re, csak a feltétel tagadása esetén teljesül. Paramétere:

- **condition**: Feltétel, minden szükséges paraméterével.

ELSE

Ha a feltétel nem teljesül, és nem szeretnénk további feltételeket szabni, de szükség van egy feltétel nélküli ágra, ahova a be kell futnia a transzformációnak, akkor használhatjuk. Szigorúan **IF** és **ENDIF** között, ha jelen van, akkor pedig **ELIF** és **ELIFNOT** után. Paraméterrel nem rendelkezik.

ENDIF

IF feltételt lezáró kulcsszó. Paraméterrel nem rendelkezik.

ReturnTrue

Kilép a feltételből és megszakítja a transzformációkat. Paraméterrel nem rendelkezik.

Transzformáció (Transform)

Volt szó arról, hogy a transzformációk listáját tudunk mezőkhöz definiálni. Ezeket a transzformációkat a **transform** kulcsszóval jelöljük, amely után megnvezzük a használandó transzformáció nevét. Rendszerben elérhetőek az alábbiak.

Add

A mező értékéhez hozzáad egy számot. Csak `Integer` és `Float` mezők esetében használható. Paraméterei:

- **number**: Szám, amellyel megszeretnénk növelni a mező aktuális értékét.

Példa YAML konfigurációra:

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

Példa a fenti példa transzformáció eredményére:

```
12
=> 16
```

Clean

Eltávolítja a különféle írásjeleket a kért mezőből. Fontos, hogy csak `String`, és `Text` mezők esetében használható. Paraméterei nem kötelezőek, de felüldefiniálhatóak:

- **stopChars**: Mely karaktereket távolítsa el a mező értékei közül. Alapértelmezetten: .,!?"
- **replaces**: Érték párok listája, hogy mit-mire cseréljen továbbá a tisztítás részeként.

Példa YAML konfigurációra:

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

Példa a fenti példa transzformáció eredményére:

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

ConvertType

Módosítja a mező típusát egy másik típusra. Mivel nem minden mező típus konvertálható veszteség nélkül másik típusra, így itt komolyabb paraméterek várnak beállításra.

- **fieldType**: Új mező típus megnevezése.
- **hard**: Kényszerített típus módosítás kérése, amely hatására a jelenlegi érték megsemmisül. Alapértelmezetten hamis értékkel rendelkezik.
- **defaultValue**: Alapértemezett érték beállítása kényszerített típus módosítás esetén. Alapértelmezetten nincs alapértelmezett érték meghatározva.

Amennyiben egy dátumot szeretnénk szöveggé konvertálni, nincsen szükség hard módra, hiszen ez az átlakítás relatív fájldalommentesen elvégezhető, illetve szerencsés esetben a másik irányban is kivitelezhető. Azonban egy dátum mező értéke Boolean-é történő alakítása már nem ennyire egyértelmű művelet, itt feltétlenül a hard módot kell alkalmazni. Fontos megjegyezni, hogy a **mező finalType érték módosítása nem hard elven történik**, így erről ezzel a transzformációval gondoskodni előtte.

Példa YAML konfigurációra:

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

vagy

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

Homogenize

Az ékezetes karaktereket ékezet nélküliekre alakítja `String` és `Text` mezők esetén. Nagyon gyakori művelet ez, ha értéket szeretnénk párosítani más forrásból érkező adatokkal, hiszen az adatok minősége nagyon kérdéses lehet, így azonban könnyen a keresztellenőrzéseket elvégezni. Paramétert nem vár.

Példa YAML konfigurációra:

```
- transform: Homogenize
```

Példa a fenti transzformáció eredményére:

```
u'árvíztűrőtükörfúrógépÁRVÍZTŰRŐTÜKÖRFÚRÓGÉP
=> 'arvizturotukorfurogeparvizturotukorfurogep'
```

LowerCase

Kisbetűssé alakítja a mező értékét `String` és `Text` mezők esetén. Paramétert nem vár.

Példa YAML konfigurációra:

```
- transform: LowerCase
```

Példa a fenti transzformáció eredményére:

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

Map

Kicserél mező értékeket, más értékre. Ezeket kulcs, érték párokban szükséges megadni. Csak `String` és `Text` mezőtípus esetén működik megfelelően. Paraméterei:

- **values**: Kulcs-érték párok halmaza, amely az átalakítási értékeket tartalmazza.
- **ignorecase**: Figyelmen kívül hagyja-e a kiértékeléskor a kis- és nagybetűk közti különbségeket.
- **elseValue**: Nem közelező paraméter. Akkor adjuk meg, ha minden értéket valamire módosítanánk ha nem szerepel a megadott listában.
- **elseClear**: Nem kötelező paraméter. Akkor adjuk meg, ha minden értéket ürítenénk, amennyiben nem szerepel a values listában.

Példa lehet, a korábban már látott YAML konfiguráció:

```
- 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
```

amely a következő eredményt produkálja:

```
'4'
=> 'Újpest'
```

RemoveWordsBySource

Egy másik forrás állományt felhasználva eltávolít szavakat egy tetszőleges mondatból, amely `String` vagy `Text` mezőtípusban szerepel. Szavakat szóköz mentén választja el, így `Clean` futtatása erősen javasolt előtte.

Saját paraméterrel nem rendelkezik, de a teljes `Source` konfigurációjára itt is szükség van. A forrás egy mezőt tartalmazhat, vagy ha többet tartalmaz csak az elsőt veszi figyelembe a transzformáció, és az ott található értékeket kezdi törölni az adott mező szövegéből.

Példa YAML konfigurációra:

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

ReplaceByRegexp

Reguláris kifejezés alapján cserét hajt végre `String` és `Text` típusú mezőkön. Használható paraméterek:

- **regexp**: Reguláris kifejezés, amely alapján a csere megoldható.
- **to**: Mivé cserélünk a reguláris kifejezés alapján. A szokásos Python-os szintaxissal szemben, a `\\` jel helyett a `$` jel használható a kiemelt paraméterek beemelésére.
- **ignorecase**: Megkülönböztesse-e a reguláris kifejezés kiértékelésekor a nagy- és kisbetűket. Alapértlemezetten megteszi.

