Spooq2 Documentation

Release 2.1.0

David Eigenstuhler

Aug 17, 2020

Contents

Spooq is your PySpark based helper library for ETL data ingestion pipeline in Data Lakes.

Extractors, Transformers, and Loaders are independent components which can be plugged-in into a pipeline instance or used separately.

CHAPTER 1

Table of Content

1.1 Installation / Deployment

1.1.1 Build egg file

```
$ cd spooq2
$ python setup.py bdist_egg
```

The output is stored as *dist/Spooq2-<VERSION_NUMBER>-py2.7.egg*

1.1.2 Build zip file

```
$ cd spooq2
$ rm temp.zip
$ zip -r temp.zip src/spooq2
$ mv temp.zip Spooq2_$(grep "__version__" src/spooq2/_version.py | \
    cut -d " " -f 3 | tr -d \").zip
```

The output is stored as *Spooq2-<VERSION_NUMBER>.zip*.

1.1.3 Include pre-build package (egg or zip) with Spark

For Submitting or Launching Spark:

```
$ pyspark --py-files Spooq2-<VERSION_NUMBER>.egg
```

The library still has to be imported in the pyspark application!

Within Running Spark Session:

```
>>> sc.addFile("Spooq2-<VERSION_NUMBER>.egg")
>>> import spooq2
```

1.1.4 Install local repository as package

```
$ cd spooq2
$ python setup.py install
```

1.1.5 Install Spooq2 directly from git

```
$ pip install git+https://github.com/breaka84/spooq@master
```

1.1.6 Development, Testing, and Documenting

Please refer to *Setup for Development, Testing, Documenting*.

1.2 Examples

1.2.1 JSON Files to Partitioned Hive Table

Sample Input Data:

```
{
  "id": 18,
  "guid": "b12b59ba-5c78-4057-a998-469497005c1f",
  "attributes": {
    "first_name": "Jeannette",
    "last_name": "O'Loghlen",
    "gender": "F",
    "email": "gpirri3j@oracle.com",
    "ip_address": "64.19.237.154",
    "university": "",
    "birthday": "1972-05-16T22:17:41Z",
    "friends": [
      {
        "first_name": "Noémie",
        "last_name": "Tibbles",
        "id": 9952
      },
      {
        "first_name": "Bérangère",
        "last_name": null,
        "id": 3391
      },
      {
        "first_name": "Danièle",
        "last_name": null,
        "id": 9637
      },
      {
        "first_name": null,
        "last_name": null,
        "id": 9939
      },
      {
        "first_name": "Anaëlle",
        "last_name": null,
        "id": 18994
```

(continues on next page)

(continued from previous page)

```
      }
    ]
  },
  "meta": {
    "created_at_sec": 1547371284,
    "created_at_ms": 1547204429000,
    "version": 24
  }
}
```

Sample Output Tables

Table 1: Table **"user"**

id	guid	forename	surname	gender	has_email	has_university	created_at
18	"b12b59ba…"	"Jeannette"	"O"Loghlen"	"F"	"1"	NULL	1547204429
…	…	…	…	…	…	…	…

Table 2: Table **"friends_mapping"**

id	guid	friend_id	created_at
18	b12b59ba…	9952	1547204429
18	b12b59ba…	3391	1547204429
18	b12b59ba…	9637	1547204429
18	b12b59ba…	9939	1547204429
18	b12b59ba…	18994	1547204429
…	…	…	…

Application Code for Updating the Users Table

```
from spooq2.pipeline import Pipeline
import spooq2.extractor as E
import spooq2.transformer as T
import spooq2.loader as L

users_mapping = [
    ("id",                  "id",                      "IntegerType"),
    ("guid",                "guid",                    "StringType"),
    ("forename",            "attributes.first_name",   "StringType"),
    ("surename",            "attributes.last_name",    "StringType"),
    ("gender",              "attributes.gender",       "StringType"),
    ("has_email",           "attributes.email",        "StringBoolean"),
    ("has_university",      "attributes.university",   "StringBoolean"),
    ("created_at",          "meta.created_at_ms",      "timestamp_ms_to_s"),
]

users_pipeline = Pipeline()

users_pipeline.set_extractor(E.JSONExtractor(input_path="tests/data/schema_v1/
↪sequenceFiles"))

users_pipeline.add_transformers(
    [
```

