data_journalism_extractor Documentation

Release 0.1

Hugo Cisneros

Oct 29, 2018

Contents:

This project is an attempt to create a tool to help journalists extract and process data at scale, from multiple **heterogenous data sources** while leveraging powerful and complex **database**, **information extraction** and **NLP** tools with **limited programming knowledge**.

CHAPTER 1

Features

This software is based on Apache Flink, a stream processing framework similar to Spark written in Java and Scala. It executes dataflow programs, is highly scalable and integrates easily with other Big Data frameworks and tools such as Kafka, HDFS, YARN, Cassandra or ElasticSearch.

Although you can work with custom dataflow programs that suits your specific needs, one doesn't need to know programming, Flink or Scala to work with this tool and build complex dataflow programs to achieve some of the following operations:

- Extract data from **relational databases** (Postgres, MySQL, Oracle), **NoSQL** databases (MongoDB), CSV files, HDFS, etc.
- Use complex processing tools such as **soft string-matching functions**, **link extractions**, etc.
- Store outputs in multiple different data sinks (CSV files, databases, HDFS, etc.)

Some examples are in *Getting started* and *Example Walkthrough*. Description of the modules and how to use them is in *Modules Reference*.

1.1 Getting started

1.1.1 Dependencies

The project makes use of the Apache Flink stream and batch processing framework. Flink needs a working **Java 8.x** installation to run and is compatible with Windows, Linux and MacOS. The code is compatible with **Flink 1.5 and above**.

Apache Flink can be installed by downloading and extracting a binary from here.

Or you can install it with a package manager of your choice (e.g. Homebrew on MacOS), a more detailed description there.

In order to run the code, **Scala 2.12.x** and **sbt** should also be installed (details).

Finally, the compiler is compatible with Python3.6 and above. You can install the dependencies with `pip install -r requirements.txt` from the main directory.

The data journalism extractor has several modules that are based on different softwares. The MongoDB Importer module needs a working MongoDB installation in order to work.

To run the example, one will need working Postgres and MongoDB installations.

1.1.2 Installation

Get the project from GitHub

The project can be downloaded from the GitHub repository with the command

`git clone git@github.com:hugcis/data_journalism_extractor.git your/path/`

then run `cd your/path/` to get started.

How to run a program

The Flink program that will execute some operations lives in the `scala/src/main/scala/core` directory, along with all the modules and classes used to run operations. The main template is `MainTemplate.scala.template`. It will be rendered with code corresponding to some specifications from a JSON file.

Specifications come in the following JSON structure:

```
{
    modules: [
        {
            moduleType: "string",
            name: "string",
            ...
        },
        {
            moduleType: "string",
            name: "string",
            ...
        },
        ...
    ]
}
```

This defines a **directed acyclic graph** (DAG) of operations, where outputs of some modules can be fed to inputs of other modules.

The JSON files defines a program, that can be compiled with a python script as follow:

`python/compile.py -s spec.json`

A more complete description of the available options and functioning of the script can be obtained by running `python/compile.py -h`.

```
Usage: compile.py [-h] [-s filename] [-t dirname] [-o output] [-p]

Command line interface for compiling JSON spec file in Scala code.

Optional arguments are:
    -h, --help                          show this help message and exit
```

(continues on next page)

(continued from previous page)

```
    -s filename, --spec filename          spec filename
    -t dirname, --template-dir dirname    template directory
    -o output, --output output            output filename
    -p, --pdf-output                      output the rendered graph in a pdf file
```

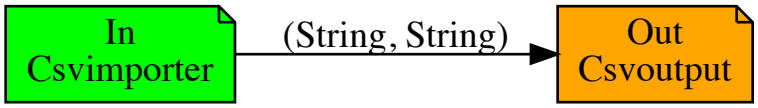
Once the DAG has been compiled, the generated Scala program can in turn be compiled and packaged into a JAR to run on a Flink cluster (either locally or on a cluster).

Minimal Example

Take the following simple program:

```
{
    "modules": [
        {
            "moduleType": "csvImporter",
            "name": "In",
            "path": "~/project/in.csv",
            "dataType": ["String", "String"]
        },
        {
            "moduleType": "csvOutput",
            "name": "Out",
            "path": "~/project/out.csv",
            "source": "In"
        }
    ]
}
```

This defines a program that will copy a CSV file with two inputs called `in.csv` and paste it in an other file called `out.csv`. The corresponding DAG looks like this:



It was generated by compiling the above JSON file with `python/compile.py -s spec.json -p`. The flag `-p` is used to generate a DAG representatin of your program in PDF format.