Példa YAML konfigurációra, amely egy szöveges formátumó dátumból csak az év-hónap párt tartja meg:

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

Fenti a következő eredményt produkálja:

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

ReplaceWordsBySource

Egy másik forrás állományt felhasználva cserél le szavakat `String` és `Text` mezők esetén. Paramétrei:

- **join**: Másik forrásban hogy hívják azt a mezőt, amely ugyanazt az értéket tartalmazza, amely alapján a jelenlegi forrást össze szeretnénk kapcsolni a másikkal. A két forrásban a mező nevének meg kell egyeznie!

Természetesen vannak további paraméterek is, hiszen a feltételhez csatolni kell a teljes `Source` konfigurációját is. **Nagyon fontos, hogy az itt megadott forrás összesen kettő (2) mezőt tartalmazhat, a join-nál megadott mezővel együtt!** Az érték, amelyre a csere meg fog történni, így a join feltételben nem szereplő oszlop lesz!

Példa YAML konfigurációra:

```
- 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

Érték beállítást végez bármilyen típusú mezőn. Paraméterei:

- **value**: Új mezőérték

Van lehetőség String vagy Text mező esetén az új értékbe beleszúrni a régi értéket is, erre nézzünk egy YAML konfigurációt:

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

Példa a fenti transzformáció eredményére:

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

Split

Szóközök mentén elválasztja a szavakat és a megadott intervallumot hagyja meg `String` és `Text` mező típusok esetén. Fontos, hogy meg lehet adni egy konkrét számot (pl.: 1), ebben az esetben azzal az index-el rendelkező szót hagyjuk meg, de megadható intervallum is kettősponttal elválasztva (pl.: 2:-1). Paramétere:

- **idx**: Kivonatolás index-e, 0-tól számozva.
- **chars**: Milyen karakter mentén történjen a vágás. Alapértelmezetten whitespace-ek mentén.

Példa YAML konfigurációra:

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

Példa a fenti transzformáció eredményére:

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

Stem

Szótőre hozza a `String` vagy `Text` mező által tartalmazott szavakat. A szavakat szóközök mentén választja el, tehát a `Clean` transzformáció használata az esetek magas százalékában szükségszerű. Paramétere:

- **language**: Szótövezés nyelve

Példa YAML konfigurációra:

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

Példa a fenti transzformáció eredményére:

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

Művelet megvalósításához az nltk **SnowballStemmer** csomagját használja.

Strip

Eltávolítja az érték elején és végén levő felesleges szóközöket vagy egyéb karaktereket. Csak String és Text mezők esetén használható, paraméterei:

- **chars**: Milyen karaktereket távolítson el a szöveg elejéről és végéről. Alapértelmezetten minden whitespace karakter.

Példa YAML konfigurációra:

```
- transform: Strip
```

Példa a fenti transzformáció eredményére:

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

Sub

A mező értékéből kivon egy számot. Csak `Integer` és `Float` mezők esetében használható. Paraméterei:

- **number**: Szám, amellyel csökkenteni szeretnénk a mező aktuális értékét.

Példa YAML konfigurációra:

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

Példa a fenti példa transzformáció eredményére:

```
12
=> 8
```

Title

Minden szót nagy kezdőbetűssé alakít. Csak String és Text mezők esetén használható, paramétert nem vár.

Példa YAML konfigurációra:

```
- transform: Title
```

Példa a fenti transzformáció eredményére:

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

UpperCase

Nagybetűssé alakítja a mező értékét `String` és `Text` mezők esetén. Paramétert nem vár.

Példa YAML konfigurációra:

```
- transform: UpperCase
```

Példa a fenti transzformáció eredményére:

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

Már többször említésre került, hogy a forrás a következő dolgokért felelős:

1. Adatokat tartalmazó állomány típusának és formátumának leírása (source)
2. Beolvasott adatstruktúra leírása (fields)
3. Fentiek közötti illesztések definiálása (map)

Az első két pontot megnéztük, nézzük meg hogy történik ezek alapján a beolvasott adat sorok illesztése a fent definiált tetszőleges mezőkre.

Illesztés (FieldMap)

Tudjuk, hogy van egy állományunk amely értékeit be szeretnénk olvasni, és már vannak mezőink amibe el szeretnénk helyezni ezeket az értékeket minden egyes sorhoz. Egyedül egy illesztés elkészítése hiányzik, amelyet minden forráshoz, egységesen meg tudunk adni.

Mezőkhöz két helyen lehet illesztést definiálni, ezek rendre:

1. `Source` rekordban egy `map` paraméter alatt.

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

2. Magában a `Field`-ben `map` néven.

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

Mindkettő azonos eredményt fog produkálni. Az első verzió szerencsésebb, ha szeretnénk származtatni belőle konfigurációs állományt, hiszen úgy a mezőket nem kell feltétlenül újra definiálnunk. A második pedig abból a szempontból szerencsésebb, hogy minden ott van ahova tartozik, segíti a könnyebb átláthatóságot. **Ha nem adunk meg egy mezőhöz map-et, akkor alapértelmezetten a mező nevével megegyező útvonalon keres értéket.**

Minden illesztés egy útvonalat jelent az "adat"-hoz. Az útvonal pedig tartalmazhat szavakat, számokat (index-eket), és ezek kombinációit `/` jellel elválasztva.

Ennek fényében nézzünk egy komplexebb példát, amely alapján könnyebben megérthető a működése. Hasonló adatszerkezet várható `XML`, `JSON`, `YAML` állományokból, illetve ennek lapított verziója várható `Database`, `GoogleSpreadsheet` forrásból is.

```
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' ]}
```

Azonban több adatforrás esetén, mint például `CSV`, `TSV`, `XLS` listák érkeznek. Megjegyzésképpen, `CSV` és `TSV`-ként a `headerRow` paraméter megadásával elérhető, hogy Ők is a fenti formátumban kapjanak lapított értékeket.