(continues on next page)

(continued from previous page)

```
        T.Mapper(mapping=users_mapping),
        T.ThresholdCleaner(
            range_definitions={"created_at": {"min": 0, "max": 1580737513, "default
↪": None}}
        ),
        T.NewestByGroup(group_by="id", order_by="created_at"),
    ]
)

users_pipeline.set_loader(
    L.HiveLoader(
        db_name="users_and_friends",
        table_name="users",
        partition_definitions=[
            {"column_name": "dt", "column_type": "IntegerType", "default_value":␣
↪20200201}
        ],
        repartition_size=10,
    )
)

users_pipeline.execute()
```

Application Code for Updating the Friends_Mapping Table

```
from spooq2.pipeline import Pipeline
import spooq2.extractor as E
import spooq2.transformer as T
import spooq2.loader as L


friends_mapping = [
    ("id",           "id",                     "IntegerType"),
    ("guid",         "guid",                   "StringType"),
    ("friend_id",    "friend.id",              "IntegerType"),
    ("created_at",   "meta.created_at_ms",     "timestamp_ms_to_s"),
]

friends_pipeline = Pipeline()

friends_pipeline.set_extractor(E.JSONExtractor(input_path="tests/data/schema_v1/
↪sequenceFiles"))

friends_pipeline.add_transformers(
    [
        T.NewestByGroup(group_by="id", order_by="meta.created_at_ms"),
        T.Exploder(path_to_array="attributes.friends", exploded_elem_name="friend
↪"),
        T.Mapper(mapping=friends_mapping),
        T.ThresholdCleaner(
            range_definitions={"created_at": {"min": 0, "max": 1580737513, "default
↪": None}}
        ),
    ]
)

friends_pipeline.set_loader(
    L.HiveLoader(
        db_name="users_and_friends",
```

(continues on next page)

(continued from previous page)

```
        table_name="friends_mapping",
        partition_definitions=[
            {"column_name": "dt", "column_type": "IntegerType", "default_value":␣
↪20200201}
        ],
        repartition_size=20,
    )
)

friends_pipeline.execute()
```

Application Code for Updating Both, the Users and Friends_Mapping Table, at once

This script extracts and transforms the common activities for both tables as they share the same input data set. Caching the dataframe avoids redundant processes and reloading when an action is executed (the load step f.e.). This could have been written with pipeline objects as well (by providing the Pipeline an `input_df` and/or `output_df` to bypass extractors and loaders) but would have led to unnecessary verbosity. This example should also show the flexibility of Spooq2 for activities and steps which are not directly supported.

```
import spooq2.extractor as E
import spooq2.transformer as T
import spooq2.loader as L

mapping = [
    ("id",                 "id",                         "IntegerType"),
    ("guid",               "guid",                       "StringType"),
    ("forename",           "attributes.first_name",      "StringType"),
    ("surename",           "attributes.last_name",       "StringType"),
    ("gender",             "attributes.gender",          "StringType"),
    ("has_email",          "attributes.email",           "StringBoolean"),
    ("has_university",     "attributes.university",      "StringBoolean"),
    ("created_at",         "meta.created_at_ms",         "timestamp_ms_to_s"),
    ("friends",            "attributes.friends",         "as_is"),
]

"""Transformations used by both output tables"""
common_df = E.JSONExtractor(input_path="tests/data/schema_v1/sequenceFiles").
↪extract()
common_df = T.Mapper(mapping=mapping).transform(common_df)
common_df = T.ThresholdCleaner(
    range_definitions={"created_at": {"min": 0, "max": 1580737513, "default": None}
↪}
).transform(common_df)
common_df = T.NewestByGroup(group_by="id", order_by="created_at").transform(common_
↪df)
common_df.cache()

"""Transformations for users_and_friends table"""
L.HiveLoader(
    db_name="users_and_friends",
    table_name="users",
    partition_definitions=[
        {"column_name": "dt", "column_type": "IntegerType", "default_value":␣
↪20200201}
    ],
    repartition_size=10,
).load(common_df.drop("friends"))

"""Transformations for friends_mapping table"""
```