The following lines of Scala code were generated during compilation

```
// set up the execution environment
val env = ExecutionEnvironment.getExecutionEnvironment

// ===== CSV Importer module In =====

val filePath_In = "~/project/in.csv"
val lineDelimiter_In = "\n"
```

(continues on next page)

(continued from previous page)

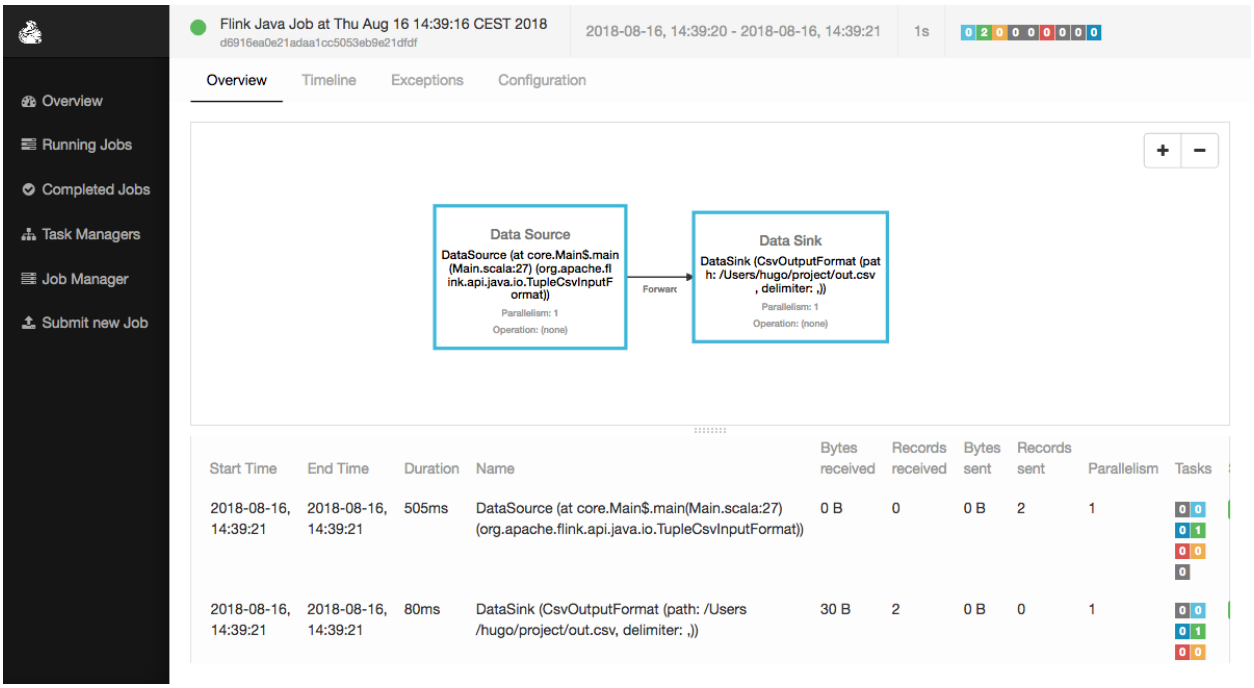
```
val fieldDelimiter_In = ","
val In = env.readCsvFile[(String, String)](filePath_In, lineDelimiter_In,␣
↪fieldDelimiter_In)

// ===== CSV Output File Out =====

val filePath_Out = "~/project/out.csv"
In.writeAsCsv(filePath_Out, writeMode=FileSystem.WriteMode.OVERWRITE)

// ===== Execution =====

env.execute()
```

To make Flink run the program, you need to pack your code into a .jar file with `sbt clean assembly` from the `scala` directory.

Next, make sure you have a Flink task manager running, for example locally (see link). You should be able to see the Flink web interface at http://localhost:8081.

You can then run your program with `flink run target/scala-2.11/test-assembly-0.1-SNAPSHOT.jar` (the exact name of the file depends on the parameters set in `build.sbt`).

You should the be able to see `out.csv` in `~/project` and the following job in your Flink web interface:



A more thorough introduction to this software can be found a *this link*

1.2 Example Walkthrough

The following example will introduce several of the project's modules and their functioning. It was the initial motivation for building this software and was both a source of inspiration for building modules and a testing dataset.

1.2.1 Context

The idea behind the example was to take advantage of multiple open data sets made available by institutions, associations, or open APIs and try to identify "links" (of influence or else) between French MPs and lobbyists.

With the available data, three ways of identifying links came to mind:

- Tweets
- Wikipedia mentions
- Semantic field

1.2.2 Data

The data is available in the `example/data` directory. The list of files is:

- `deputes.csv` the standard CSV export of http://www.nosdeputes.fr 's data. (Available at https://www.nosdeputes.fr/deputes/enmandat/csv)
- `hatvpmongo.json` the JSON export from https://www.hatvp.fr/agora/opendata/agora_repertoire_opendata.json where the root `publications` has been removed to keep the array.