```
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'}
```

Fontosabb operátorok:

- `/`: Elválaszt egy szintet az bejeárási útvonalban.
- `*`: Lista elem várásakor az összes elemet bejárja. List típus használata, és JSON vagy XML céltípus javasolt, ha ebben a formában szeretnénk eltárolni és nem szöveggé konvertálva. XML és JSON forrás használata esetén használatos leggyakrabban.
- `~`: Próba operátor (List 2 Dict), amennyiben lehet a szinten lista illetve dict is, akkor használjuk, ha létezik a lista megmondhatjuk mire vagyunk kíváncsiak, ha dict van nem történik semmi, megyünk tovább az útvonalon a következő elemtől. XML és JSON forrás esetén használatos.
- `!`: Konvertáló operátor (Dict 2 List), amely akkor használatos ha mindenképpen listát szeretnénk kapni, de nem tudjuk hogy azt fogunk-e kapni. XML forrás esetén használatos.

Manipulációk (Manipulation)

Miután a teljes sor feldolgozásra került, és a mezőkbe bekerültek az értékek s lefutottak a mező szintű transzformációk van lehetőség manipulálni a teljes, tisztított értékeket azok összefüggései alapján. Négy kulcsszó használható, ezek a `modifier`, `filter`, `expand`, `aggregator` mindegyik egy-egy típust jelöl. Feladataink során többnyire őket fogjuk használni, ha komplexebb feladat elvégzése a cél. (pl.: API kommunikáció). Manipulációs lépések **bármilyen sorrendben jöhetnek** egymás után, függetlenül típusoktól. Ha az egyik végzett, átadja eredményét a következőnek, és így tovább, amíg `Target` objektumhoz nem ér.

```
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
```

Fenti példa elmagyarázására nem kerül sor, célja a kulcsszó használat és formátum bemutatása.

Módosító (Modifier)

Módosítók azok az objektumok, amelyek egy teljes sort (rekordot) kapnak, és mindig egy teljes sorral térnek vissza. Azonban a folyamataik során érték módosításokat végeznek a különböző mezők összefüggő értékeinek felhasználásával. Manipulációk közül mindig `modifier` kulcsszóval kezdődnek, és őket fogjuk munkánk során legtöbbször használni.

Mielőtt megnéznénk, milyen rendszerszintű módosítókkal rendelkezik az mETL, megnézzük hogy tudunk újakat hozzádni, hiszen ahogy korábban említettük, ezekre lesz a leggyakrabban szükség.

```
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
```

Ez a példa azt mutatja meg, hogy egy bármilyen forrásban szereplő három érték (vezetéknév, keresztnév, város) felhasználásával előállítunk egy új értéket (telefonszámok listája). Viszont az adatot egy API híváson keresztül gyűjtjük be. A fenti nem rendelkezik paraméterekkel, egyszerűen beilleszthető a folyamatba.

```
- modifier: utvonal.MitoAPIPhoneSearch
```

Felüldefiniálásnál tehát két dolog van, amire figyelni kell:

1. `Modifier` osztályból számraztassuk
2. `modify` függvényt írjuk felül, és az az egész `record`-al térjen vissza.

JoinByKey

Két forrást kapcsol össze a belső forrás által definiált kulcs alapján. A művelet úgy történik, hogy a belső forrás számára ki kell emelni a kulcs mezőket, ezeket keresi meg a külső forrásban. Ha egyezés történik, akkor a `fieldNames`-ben felsorolt mezőkkel frissíti a külső forrást. Mind a kulcs, és mind a frissítendő mezőknek **ugyanazt a nevet kell adni**! Kulcs alapú kapcsoláskor minden sorhoz egy rekord tartozhat!

- **fieldNames**: Mely mezők kerüljenek frissítésre. Mindkét forrásban ugyanilyen néven kell szereplniük a mezőknek!

Példa YAML konfigurációra:

```
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

Sorrendbe rendezi a rekordokat a megadott mezők alapján. Paraméterei:

- **fieldNamesAndOrder**: Mely mezőkön történjen a sorbarendezés és milyen sorrendben. Sorrendnek `ASC` és `DESC` adható csak meg.

Példa YAML konfigurációra:

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

Set

Érték beállítást végez fix érték séma, függvény, vagy másik forrás felhasználásával. Leggyakrabban használt módosító, azonban fontos megemlíteni, hogy gyorsabb és optimális futás eredményeképpen gyakran célszerű saját módosítót írni helyette. Inicializálásának paraméterei:

- **fieldNames**: Mely mezőkön történjen az értékbeállítás.
- **value**: Milyen új értéket vegyenek fel a mezők.

Set működése komplikált, és kiegészíthető továbbá egy `fn` paraméterrel is, ahol tetszőleges értékadó függvény adható meg számára, amit minden mezőre meghív, illetve adható neki teljes forrásleírás is egyaránt.

Használatának módjai:

1. **Érték módosítás**

   Röviden arról szól, hogy a mezők neveit a `value` paraméterbe illesztve, elállítható az értékmódosítás a mezők aktuális értékei alapján. Minden `fieldNames`-ben felsorolt mezőre megtörténik az értékbeállítás.

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

2. **Érték módosítás függvény segítségével**

   Komplex számítást akarunk végrehajtani, akkor érdemes ilyen módon használni a `Set` módosítót. Természetesen a függvényeket nekünk kell elkészíteni, így a `-p` paraméter így itt is szükséges lesz a metl script futtatásakor.

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

   Fentihez a következő függvény kerülhet megírásra:

```
 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 )
```

A fenti függvényben a `record` a teljes sort jelenti, a `field` az aktuálisan beállítandó mezőt (minden `fieldNames`-ben feltüntetett értékre meghívásra kerül ez a függvény), a scope pedig magát a `SetModifier`-t jelenti.

3. **Érték módosítás más forrás segítségével**

   A legbonyolultabb típusa a módosítónak, viszont ha fontos az optimális sebesség, akkor ezt mindenféleképpen érdemes felüldefiniálni az ismert forrás adatszerkezetének fényében. Szükséges `fn` és `source` megadása is.