(continues on next page)

(continued from previous page)

```
friends_df = T.Exploder(path_to_array="friends", exploded_elem_name="friend").
↪transform(
    common_df
)
friends_df = T.Mapper(
    mapping=[
        ("id",          "id",          "IntegerType"),
        ("guid",        "guid",        "StringType"),
        ("friend_id",   "friend.id",   "IntegerType"),
        ("created_at",  "created_at",  "IntegerType"),
    ]
).transform(friends_df)
L.HiveLoader(
    db_name="users_and_friends",
    table_name="friends_mapping",
    partition_definitions=[
        {"column_name": "dt", "column_type": "IntegerType", "default_value":␣
↪20200201}
    ],
    repartition_size=20,
).load(friends_df)
```

1.3 Extractors

Extractors are used to fetch, extract and convert a source data set into a PySpark DataFrame. Exemplary extraction sources are **JSON Files** on file systems like HDFS, DBFS or EXT4 and relational database systems via **JDBC**.

1.3.1 JSON Files

class JSONExtractor(*input_path=None*, *base_path=None*, *partition=None*)

Bases: *spooq2.extractor.extractor.Extractor*

The JSONExtractor class provides an API to extract data stored as JSON format, deserializes it into a PySpark dataframe and returns it. Currently only single-line JSON files are supported, stored either as textFile or sequenceFile.

Examples

```
>>> from spooq2 import extractor as E
```

```
>>> extractor = E.JSONExtractor(input_path="tests/data/schema_v1/sequenceFiles
↪")
>>> extractor.input_path == "tests/data/schema_v1/sequenceFiles" + "/*"
True
```

```
>>> extractor = E.JSONExtractor(
>>>     base_path="tests/data/schema_v1/sequenceFiles",
>>>     partition="20200201"
>>> )
>>> extractor.input_path == "tests/data/schema_v1/sequenceFiles" + "/20/02/01"␣
↪+ "/*"
True
```

Parameters

- **input_path** (str) – The path from which the JSON files should be loaded ("/*" will be added if omitted)
- **base_path** (str) – Spooq tries to infer the input_path from the base_path and the partition if the input_path is missing.
- **partition** (str or int) – Spooq tries to infer the input_path from the base_path and the partition if the input_path is missing. Only daily partitions in the form of "YYYYMMDD" are supported. e.g., "20200201" => <base_path> + "/20/02/01/*"

Returns The extracted data set as a PySpark DataFrame

Return type pyspark.sql.DataFrame

Raises exceptions.AttributeError – Please define either input_path or base_path and partition

Warning: Currently only single-line JSON files stored as SequenceFiles or TextFiles are supported!

Note: The init method checks which input parameters are provided and derives the final input_path from them accordingly.

If input_path is not None: Cleans input_path and returns it as the final input_path

Elif base_path and partition are not None: Cleans base_path, infers the sub path from the partition and returns the combined string as the final input_path

Else: Raises an exceptions.AttributeError

extract()

This is the Public API Method to be called for all classes of Extractors

Returns Complex PySpark DataFrame deserialized from the input JSON Files

Return type pyspark.sql.DataFrame

1.3.2 JDBC Source

class JDBCExtractor(*jdbc_options*, *cache=True*)

Bases: *spooq2.extractor.extractor.Extractor*

class JDBCExtractorFullLoad(*query*, *jdbc_options*, *cache=True*)

Bases: *spooq2.extractor.jdbc.JDBCExtractor*

Connects to a JDBC Source and fetches the data defined by the provided Query.