  It should further be inserted in a Mongo database with `mongoimport --jsonArray --db $DB_NAME --collection publications --file hatvpmongo.json`
- `wiki.csv` is the result of retreiving the Wikipedia pages of all MPs and splitting those into sentences to write in a CSV file `$NAME,$SENTENCE` for each MP.
- A Postgres database dump is also available. It contains a corpus of tweets of MPs and lobbyists along with the accounts that retweeted them. The whole database dump is available to download from this Google Drive link. It is a tar archive that you can uncompress with `tar -xvf twitter.dump.bz2`.

  If you have Postgres installed, you can then do `psql twitter < twitter.dump`

1.2.3 Anatomy of the program

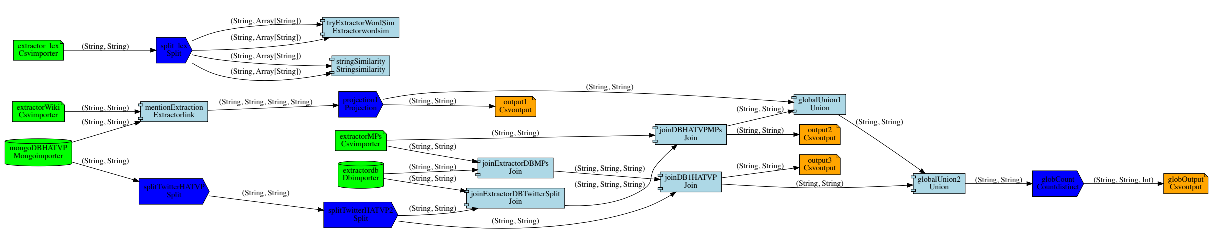


Fig. 1: DAG representing the program. (It was generated with the `-p` option of compile.py)

Tweets

The tweets are organised in three tables.

- The `user` table contains the primary key `uid` (the Twitter id), and the `screen_name` of a user.

- The `checked_statuses` table that represents single tweets. It contains a `sid`, a `uid` referencing a user a `text` and a boolean `ret_checked` indicating wethter retweeters have been fetched from the Twitter API for this status (This is not always the case because of the limits of the API that make fetching tweets a lot faster than fetching retweeters)
- The `retweeted_statuses` table that represents the action of retweeting a tweet. They have an ID `rid`, a user ID `uid` and a status ID `sid`.

The twitter accounts of both the MPs and lobbyists come from their respective data export `deputes.csv` under the column `twitter` and the `lienPageTwitter` field in MongoDB documents representing the lobbyists.

Extraction

Three extraction modules are needed for reading the Mongo database, reading the `deputes.csv` file and fetching data from the Postgres database.

The operations and modules needed for extracting and using data from those sources are the following (you can find more detailed explanation of the capacities of each module in the detailed module *reference*:

- The Mongo reader module is specified below, where `dbName` is the name you chose for your Mongo DB, and `"publications"` is the name of the main collection.

```
{
     "name": "mongoDBHATVP",
     "moduleType": "mongoImporter",
     "dbName": "testdb",
     "collection": "publications",
     "requiredFields": ["lienPageTwitter", "denomination"]
}
```

  Here the reader is instructed to query both the content of `"lienPageTwitter"` and `"denomination"` for each element of the collection.

- The CSV file reader is another module and therefore also needs both a name and type. Moreover, a `path` to the file and `dataType` must be provided.

  Since the delimiter in the file is `;` instead of the standard comma, a `fieldDelimiter` is also added. Finally, since the CSV file has named columns, we can use those name to select only the two that are necessary.

```
{
     "name": "extractorMPs",
     "moduleType": "csvImporter",
     "path": "~/data_journalism_extractor/example/data/deputes.csv",
     "dataType": ["String", "String"],
     "fieldDelimiter": ";",
     "namedFields": ["nom", "twitter"]
}
```

- The Database Extractor works with any relational database that has JDBC support. The module is as follow:

```
{
     "name": "extractordb",
     "moduleType": "dbImporter",
     "dbUrl": "jdbc:postgresql://localhost/twitter",
     "fieldNames": ["rt_name","screen_name"],
     "dataType": ["String", "String"],
     "query": "select rt_name, screen_name from (select rt_name, uid from (select␣
↪us.screen_name as rt_name, rt.sid from retweetedstatuses as rt join users as us␣
↪on (rt.uid=us.uid)) as sub join checkedstatuses as ch on (sub.sid=ch.sid)) as␣
↪subsub join users on (subsub.uid=users.uid);",
```

(continues on next page)

(continued from previous page)

```
    "filterNull": true
}
```

The `dbUrl` field corresponds to the jdbc-format endpoint used to access the database. The field names and data types must also be specified. The query above is quite complete because it is nested and basically retreives every pair of (Twitter user, Retweeter).