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

   Hozzá tartozó függvény a következő:

```
 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

Érték beállítást végez egy összetett típus bejárásának segítségével.

- **fieldNamesWithMap**: Mező nevek és map útvonalak, amiken végre kell hajtani a beállítást.

- **complexFieldName**: Összetett mező neve, amiből ki akarjuk venni az értékeket. Ennek típusa lehet `List` vagy `Complex`.

Minden `fieldNames`-ben feltüntetett mezőre egyesével végrehajtjuk a transzformációkat.

Példa YAML konfigurációra:

```
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
    map: 0/createdAt
  - filter: DropField
    fieldNames:
      - LISTELEMENT
      - LISTITEMS
```

TransformField

Hagyományos mező szintű transzformáció hívható általa a manipulációs lépés során. Inicializálásának paraméterei:

- **fieldNames**: Mező nevek, amikre végre kell hajtani a transzformációkat.
- **transforms**: Mező szintű transzformációk listája.

Minden `fieldNames`-ben feltüntetett mezőre egyesével végrehajtjuk a transzformációkat.

Példa YAML konfigurációra:

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

Szűrő (Filter)

Szűrést végeznek elsősorban. Olyankor használatosak, amikor a korábbi lépésekben transzformációk segítségével megtisztított értékeket szeretnénk kiértékelni és eldobni, ha a rekordot hiányosnak, vagy nem megfelelőnek ítéljük meg.

Ha új szűrőt szeretnénk a rendszerbe helyezni, akkor a következő adhat hozzá segítséget:

```
from metl.utils import *

class TetszolegesFilter( Filter ):

    # bool
    def isFiltered( self, record ):

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

DropByCondition

Feltétel alapján dönthető el a rekord sorsa. Inicializálásának paraméterei a következők:

- **condition**: Már korábban is ismertetett feltétel, az összes paraméterével.
- **fieldNames**: Mely mező(k)re szeretnénk a vizsgálatot értelmezni.
- **operation**: Milyen feltétel van a mezők kiértékelése között. Alapértelmezetten `AND` feltétel.

Nézzünk három példát. Az első legyen az, hogy ha a `NAME` mező értéke illeszkedik egy mintára szeretnénk ha nem kerülbe az eredményhalmazba. Egy mező kiértékelése esetén az `operation` paramétere érdektelen.

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

Második példánkban tekintsünk egy picit más jellegű kiértékelést. Szeretnénk törölni az adott sort, ha az `EMAIL` és a `NAME` értékek is üresek.

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

Harmadik példában pedig nézzük meg az `OR` `operation`-el a korábbit. Itt akkor törlődik az eredményhalmazból a sor, ha a `NAME` és `EMAIL` mezők bármelyike üres.

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

DropBySource

Másik forrás állományban történő szereplés dönt a rekord sorsáról. Minden szabály megegyezik a DropByCondition-al, tehát az esetek itt nem kerülnek bemutatásra. Inicializálása:

- **condition**: Már korábban is ismertetett feltétel, az összes paraméterével. Csak olyan feltétel használható, amely 1 paraméterrel rendelkezik!
- **join**: Mező megvezezése, amik összetartoznak. A két forrásban egyforma néven kell szerepelniük.
- **operation**: Milyen feltétel van a mezők kiértékelése között. Alapértelmezetten `AND` feltétel.

és természetesen a `source` összes paraméterével. A feltételhez tartozó egyetlen paraméternek (amely általában a `value`) kell tartalmaznia, hogy a másik forrás melyik oszlopában található az összehasonlítási érték.

Példa YAML konfigurációra:

```
- 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

Mezők törölhetőek a segítségével egy rekordból. Előfordulhat, hogy egy-egy forrásban szereplő értéket több mezőbe is beillesztjük, hogy azokkal különféle transzformációkat végezzünk el, és a folyamat végén szeretnénk törölni a már nem kellő mezőinket. Erre ad lehetőséget ez a szűrő.

- **fieldNames**: Törlendő mezők listája

Példa YAML konfigurációra:

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

KeepByCondition

Feltétel alapján dönthető el a rekord sorsa. Inicializálásának paraméterei a következők:

- **condition**: Már korábban is ismertetett feltétel, az összes paraméterével.
- **fieldNames**: Mely mező(k)re szeretnénk a vizsgálatot értelmezni.
- **operation**: Milyen feltétel van a mezők kiértékelése között. Alapértelmezetten `AND` feltétel.

Szinte egy az egyben megegyezik a `DropByCondition` függvénnyel, azzal a különbséggel hogy itt a rekord akkor nem kerül filterezésre ha teljesül rá a feltétel!

Bővítő (Expand)

Bővítésre használjuk, ha további értékeket szeretnénk a jelenlegi forrás után helyezni.

Append

Az épp akutális forrással teljesen megyező formátumú állomány beolvasására és az akutális folyamatba történő behelyezésére ad lehetőséget. Paraméterei:

- **skipIfFails**: Ha hibás a forrás állomány, akkor kihagyjuk, nem állítjuk le a folyamatot. `logFile` és `appendLog`-al lehet menteni a hibás források listáját.

Fontos, hogy **új mezőkkel nem bővíti a korábbi forrást**, minden ugyanúgy folytatódik, mint ha egy másik `resource`-t kapott volna az aktuális állomány, `modifier`-ek és `target` nélkül. Természetesen bármilyen **resouce** attribútomot felül lehet írni, akár htaccess-hez kapcsolódó `username`, `password`, vagy `encoding` adatokat is!

Példa YAML konfigurációra:

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

AppendAll

Az épp akutális forrással teljesen megyező formátumú állományok beolvasására és az akutális folyamatba történő behelyezésére ad lehetőséget. Paraméterei:

- **folder**: Mappa amelyben levő fájlokat hozzá kell szúrnia a végére. Amennyiben tartalmazza a kiinduló állományt, őt figyelmen kívül hagyja.
- **extension**: Kiterjesztésű fájlokon jár csak végig.
- **skipIfFails**: Ha hibás a forrás állomány, akkor kihagyjuk, nem állítjuk le a folyamatot. `logFile` és `appendLog`-al lehet menteni a hibás források listáját.

Fontos, hogy **új mezőkkel nem bővíti a korábbi forrást**, minden ugyanúgy folytatódik, mint ha egy másik `resource`-t kapott volna az aktuális állomány, `modifier`-ek és `target` nélkül. `Append`-et hoz létre minden fájlra, és úgy történik a folyamat végrehajtása.

Példa YAML konfigurációra:

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

AppendBySource

Másik forrás állomány tartalma szúrható az eredeti forrás után. Inicializáláshoz összesen egy `source`-t vár minden szükséges paraméterével együtt.

Fontos, hogy **új mezőkkel nem bővíti a korábbi forrást**, minden név-név alapján párosít be az eredeti forrás adatai és oszlopaiba. Így mindig ugyanolyan néven kell szereplnie a két forrásban szereplő, megegyező mezőknek. Az itt definiált forrás, eredeti forrásban nem létező mezői pedig nem kerülnek bele az eredményhalmazba!

Példa YAML konfigurációra:

```
- 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

Paraméterül megadott oszlopokat egy másik oszlopba gyűjt össze az oszlop értékeivel. Akkor használhatjuk ha egy statisztika pár sorát egymás alá szeretnénk felsorolni kulcs-érték formában, úgy hogy közben meghagyjuk az összes eredeti oszlopot. Inicializálása:

- **fieldNamesAndLabels**: Mezők és hozzájuk tartozó megnevezések, amelyeket össze akarunk vonni két oszlopba.
- **valueFieldName**: Annak a mezőnek a neve, ahova az oszlop értéke beírásra kerül.
- **labelFieldName**: Annak a mezőnek a neve, ahova az érték oszlopának megnevezése bírásra kerül.

Példa YAML konfigurációra:

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

BaseExpander

Kiegészítésre használatos osztály, amely önmagában nem működőképes. Feladata akkor van, amikor egy sorból szeretnénk több sort létrehozni a folyamatban. A prototípus lekérdezésénél a `clone()` metódusról ne feledkezzünk el!

```
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

Lista típusú elemet bont értékei alapján külön sorokba. `BaseExpander`-ből származik, így működésileg nagyon közel van hozzá. Paraméterei:

- **listFieldName**: Lista típusú elem neve, amely értékeit külön sorokba kell tördelni.
- **expandedFieldName**: Hova írjuk a lista elem aktuális értékét. Adható neki típus további típuskonverzió végett.
- **expanderMap**: Akkor használatos, ha egy lista értékét több mezőbe szeretnénk belírni. Ilyenkor adhatunk minden mezőhöz egy-egy map-et, arról hogy a listán belül honnan szedje az értékeket.

Fontos, hogy a két mező soha nem lehet azonos. Ha utólag nincs szükség a lista elemre, egy `filter` lépéssel eltávolítható a feleslegessé váló mező. Az `expandedFieldName` és `expanderMap` egyike közelező.

```
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

Megadott oszlopokat rögzíti és a többi oszlopot kulcs-érték párok alapján jeleníti meg. **A folyamat során a nem rögzített és nem kulcs-értéket tartalmazó mezők egytől-egyig törlődnek**. Ha nincs szükség a mezők eltávolítására, és csak pár mezők akarunk összeolvasztani, használjuk a `Field` bővítőt. Iniclaizálása:

Példa YAML konfigurációra:

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

Csoportosító (Aggregator)

Csoportok képzésére és azokból információk kalkulálására használjuk. Az Aggregator-ok gyakran viselkednek Filter, és Modifier-ként is, hiszen gyakran törölnek sorokat, oszlopokat, és módosítanak és gyűjtenek össze bizonyos értékeket. Az ilyen műveletek mindig `aggregator` kulcsszóval kezdődnek.

Avg

Átlag érték meghatározására használható. Inicializálása:

- **fieldNames**: Mely mezők számítanak a csoportba. Ezek az értékek distinct formában fognak szerepelni ezt követően!
- **targetFieldName**: Melyik mezőbe írjuk a számosságot.
- **valueFieldName**: Mely mező értékét adjuk össze.
- **listFieldName**: Lista mező neve, amibe azon sorokat menthetjük, akik beleszámítanak a számosságba. Megadása nem kötelező.

A `fieldNames`, `targetFieldName`, `listFieldName` kivételével a többi oszlopot törli az csoportosító.

Példa YAML konfigurációra:

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

Count

Számosság meghatározására használható. Inicializálása:

- **fieldNames**: Mely mezők számítanak a csoportba. Ezek az értékek distinct formában fognak szerepelni ezt követően!
- **targetFieldName**: Melyik mezőbe írjuk a számosságot
- **listFieldName**: Lista mező neve, amibe azon sorokat menthetjük, akik beleszámítanak a számosságba. Megadása nem kötelező.

A `fieldNames`, `targetFieldName`, `listFieldName` kivételével a többi oszlopot törli az csoportosító.

Példa YAML konfigurációra:

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

Sum

Érték összeadására használható. Inicializálása:

- **fieldNames**: Mely mezők számítanak a csoportba. Ezek az értékek distinct formában fognak szerepelni ezt követően!
- **targetFieldName**: Melyik mezőbe írjuk a számosságot.
- **valueFieldName**: Mely mező értékét adjuk össze.
- **listFieldName**: Lista mező neve, amibe azon sorokat menthetjük, akik beleszámítanak a számosságba. Megadása nem kötelező.

A `fieldNames`, `targetFieldName`, `listFieldName` kivételével a többi oszlopot törli az csoportosító.

Példa YAML konfigurációra:

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

Célállomány (Target)

Miután megtörtént a forrás állományból az adatok beolvasása, illesztése, transzformációja, és végigmentek az adatok a manipulációs lépéseken a `Target`-hez kerül a beolvasott, véglegesített sor. Ő fogja kiírni és létrehozni az állományt a végleges adatokkal.

```
target:
  type: <celallomany_tipus>
  …
```

Target-ből **csak egy lehet** egy mETL folyamatban. Meglévő állományokat folytatni más-más mETL folyamatokkal csak `CSV`, `TSV`, `Database` céltípusok esetén van lehetőség.

CSV

CSV állományok esetén használatos cél típus. Inicializálásának paraméterei:

- **delimiter**: CSV állományban használt elválasztó jel. Alapértelmezetten `,`-t használunk.
- **quote**: Milyen karaktert használunk adatok levédésére, ha a szöveg tartalmazza a korábban említett delimiter-t. Alapértelmezetten `"`-t használunk.
- **addHeader**: A mezők elnevezéseit tegyük-e fel az első sorba, mint header.
- **appendFile**: Ha létezik a célfájl, folytassuk-e az írását, és ne elölről kezdjük. Alapértelmezetten, mindig felülírjuk a fájlokat.