Examples

```
>>> import spooq2.extractor as E
>>>
>>> extractor = E.JDBCExtractorFullLoad(
>>>     query="select id, first_name, last_name, gender, created_at test_db.
↪from users",
>>>     jdbc_options={
>>>         "url": "jdbc:postgresql://localhost/test_db",
>>>         "driver": "org.postgresql.Driver",
>>>         "user": "read_only",
>>>         "password": "test123",
```

(continues on next page)

(continued from previous page)

```
>>>     },
>>> )
>>>
>>> extracted_df = extractor.extract()
>>> type(extracted_df)
pyspark.sql.dataframe.DataFrame
```

Parameters

- **query** (str) – Defines the actual query sent to the JDBC Source. This has to be a valid SQL query with respect to the source system (e.g., T-SQL for Microsoft SQL Server).
- **jdbc_options** (dict, optional) –

  A set of parameters to configure the connection to the source:

  - **url** (str) - A JDBC URL of the form jdbc:subprotocol:subname. e.g., jdbc:postgresql://localhost:5432/dbname
  - **driver** (str) - The class name of the JDBC driver to use to connect to this URL.
  - **user** (str) - Username to authenticate with the source database.
  - **password** (str) - Password to authenticate with the source database.

  See pyspark.sql.DataFrameReader.jdbc() and https://spark.apache.org/docs/2.4.3/sql-data-sources-jdbc.html for more information.
- **cache** (bool, defaults to True) – Defines, weather to cache() the dataframe, after it is loaded. Otherwise the Extractor will reload all data from the source system eachtime an action is performed on the DataFrame.

Raises exceptions.AssertionError: – All jdbc_options values need to be present as string variables.

extract()

This is the Public API Method to be called for all classes of Extractors

Returns PySpark dataframe from the input JDBC connection.

Return type pyspark.sql.DataFrame

class JDBCExtractorIncremental(*partition*, *jdbc_options*, *source_table*, *spooq2_values_table*, *spooq2_values_db='spooq2_values'*, *spooq2_values_partition_column='updated_at'*, *cache=True*)

Bases: *spooq2.extractor.jdbc.JDBCExtractor*

Connects to a JDBC Source and fetches the data with respect to boundaries. The boundaries are inferred from the partition to load and logs from previous loads stored in the spooq2_values_table.

Examples

```
>>> import spooq2.extractor as E
>>>
>>> # Boundaries derived from previously logged extractions => ("2020-01-31␣
↪03:29:59", False)
>>>
>>> extractor = E.JDBCExtractorIncremental(
>>>     partition="20200201",
>>>     jdbc_options={
>>>         "url": "jdbc:postgresql://localhost/test_db",
>>>         "driver": "org.postgresql.Driver",
```

(continues on next page)

(continued from previous page)

```
>>>         "user": "read_only",
>>>         "password": "test123",
>>>     },
>>>     source_table="users",
>>>     spooq2_values_table="spooq2_jdbc_log_users",
>>> )
>>>
>>> extractor._construct_query_for_partition(extractor.partition)
select * from users where updated_at > "2020-01-31 03:29:59"
>>>
>>> extracted_df = extractor.extract()
>>> type(extracted_df)
pyspark.sql.dataframe.DataFrame
```

Parameters

- **`partition`** (`int` or `str`) – Partition to extract. Needed for logging the incremental load in the `spooq2_values_table`.
- **`jdbc_options`** (`dict`, optional) –

  A set of parameters to configure the connection to the source:

  - **url** (`str`) - A JDBC URL of the form jdbc:subprotocol:subname. e.g., jdbc:postgresql://localhost:5432/dbname
  - **driver** (`str`) - The class name of the JDBC driver to use to connect to this URL.
  - **user** (`str`) - Username to authenticate with the source database.
  - **password** (`str`) - Password to authenticate with the source database.