The `filterNull` flag set to `true` ensures that no null values are outputed.

Processing

After the three extractions above, three data flows are available to work with:

- `(lobbyists Twitter name, lobbyists name)`
- `(MPs name, MPs Twitter name)`
- `(retweeter name, tweeter name)`

First, the extracted Twitter names of the lobbyists aren't names but URLs to their Twitter accounts. The first step is to extract the names from the pattern `"https://twitter.com/twitter-name?lang=fr"`. This can easily be done by chaining two string splitters to separate the strings on `"/"` and `"?"`. The corresponding modules are:

```
{
    "name": "splitTwitterHATVP",
    "moduleType": "split",
    "source": "mongoDBHATVP",
    "delimiter": "/",
    "field": 0,
    "reduce": -1
},
{
    "name": "splitTwitterHATVP2",
    "moduleType": "split",
    "source": "splitTwitterHATVP",
    "delimiter": "\\?",
    "field": 1,
    "reduce": 0
}
```

Note: The delimiter is a regex pattern and therefore the character `"?"` is represented by `"\?"`, but the antislash must be escaped in strings hence `"\\?"`. Also, column indexing starts at 0 and negative indexing is supported for -1 only (`"reduce": -1`).

Then, a series of joins transforms pairs of Twitter names and retweeter names into pairs of Lobbyists and MPs names.

There are two separate flows:

- One for the tweets authored by lobbyists and retweeted by MPs (`joinExtractorDBTwitterSplit` and `joinDBHATVPMPs`)
- The other for tweets authored by MPs and retweeted by lobbyists (`joinExtractorDBMPs` and `joinDB1HATVP`)

They are explained below:

```
{
    "name": "joinExtractorDBTwitterSplit",
    "moduleType": "join",
    "source1": "extractordb",
    "source2": "splitTwitterHATVP2",
    "field1": 1,
    "field2": 0,
    "leftFields": [0]
},
{
    "name": "joinExtractorDBMPs",
    "moduleType": "join",
    "source1": "extractordb",
    "source2": "extractorMPs",
    "field1": 1,
    "field2": 1,
    "leftFields": [0]
},
{
    "name": "joinDBHATVPMPs",
    "moduleType": "join",
    "source1": "joinExtractorDBTwitterSplit",
    "source2": "extractorMPs",
    "field1": 0,
    "field2": 1,
    "leftFields": [2],
    "rightFields": [0]
},
{
    "name": "joinDB1HATVP",
    "moduleType": "join",
    "source1": "joinExtractorDBMPs",
    "source2": "splitTwitterHATVP2",
    "field1": 0,
    "field2": 0,
    "leftFields": [1],
    "rightFields": [1]
}
```

Join modules have two sources with names of other upstream modules; for each of the sources, a field on wich to perform the join, and two optional list of fields that allow to project the output on the desired columns. For example column 0 on the left and all columns on the right for module `joinExtractorDBMPs`.

Output

An arbitrary number of outputs can be added at any step of the process to log an intermediray output for debugging or store a result.

Two CSV outputs correspond to both MPs' retweets of lobbyists and lobbyists' retweets of MPs.

```
{
    "name": "output2",
    "moduleType": "csvOutput",
    "source": "joinDBHATVPMPs",
    "path": "/Users/hugo/Work/limsi-inria/tests/data_journalism_extractor/example/
↪output/output_dep_retweet_hatvp.csv"
```

(continues on next page)

(continued from previous page)

```
},
{
    "name": "output3",
    "moduleType": "csvOutput",
    "source": "joinDB1HATVP",
    "path": "/Users/hugo/Work/limsi-inria/tests/data_journalism_extractor/example/
↪output/output_hatvp_retweet_dep.csv"
}
```

The

Wikipedia mentions

All French MPs have Wikipedia pages. They usually contain a short bio that gives useful information such as previous occupations or major events in the MP's political career. The Wikipedia API can be used to download the bios and sentence splitting can be applied to obtain the file in `example/data/wiki.csv`.

From a list of pairs of MP name and sentence, different approaches can extract links between MPs and lobbyists. The simplest one is consists in matching every occurence of a lobby's name in the sentences and treating it as an indication of the existence of a link between the two entities. It obviously yields some false positives but nonetheless give an indication that the corresponding lobby has had a relation with the MP. It is the example described below.

Extraction

Data in `wiki.csv` is already pre-processed and thus simply needs an CSV importer module to extract the data.