Cél helyszínének megadásához szükséges további paraméterek:

- **resource**: CSV állomány célútvonala, amely akár URL is lehet.
- **encoding**: CSV állomány kódolása. Alapértelmezetten `UTF-8`-ba kódolunk.

YAML konfigurációra példa:

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

Database

Ha adatbázisba szeretnénk írni a rekordjainkat, akkor használjuk ezt a céltípust.
Inicializálásának sok paramétere van:

- **createTable**: Ha nem létezik a tábla az adatbázisba, létre hozza-e. Alapértelmezetten nem hozza létre, feltételezi hogy már létezik a megfelelő sémával.
- **replaceTable**: Akár létezik, akár nem létezik a tábla az adatbázisba, törölje-e és hozza-e létre újra. Alapértelmezetten nem teszi. Használata feltétlenül szükséges, ha a tábla bár létezik, de az új folyamat új oszlopokkal bővítené.
- **truncateTable**: Ürítse-e a már létező táblát, hogy üres állapotba történjen a rekordok írása. Alapértelmezetten nem teszi. Fontos megjegyezni, ha a `replaceTable` értéke igazra van állítva, akkor mindenképpen üres állapotba kerül a tábla, így ennek megadása nem szükséges.
- **addIDKey**: Adjon-e egyértelmű kulcsot autoincrement szekvenciával a táblához a létrehozás pillanatában. Alapértelmezetten megteszi.
- **idKeyName**: Ha az `addIDKey` értéke igaz, mi legyen annak az oszlopnak a neve. Nem lehet ilyen nevű oszlop a kiírandó értékek között.
- **continueOnError**: Írás/módosítás esetén ha hiba történik (pl.: Foreign Key nem szerepel) kihagyható a sor. Alapértelmezetten ki van kapcsolva.

Cél helyszínének megadásához szükséges további paraméterek:

- **url**: Adatbázis kapcsolódás linkje.
- **table**: Tábla neve, amibe írni szeretnénk. Ha megadjuk az írás/módosítást automatikusan elvégzi a rendszer.
- **fn**: Függvény neve, amivel írni/módosítani fogunk. Megadása akkor kötelező, ha táblát nem adunk. Érdemes használni, ha nem feltétlenül egy táblába, hanem komoly Foreign Key-es környezetbe kell több táblába írni. Ha van megadva `table` és `fn` is, akkor az automatikus írás is és a függvény hívás is megtörténik!
- **schema**: Schema megnevezése, amiben a tábla található. Megadása nem kötelező.

YAML konfigurációra példa:

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

vagy

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

a következő Python állománnyal:

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

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

        ...
```

`fn` megadásának fő haszna azonban migrációk készítésekor van. Példa később.

FixedWidthText

Fix szélességű állományok esetén használatos cél típus. Inicializálásának paraméterei:

- **addHeader**: A mezők elnevezéseit tegyük-e fel az első sorba, mint header. Alapértelmezetten teszünk.

Cél helyszínének megadásához szükséges további paraméterek:

- **resource**: TXT állomány célútvonala, amely akár URL is lehet.
- **encoding**: TXT állomány kódolása. Alapértelmezetten `UTF-8`-ba kódolunk.

YAML konfigurációra példa:

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

GoogleSpreadsheet

Spreadsheet állományok esetén használatos cél típus. Inicializálásának paraméterei:

- **addHeader**: A mezők elnevezéseit tegyük-e fel az első sorba, mint header. Alapértelmezetten teszünk.

Cél helyszínének megadásához szükséges további paraméterek:

- **username**: Felhasználó neve.
- **password**: Jelszava.
- **spreadsheetKey**: Írni szándékozott spreadsheet azonosítója.
- **spreadsheetName**: Írni szándékozott spreadsheet neve. spreadsheetKey vagy spreadsheetName egyikének a megadása kötelező!
- **worksheetName**: Munkalap neve. Amennyiben nem létezik, automatikusan létrehozzuk.
- **truncateSheet**: Ürítse-e a munkalap tartalmát. Alapéretelmezetten nem teszi.

YAML konfigurációra példa:

```
target:
  type: GoogleSpreadsheet
  username: ***
  password: ***
  spreadsheetKey: 0ApA_54tZDwKTdDlibXppSkd1MExxb3Y5WmJrZjFxR1E
  worksheetName: Sheet1
```

JSON

JSON állományok esetén használatos cél típus. Inicializálásának paraméterei:

- **rootIterator**: Változó neve, amiben összeszeretnénk gyűjteni a rekordokat. Megadása nem kötelező, üresen hagyása esetén a JSON állomány csak egy listát fog tartalmazni.
- **flat**: Amennyiben csak egy field létezik, kérhetjük hogy az értékek csak felsorolva jelenjenek meg a mező megnevezése nélkül ezzel az opcióval. Alapértelmezetten nem használjuk.
- **compact**: Formázott JSON generálása. Alapértelmezetten az értéke hamis, egy sorba történik a JSON generálása.

Üresen hagyva:

```
[ { … }, { … }, …, { … } ]
```

Kitöltve, például `items` névvel:

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

Cél helyszínének megadásához szükséges további paraméterek:

- **resource**: JSON állomány célútvonala, amely akár URL is lehet.
- **encoding**: JSON állomány kódolása. Alapértelmezetten `UTF-8`-ba kódolunk.

YAML konfigurációra példa:

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

Static

Tesztelés céllal létrehozott típus, amely `stdout`-ra dolgozik TSV formátummal. Inicializálásának paraméterei:

- **silence**: Írjon-e stdout-ra.

YAML konfigurációra példa:

```
target:
  type: Static
  silence: false
```

TSV

TSV állományok esetén használatos cél típus. Inicializálásának paraméterei:

- **delimiter**: TSV állományban használt elválasztó jel. Alapértelmezetten `,`-t használunk.
- **quote**: Milyen karaktert használunk adatok levédésére, ha a szöveg tartalmazza a korábban említett delimiter-t. Alapértelmezetten `"`-t használunk.
- **addHeader**: A mezők elnevezéseit tegyük-e fel az első sorba, mint header. Alapértelmezetten teszünk.

- **appendFile**: Ha létezik a célfájl, folytassuk-e az írását, és ne előlről kezdjük. Alapértelmezetten, mindig felülírjuk a fájlokat.

Cél helyszínének megadásához szükséges további paraméterek:

- **resource**: TSV állomány célútvonala, amely akár URL is lehet.
- **encoding**: TSV állomány kódolása. Alapértelmezetten `UTF-8`-ba kódolunk.

YAML konfigurációra példa:

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

XLS

XLS állományok esetén használatos cél típus. Inicializálásának paraméterei:

- **addHeader**: A mezők elnevezéseit tegyük-e fel az első sorba, mint header. Alapértelmezetten odatesszük.

Cél helyszínének megadásához szükséges további paraméterek:

- **resource**: XLS állomány célútvonala, amely akár URL is lehet.
- **encoding**: XLS állomány kódolása. Alapértelmezetten `UTF-8`-ba kódolunk.
- **sheetName**: XLS állományban a munkalap neve, amibe az írást szeretnénk végrehajtani.
- **replaceFile**: Cseréljük-e a le az egész XLS állományt. így ha már volt korábban ilyen állományunk akár több munkalappal, úgy értékei elvesznek. Alapértelmezetten lecseréljük, ahogy minden más céltípus alapértelmezetten működik.
- **truncateSheet**: Ürítse-e a munkalapot. Logikusan ha a `replaceFile` értéke igaz, akkor ennek a paraméternek nincsen jelentősége, azonban ellenkező esetben nagyon is sok szerepe van hogy a munkalap legvégére kezdjünk írni, vagy pedig cseréljük le a munkalapot az új adatokra. Alapértelmezetten a munkalapot cseréljük.
- **dinamicSheetField**: Amennyiben a fő mezőn kívül szeretnénk további munkalapokat is létrehozni az adatok szétszórásával, akkor itt annak a mezőnek a nevét kell megadni, amely alapján szeretnénk az adatokat csoportosítani. Az itt megadott mező értéke lesz a munkalap neve. Megadása nem kötelező!

Ha nem létező munkalapot adunk meg, a folyamat automatikusan létrehozza azt, ahogy az egész XLS állományt is.

YAML konfigurációra példa:

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

XML

XML állományok esetén használatos cél típus. Inicializálásának paraméterei:

- **itemName**: XML állományban egy rekord milyen tag között szerepeljen. Megadása kötelező!
- **rootIterator**: A fenti XML adatokat milyen nevű gyökérelembe gyűjtsük. Alapértelmezetten `root`-ot használunk. Üresen nem hagyható, így nem generálható 1 elemet tartalmazó XML-ek.
- **flat**: Amennyiben csak egy field létezik, kérhetjük hogy az értékek csak felsorolva jelenjenek meg a mező megnevezése nélkül ezzel az opcióval. Alapértelmezetten nem használjuk.

Cél helyszínének megadásához szükséges további paraméterek:

- **resource**: XML állomány célútvonala, amely akár URL is lehet.
- **encoding**: XML állomány kódolása. Alapértelmezetten `UTF-8`-ba kódolunk.

YAML konfigurációra példa:

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

Yaml

Yaml állományok esetén használatos cél típus. Inicializálásának paraméterei:

- **rootIterator**: Adatokat milyen nevű gyökérelembe gyűjtsük. Üresen nem hagyható, így nem generálható 1 elemet tartalmazó YAML állomány.
- **safe**: Távolítsuk-e el a Python által generált megjelöléseket. (pl.: unicode karakterkódolás) Alapértelmezetten benne hagyjuk őket.
- **flat**: Amennyiben csak egy field létezik, kérhetjük hogy az értékek csak felsorolva jelenjenek meg a mező megnevezése nélkül ezzel az opcióval. Alapértelmezetten nem használjuk.

Cél helyszínének megadásához szükséges további paraméterek:

- **resource**: YML állomány célútvonala, amely akár URL is lehet.
- **encoding**: YML állomány kódolása. Alapértelmezetten `UTF-8`-ba kódolunk.

YAML konfigurációra példa:

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

Migrációk (Migrate)

mETL script futtatásakor van lehetőség migrációs fájl megadására (`-m` paraméter), és új migrációs állomány generálására (`-t` paraméter). Két féle migráció létezik, a **kulcs nélküli** és a **kulcsal rendelkező**. Logikusan ha módosulás történik a konfigurációs állományban a korábbi migráció nem lesz felhasználható a jövőben. A két típusi migráció nem keverhető egymással semmilyen körülményben.

Key, Hash, ID

Minden sor rendelkezik a fenti paramétrekkel.

- **Key**: Kulcs mezőnek megjelölt mezők értékei `-` jellel elválasztva. Ezen értékek egyértelműen azonosítják a rekordot (sort). Nem rendelkezik a sor `key`-el, ha nincs mező megjelölve.
- **Hash**: Teljes rekord (sor) értékeiből képzett, vissza nem fejthető hosszú szám és betűsor.
- **ID**: Fentiek összefűzve, `:`-al elválasztva. Egyértelműen azonosítja egy egyértelműen azonosítható sor, összes értékének jelenlegi állapotát.

Naplózás

Kiinduló forrás, manipuláció, és célállománynak adható egy `log` változó, amelynek egy fájl elérési útját kell magadni. Így lehet bekapcsolni a logolást egy adott lépéshez. Minden lépés különböző formátumú naplót készít, de többnyire a következő mondható el általánosságban.

1. **Forrás napló**: Tartalmazza soronként a beolvasott sor `ID`-ját, a beolvasott sort dictionary-jét JSON-ként, és a transzformálás utáni értékét JSON-ban.
2. **Filter módosító napló**: Tartalmazza soronként a beolvasott sor `ID`-ját, és az eldobott sor transzformált értékét JSON-ban.
3. **Célállomány napló**: Tartalmazza soronként a beolvasott sor `ID`-ját, a műveletet (írás, vagy módosítás), és a kiírásra kerülő sor értékét JSON-ban.

Módosítók és Bővítők, illetve a transzformációs lépések külön-külön nem kerülnek

naplózásra.

Migráció kulcs mezők nélkül

Ebben az esetben nem tudjuk beazonosítani a sorokat egyértelműen, így nem tudjuk meghatározni hogy változott-e az adott sor értéke a korábbihoz képest. Ebben a verzióban egyik mező konfigurációja sem tartalmaz `key` megnevezést, így a migráció összesen annyit képes meghatározni, hogy a korábbi verzióban (`-m`) volt-e megegyező érték a `hash` alapján.

Target-be csak a még nem létező mezők kerülnek beírása. Ha van kérve új migárció generálása (`-t`), akkor a migárciós állomány azonban minden régi és új értékkel rendelkezni fog. Természetesen amik nem szereplnek az új állományban azok nem kerülnek bele az új migárcióba, törölt elemként tekintjük őket, azonban nem jelöljük törlése őket sehol sem.

Migráció kulcs mezők felhasználásával

Gyakrabban használt eset, hiszen majdnem minden forrás esetén találhatunk olyan kombinációt amellyel pontosan azonosítani tudunk egy sort több adat alapján. Ebben az esetben a migráció úgy működik hogy minden `ID`-hoz, eltárolja a sorhoz tartozó `hash`-t. Így ha a megegyező `ID`-hoz a hash megváltozik pontosan tudjuk hogy melyik rekord (sor) értéke módosult. Szöveges források esetén itt is a célállományba kerül minden új, illetve módosult rekord. Adatbázis cél esetén azonban rendes `UPDATE` parancsok kerülnek hívásra. Generálandó (-t) migráció itt is a végleges állapot fogja tartalmazni.

Új/Módosult/Változatlan/Törölt elemek kulcsainak listája

Dokumentáció elején említett `metl-differences` script képes migrációkat összehasonlítani. Erre példa a következő lehet:

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

Amely jól látható kap egy -d paramétert egy konfigurációs állománnyal. Ez mondja meg, hogy hova szeretnénk írni az elemek kulcsait, amik törlésre kerültek az új migrációra. Példa a delete.yml konfigurációra:

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

Jól látszik, hogy csak és kizárólag target-et kell megadni, hisz a többiről a script intézkedik. A fenti ehhez hasonló listát generál:

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

Természetesen az eredeti konfigurációs állományban egyetlen `id` mező tartalmazta a `key` beállítást. **A fenti script csak azonos típusú migrációk között használható!!!**

Azonban legtöbbször a módosítást, vagy a törölt rekordok listájára teljesen más miatt van szükség. Gyakoribb, hogy egy teljes migráció új rekordjait egy adatbázisba írjuk, a törölt rekordokat azonban inaktiválnánk. Erre használatos a `DatabaseTarget` `fn` attribútuma.

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

esetén a delete.yml tartalma:

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

míg, az mgmt.py állomány tartalma pedig:

```
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 ] ) )
    )
```

Példa

Most hogy túljutottunk az eszköz által nyújtott rövid lehetőségeken, nézzünk meg pár dolog, hogy mire is lehet használni.

Spanyol adatok betöltése

Nagyon egyszerű betöltőről van szó összeségében, azonban az adatok mennyisége miatt lehet érdekes példa bevezetőnek. Több GB-nyi adatról van szó, amely közel egy évtizedet ölel fel. Havonta megközelítőleg 10db 80MB-os állományból áll, fájlonként kb. 250 000 sorral. Cél az adatok betöltése egy adatbázis táblába, lehetőleg minél gyorsabban.

Következő konfiguráció készült hozzá:

```
source:
```

```
source: FixedWidthText
map:
  FLOW: '0:1'
  YEAR: '1:3'
  MONTH: '3:5'
  CUSTOM_ENCLOSURE_RPOVINCE: '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_RPOVINCE
  - 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
```

Ahogy látható nincs megadva `resource` paraméter a `Source` esetében, mivel nem áll szándékunkban hasonló formátumú állományokhoz külön-külön konfigurációkat készíteni. Futtatáshoz emiatt `metl-walk`-ot használunk, multiprocess (-m) beállítással, hogy a havonta szereplő 10 fájl egyszerre dolgozódjanak fel lehetőleg minél hamarabb.

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

Csoprtos adatkonvertálás és gyűjtés

Gyakran van szükség arra, hogy teljesen értelmetlen adatforrásokból, értelmes, átlátható adatokat készítsünk. Következő formátumú állományok érkeztek minden megyéhez:

```
{
   "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"
      },
      …
   ]
}
```


Célunk egy TSV állomány generálása, amely minden adatot tartalmaz, ami ezekben a fájlokban fellelhető. Ehhez használt konfiguráció:

```
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
```

A programot a következő formátumban futtattuk: `metl-walk config.yml data/`

Long format konvertálás táblázatos formából

Field bővítő használatával

Van egy TSV állományunk a következő formával:

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

Amelyhez a következő konfigurációt készítjük el:

```
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
```

Ezzel a következő eredményt kaphatjuk meg.

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

Melt bővítő használatával

Nézzük meg a következő input állományt:

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

Példa konfigurációs állomány az adatok long értékre hozására key-value párok mentén:

```
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
```

Amely hatására a következő jön létre:

```
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
```

Adat betöltésre adatbázis két táblájába

Nézzünk egy komplex példát, amely a ListExpander bővítő felhasználásán alapul.

```
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
```

Itt az adatbázisban a `geometry` mező értéke `JSON` lesz. Szeretnénk egy másik táblába szétbontani ezt a listát `latitude` és `longitude` koordinátánként. Jelenleg a `geometry` mezőben ilyen értékek szerepelnek:

```
[[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]]
```

Ezt pedig a következő konfigurációval akarjuk megvalósítani:

```
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
```

Itt kiolvassuk az előbb betöltött `geometry` mező értékét egy listába, majd a `ListExpander` segítségével definiáljuk, hogy a `geometry` mezőt bejárva, a `latitude` és `longitude` mezőbe mit is írjunk pontosan. Ezzel a megvalósítással létrehoztunk egy táblát és hozzá egy kapcsolótáblát.