  See `pyspark.sql.DataFrameReader.jdbc()` and https://spark.apache.org/docs/2.4.3/sql-data-sources-jdbc.html for more information.
- **`source_table`** (`str`) – Defines the tablename of the source to be loaded from. For example 'purchases'. This is necessary to build the query.
- **`spooq2_values_table`** (`str`) – Defines the Hive table where previous and future loads of a specific source table are logged. This is necessary to derive boundaries for the current partition.
- **`spooq2_values_db`** (`str`, optional) – Defines the Database where the `spooq2_values_table` is stored. Defaults to *'spooq2_values'*.
- **`spooq2_values_partition_column`** (`str`, optional) – The column name which is used for the boundaries. Defaults to *'updated_at'*.
- **`cache`** (`bool`, defaults to `True`) – Defines, weather to `cache()` the dataframe, after it is loaded. Otherwise the Extractor will reload all data from the source system again, if a second action upon the dataframe is performed.

Raises `exceptions.AssertionError:` – All jdbc_options values need to be present as string variables.

`extract()`

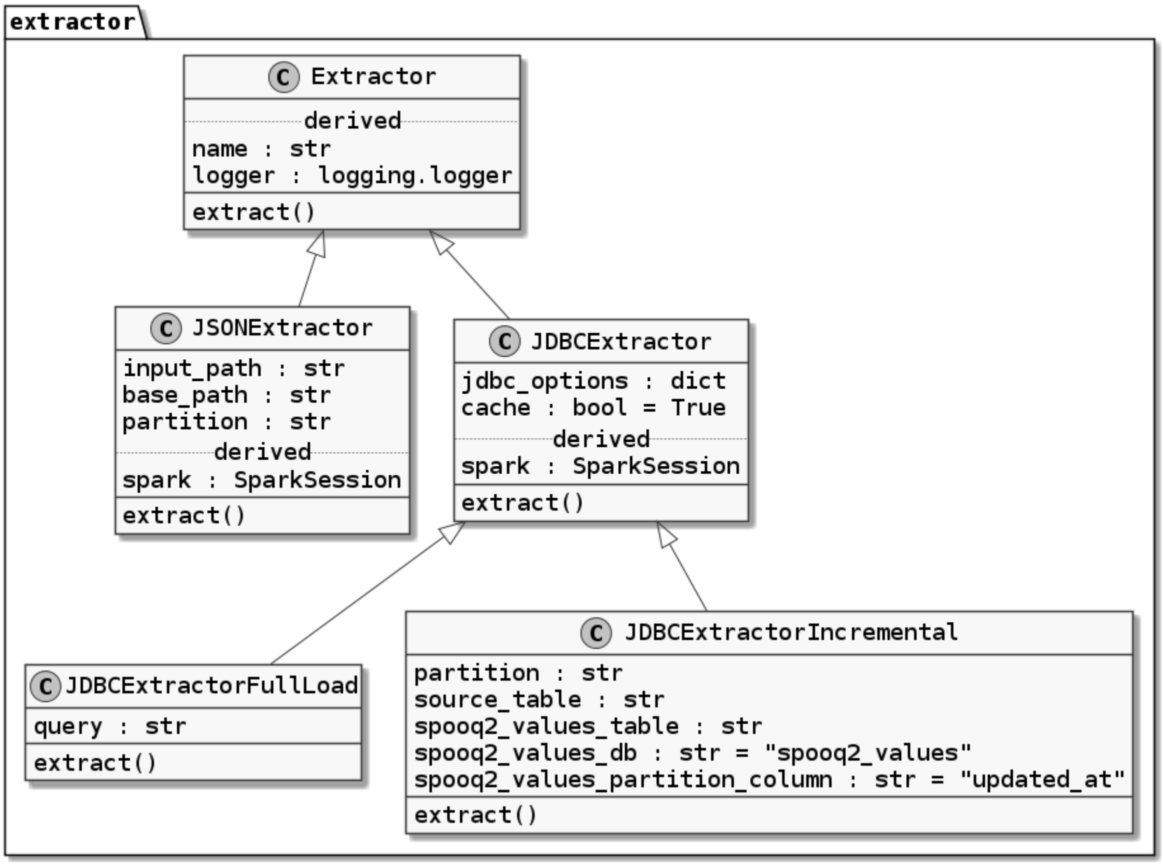
Extracts Data from a Source and converts it into a PySpark DataFrame.