The second field is quoted between `$` s.

```
{
   "name": "extractorWiki",
   "moduleType": "csvImporter",
   "path": "/Users/hugo/Work/limsi-inria/tests/data_journalism_extractor/example/data/
↪wiki.csv",
   "dataType": ["String", "String"],
   "quoteCharacter": "$",
   "fieldDelimiter": "|"
}
```

Processing

The `extractorLink` module implements a mention extraction algorithm to extract mentions of a given data flow's elements into an other data flow.

The `sourceExtract` and `targetExtract` fields correspond to the column index of the source and target flow. **The source is the data flow mentions of the target will be extracted from.**

```
{
   "name": "mentionExtraction",
   "moduleType": "extractorLink",
   "source1": "extractorWiki",
   "source2": "mongoDBHATVP",
   "sourceExtract": 1,
```

(continues on next page)

(continued from previous page)

```
    "targetExtract": 1
}
```

1.3 Modules Reference

1.3.1 Renderer

Module containing the render engine

class renderer.**ModuleMap**
The main mapping that links modules to their name through the `get` method.

(Essentially an enum or dictionary)

classmethod add_module(*module_name*, *module*)
Set a new module in the mapping.

Parameters

- **module_name** (*str*) – Name of the new module
- **module** (BaseModule) – Module class to add to the mapping

classmethod get(*module_type*)
Returns the module corresponding to the name passed in argument.

Parameters **module_type** (*str*) – The desired module type.

Return type Type[*BaseModule*]

class renderer.**Renderer**(*module_list*, *template_dir*)
The render engine that can build the DAG, check the integrity of the operation graph and generate the rendered Scala code.

Parameters

- **module_list** (*List[Dict[str, Any]]*) – The list of module specifications to be parsed and added to the operation graph.
- **template_dir** (*str*) – The path to the template directory.

check_integrity()
Check the integrity of the graph. Should be called after all the modules have been added to the graph (i.e. after initialization).

get_rendered()
Get the rendered code from the module list.

render_pdf_graph()
Create the graphviz Digraph and render the pdf output graph.

1.3.2 Base Module

The module containing the abstract base class for all operation modules used throughout the rest of the code.

class modules.base_module.**BaseModule**(*module*, *env*, *named_modules*)
The abstract base class for modules. All modules are subclasses of `BaseModule`

Every module object passed to the constructor must contain the `moduleType` and `name` fields.

All modules expose the following common API.

Parameters

- **module** (*dict*) – The `dict` containing the specification of the module. Every module has this parameter that should contain the fields from all its parent classes.
- **env** (*jinja2.Environment*) – The jinja environment where the templates can be retrieved.
- **named_modules** (*Dict[str, Type[BaseModule]]*) – A list of all the other modules of the DAG.

add_to_graph(*graph*)
A method for adding the module to a graphviz graph instance.

Parameters **graph** (*graphviz.dot.Digraph*) – A graphviz Digraph object

check_integrity()
Performs some check on the upstream modules and types when necessary to ensure the integrity of the DAG.

get_out_type()
Returns the output type of the module as a list of strings.

Return type `List[str]`

rendered_result()
Returns a pair of strings containing the rendered lines of codes and external classes or objects definitions.

Return type `Tuple[str, str]`

to_graph_repr()
Generate the representation of the node in the form

```
Name Type:  $moduleType
```

Used for pdf graph generation

Return type `str`

Extractors

The package containing all the extractor operation modules

File Importer

The file importer operation module base class

class modules.extractors.file_importer.**FileImporter**(*module*, *env*, *named_modules*)
File Importer is an abstract class that is used for building modules that read files on disk.

It cannot be used by as is because it is an abstract class.

Parameters **module** (*dict*) – The module dict must have a `path` field that contains the path to the file to be read by the module (Ex: `~/project/file.csv`).

add_to_graph(*graph*)
A method for adding the module to a graphviz graph instance.

Parameters **graph** (*graphviz.dot.Digraph*) – A graphviz Digraph object

check_integrity()
Performs some check on the upstream modules and types when necessary to ensure the integrity of the DAG.

rendered_result()
Returns a pair of strings containing the rendered lines of codes and external classes or objects definitions.

Return type Tuple[str, str]

CSV Importer

The CSV loader operation module

class modules.extractors.csv_importer.**CsvImporter**(*module*, *env*, *named_modules*)
Bases: *modules.extractors.file_importer.FileImporter*

Main CSV loader operation module class.

Parameters **module** (*dict*) – The module dict must have a dataType field that contains the input types as a list of strings. (Ex: ["String", "Int", "Double"])

Other optional fields are:

- fieldDelimiter (csv delimiter if other than comma, Ex: "|")
- quoteCharacter (don't separate within quoted fields, Ex: """)
- namedFields (for selecting only some of the columns by their name, Ex: ["name", "age"])

get_out_type()
Returns the output type of the module as a list of strings.

rendered_result()
Returns a pair of strings containing the rendered lines of codes and external classes or objects definitions.

Return type Tuple[str, str]

JSON Importer

Warning: The JSON importer has limited extraction capacities and the MongoDB should be used instead when possible.

The JSON loader operation module

class modules.extractors.json_importer.**JsonImporter**(*module*, *env*, *named_modules*)
Bases: *modules.extractors.file_importer.FileImporter*

Main JSON loader operation module class

get_out_type()
Returns the output type of the module as a list of strings.

rendered_result()
Returns a pair of strings containing the rendered lines of codes and external classes or objects definitions.

Return type Tuple[str, str]

DB Importer

The Database loader operation module

class modules.extractors.db_importer.**DbImporter**(*module*, *env*, *named_modules*)
Bases: *modules.base_module.BaseModule*

Main database loader operation module class.

Parameters **module** (*dict*) – The module dict must have:

- A `dbUrl` field with the database entrypoint for JDBC. (e.g for a Postgres db named test running on localhost `"jdbc:postgresql://localhost/test"`).
- A `dataType` field with the input data types (Ex: `["String", "Int", "Double"]`).
- The names of the desired columns in `fieldNames` (Ex: `["age", "date", "name"]`).
- The `query` to be interpreted by the db.

Other optional fields are:

- `filterNull` a boolean value for filtering null values from the db output.

add_to_graph(*graph*)
A method for adding the module to a graphviz graph instance.

Parameters **graph** (*graphviz.dot.Digraph*) – A graphviz Digraph object

check_integrity()
Performs some check on the upstream modules and types when necessary to ensure the integrity of the DAG.

get_out_type()
Returns the output type of the module as a list of strings.

rendered_result()
Returns a pair of strings containing the rendered lines of codes and external classes or objects definitions.

Return type `Tuple[str, str]`

Mongo Importer

The Mongo loader operation module

class modules.extractors.mongo_importer.**MongoImporter**(*module*, *env*, *named_modules*)
Bases: *modules.base_module.BaseModule*

Main Mongo loader operation module class The Mongo loader allows to retreive an arbitrary number of fields from MongoDb Documents on convert the into a Flink DataSet.

Parameters **module** (*dict*) – The module dict must have a `dbName` field with the name of the DB (ex: `"hatvpDb"`), a `collection` with the name of the desired collection (ex: `"publications"`), the `requiredFields` of the obtained documents (ex: `["age", "name"]`)

add_to_graph(*graph*)
A method for adding the module to a graphviz graph instance.

Parameters **graph** (*graphviz.dot.Digraph*) – A graphviz Digraph object

check_integrity()
Performs some check on the upstream modules and types when necessary to ensure the integrity of the DAG.

get_out_type()
Returns the output type of the module as a list of strings.

rendered_result()
Returns a pair of strings containing the rendered lines of codes and external classes or objects definitions.

Return type Tuple[str, str]

Operations

Unary Operations

The abstract class for unary operation modules.

class modules.operations.unary_operation.**UnaryOperation**(*module*, *env*, *named_modules*)
The Unary operation base abstract class.

Parameters **module** (*dict*) – The module must contain a source field with the name of the incoming data flow.

add_to_graph(*graph*)
A method for adding the module to a graphviz graph instance.

Parameters **graph** (*graphviz.dot.Digraph*) – A graphviz Digraph object

rendered_result()
Returns a pair of strings containing the rendered lines of codes and external classes or objects definitions.

Return type Tuple[str, str]

Map

The map operation module

class modules.operations.map.**Map**(*module*, *env*, *named_modules*)
Bases: *modules.operations.unary_operation.UnaryOperation*

A module that maps an arbitrary scala function to the incoming data flow.

Warning: Arbitrary scala code will only be checked at compilation and therefore could make the final program fail

Parameters **module** (*dict*) – The module dict must contain a function field that contains the desired scala function to be mapped to the data flow. (ex: "(tuple) => (tuple._1*2, tuple._2)").

The outType field must also be provided to ensure compatibility with downstream modules.

check_integrity()
Performs some check on the upstream modules and types when necessary to ensure the integrity of the DAG.

get_out_type()
Returns the output type of the module as a list of strings.

rendered_result()
Returns a pair of strings containing the rendered lines of codes and external classes or objects definitions.

Return type Tuple[str, str]

Count Distinct

The distinct count operation module

class modules.operations.count_distinct.**CountDistinct**(*module*, *env*, *named_modules*)
Bases: *modules.operations.unary_operation.UnaryOperation*

A module that count distinct elements of a dataflow and append it to the dataflow as a separate column.

Parameters **module** (*dict*) – The module dict must contain the `fields` field which corresponds to a list of columns to group the flow by in order to count the number of distinct elements.

check_integrity()
Performs some check on the upstream modules and types when necessary to ensure the integrity of the DAG.

get_out_type()
Returns the output type of the module as a list of strings.

rendered_result()
Returns a pair of strings containing the rendered lines of codes and external classes or objects definitions.

Return type Tuple[str, str]

Projection

The projection operation module

class modules.operations.projection.**Projection**(*module*, *env*, *named_modules*)
Bases: *modules.operations.unary_operation.UnaryOperation*

A module that projects the incoming dataflow on the fields specified in *fields*.

Parameters **module** (*dict*) – The module dict must contain a `fields` field, an array of integer representing the columns to project on (ex: `[0, 2]`).

check_integrity()
Performs some check on the upstream modules and types when necessary to ensure the integrity of the DAG.

get_out_type()
Returns the output type of the module as a list of strings.

rendered_result()
Returns a pair of strings containing the rendered lines of codes and external classes or objects definitions.

Return type Tuple[str, str]

Split

The split operation module

class modules.operations.split.**Split**(*module*, *env*, *named_modules*)
Bases: *modules.operations.unary_operation.UnaryOperation*

A module that split a given string field from an incoming dataflow according to a regex.

Parameters **module** (*dict*) – The module dict must contain a `field` that specify the column index on which to perform the split operation (ex: `0`) A `delimiter` field indicates the separator (ex: `","`)

Other optional fields are:

- The `reduce` optional field is used to select only one element of the array resulting of the split function (ex: `null`, `-1`, `2`).

check_integrity()
Performs some check on the upstream modules and types when necessary to ensure the integrity of the DAG.

get_out_type()
Returns the output type of the module as a list of strings.

rendered_result()
Returns a pair of strings containing the rendered lines of codes and external classes or objects definitions.

Return type `Tuple[str, str]`

Binary Operations

The abstract class for binary operation modules.

class modules.operations.binary_operation.**BinaryOperation**(*module*, *env*, *named_modules*)
The abstract base module for all binary operations (Operations that take two data flows as input).

Parameters **module** (*dict*) – The module dict must have the two fields `source1` and `source2` that contain the names of the two input flows.

add_to_graph(*graph*)
A method for adding the module to a graphviz graph instance.

Parameters **graph** (*graphviz.dot.Digraph*) – A graphviz Digraph object

rendered_result()
Returns a pair of strings containing the rendered lines of codes and external classes or objects definitions.

Return type `Tuple[str, str]`

Join

The join operation module

class modules.operations.join.**Join**(*module*, *env*, *named_modules*)
Bases: *modules.operations.binary_operation.BinaryOperation*

A module that joins two incoming dataflows on field1 == field2

Parameters **module** (*dict*) – The module dict must contain the `field1` and `field2` fields that correspond to the desired index to make the join on.

Other optional fields are:

- `leftFields` and `rightFields` are lists of integers specifying the indexes to keep in the join's output. Default value is `"all"`. (ex: `[0, 2]`)

check_integrity()
Performs some check on the upstream modules and types when necessary to ensure the integrity of the DAG.

get_out_type()
Returns the output type of the module as a list of strings.

rendered_result()
Returns a pair of strings containing the rendered lines of codes and external classes or objects definitions.

Return type `Tuple[str, str]`

String Similarity

The string similarity operation module

class modules.operations.string_similarity.**StringSimilarity**(*module*, *env*, *named_modules*)
Bases: *modules.operations.binary_operation.BinaryOperation*

A module that compute similarity scores between two string inputs with one of the available soft string matching algorithms.

Warning: Some algorithms compute a distance, some others a similarity score. Besides, some are normalized and some aren't. See the documentation for more details on each algorithm.

The default `algorithm` is the Levenshtein distance, but any one from the following list can be chosen:

- Levenshtein
- NormalizedLevenshtein
- Damerau
- OptimalStringAlignment
- JaroWinkler
- LongestCommonSubsequence
- MetricLCS
- Cosine

All implemetations are from Thibault Debatty's string similarity Java library. See the javadoc for a detailed description of all algorithms

Parameters **module** (*dict*) – The module dict must contain the fields `leftField` and `rightField` that are integers corresponding to the columns that will be compared (ex: `0`).

An `algorithm` field should also be included in module with a string containing the name of the desired algorithm (ex: `"Levenshtein"`).

Other optional fields are:

- `leftOutFields` and `rightOutFields` that are by default set to `"all"` but can be a list of integers that represent the columns on which the result should be projected (ex: [0, 2, 3]).

check_integrity()
Performs some check on the upstream modules and types when necessary to ensure the integrity of the DAG.

get_out_type()
Returns the output type of the module as a list of strings.

rendered_result()
Returns a pair of strings containing the rendered lines of codes and external classes or objects definitions.

Return type Tuple[str, str]

Mention Extraction

The mention link extractor operation module

class modules.operations.extractor_link.**ExtractorLink**(*module*, *env*, *named_modules*)
Bases: *modules.operations.binary_operation.BinaryOperation*

A module that extracts the occurrences of a given field of a data flow into a field of an other data flow. The source extraction will always be the right flow and the target will be the left flow.

Parameters **module** (*dict*) – The module dict must contain sourceExtract and targetExtract with the index of source and target columns.

sourceExtract corresponds to the source1 data flow with text to make extraction from.

targetExtract corresponds to source2 and contains the patterns that will be searched for in the sourceExtract.

check_integrity()
Performs some check on the upstream modules and types when necessary to ensure the integrity of the DAG.

get_out_type()
Returns the output type of the module as a list of strings.

rendered_result()
Returns a pair of strings containing the rendered lines of codes and external classes or objects definitions.

Return type Tuple[str, str]

Jaccard Measure on Sets

The word similarity operation module

class modules.operations.extractor_word_similarity.**ExtractorWordSimilarity**(*module*, *env*, *named_modules*)
Bases: *modules.operations.binary_operation.BinaryOperation*

A module that computes a Jaccard similarity measure on two input sets.

Parameters **module** (*dict*) – The module dict must contain leftField rightField that correspond to the index of the set to be compared in the input flows.

check_integrity()
Performs some check on the upstream modules and types when necessary to ensure the integrity of the DAG.

get_out_type()
Returns the output type of the module as a list of strings.

rendered_result()
Returns a pair of strings containing the rendered lines of codes and external classes or objects definitions.

Return type Tuple[str, str]

Union

The union operation module

class modules.operations.union.**Union**(*module*, *env*, *named_modules*)
Bases: *modules.operations.binary_operation.BinaryOperation*

A module that performs the union of two incoming data flows.

Parameters **module** (*dict*) – The module dict must contain the fields leftField and rightField that are integers corresponding to the columns that will be compared (ex: 0).

check_integrity()
Performs some check on the upstream modules and types when necessary to ensure the integrity of the DAG.

get_out_type()
Returns the output type of the module as a list of strings.

rendered_result()
Returns a pair of strings containing the rendered lines of codes and external classes or objects definitions.

Return type Tuple[str, str]

Outputs

File Output

The base class for file output modules

class modules.outputs.file_output.**FileOutput**(*module*, *env*, *named_modules*)
A file output module has a source and a path to a file.

add_to_graph(*graph*)
A method for adding the module to a graphviz graph instance.

Parameters **graph** (*graphviz.dot.Digraph*) – A graphviz Digraph object

rendered_result()
Returns a pair of strings containing the rendered lines of codes and external classes or objects definitions.

Return type Tuple[str, str]

CSV Output

The CSV output operation module

class modules.outputs.csv_output.**CsvOutput**(*module*, *env*, *named_modules*)
Bases: *modules.outputs.file_output.FileOutput*

A module that writes a dataflow to a CSV flink sink (a csv file)

check_integrity()
Performs some check on the upstream modules and types when necessary to ensure the integrity of the DAG.

get_out_type()
Returns the output type of the module as a list of strings.

rendered_result()
Returns a pair of strings containing the rendered lines of codes and external classes or objects definitions.

Return type Tuple[str, str]

1.3.3 Adding Modules

To create new modules with new functionalities, one can subclass any of the following base classes:

- BaseModule: Works for any new module.
- FileImporter: For extractor modules with files as input.
- UnaryOperation: For modules that do work on one input data flow.
- BinaryOperation: For modules that implement an operation on two separate inputs.
- FileOutput: For output modules with files as output.

When implementing a new module, one should use the following template:

```
class MyModule(BaseModule): # Any of the base classes
    """ Documentation of the module

    Args:
        module (dict): Description of the module dict to
            be passed as argument.
    """
    def __init__(self, module, env: Environment, named_modules):
        super().__init__(module, env, named_modules)

        self.template_path = # Path to template
        self.template = self.env.get_template(self.template_path)

    def rendered_result(self) -> Tuple[str, str]:
        return self.template.render(
            name=self.name,
            # Other arguments
        ), '' # or ext template if applicable

    def get_out_type(self):
        # This function should return the output type of the module
        # as a list of strings.

    def check_integrity(self):
        # This function performs integrity checks if applicable.
```

The module should have a scala template associated with it for generating the corresponding code.

```
// ===== My module {{name}} =====

// Insert code here
val {{name}} = // The Flink Dataset
```

Once the module is defined, it can be added to the rendering engine by adding it to the ModuleMap class directly for example.

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Python Module Index

m

r

Index

S

T

U