Returns

Return type `pyspark.sql.DataFrame`

Note: This method does not take ANY input parameters. All needed parameters are defined in the initialization of the Extractor Object.

1.3.3 Class Diagram of Extractor Subpackage



1.3.4 Create your own Extractor

Please see the *Create your own Extractor* for further details.

1.4 Transformers

Transformers take a `pyspark.sql.DataFrame` as an input, transform it accordingly and return a PySpark DataFrame.

Each Transformer class has to have a *transform* method which takes no arguments and returns a PySpark DataFrame.

Possible transformation methods can be **Selecting the most up to date record by id**, **Exploding an array**, **Filter (on an exploded array)**, **Apply basic threshold cleansing** or **Map the incoming DataFrame to at provided structure**.

1.4.1 Exploder

class Exploder(*path_to_array='included'*, *exploded_elem_name='elem'*)

Bases: *spooq2.transformer.transformer.Transformer*

Explodes an array within a DataFrame and drops the column containing the source array.

Examples

```
>>> transformer = Exploder(
>>>     path_to_array="attributes.friends",
>>>     exploded_elem_name="friend",
>>> )
```

Parameters

- **path_to_array** (str, (Defaults to 'included')) – Defines the Column Name / Path to the Array. Dropping nested columns is not supported. Although, you can still explode them.
- **exploded_elem_name** (str, (Defaults to 'elem')) – Defines the column name the exploded column will get. This is important to know how to access the Field afterwards. Writing nested columns is not supported. The output column has to be first level.

Warning: Support for nested column:

path_to_array: PySpark cannot drop a field within a struct. This means the specific field can be referenced and therefore exploded, but not dropped.

exploded_elem_name: If you (re)name a column in the dot notation, is creates a first level column, just with a dot its name. To create a struct with the column as a field you have to redefine the structure or use a UDF.

Note: The `explode()` method of Spark is used internally.

Note: The size of the resulting DataFrame is not guaranteed to be equal to the Input DataFrame!

transform(*input_df*)

Performs a transformation on a DataFrame.

Parameters **input_df** (pyspark.sql.DataFrame) – Input DataFrame

Returns Transformed DataFrame.

Return type pyspark.sql.DataFrame

Note: This method does only take the Input DataFrame as a parameters. All other needed parameters are defined in the initialization of the Transformator Object.

1.4.2 Sieve (Filter)

class Sieve(*filter_expression*)

Bases: *spooq2.transformer.transformer.Transformer*

Filters rows depending on provided filter expression. Only records complying with filter condition are kept.

Examples

```
>>> transformer = T.Sieve(filter_expression=""" attributes.last_name rlike "^.
↪{7}$" """)
```

```
>>> transformer = T.Sieve(filter_expression=""" lower(gender) = "f" """)
```

Parameters **filter_expression** (`str`) – A valid PySpark SQL expression which returns a boolean

Raises `exceptions.ValueError` – filter_expression has to be a valid (Spark)SQL expression provided as a string

Note: The `filter()` method is used internally.

Note: The Size of the resulting DataFrame is not guaranteed to be equal to the Input DataFrame!

transform(*input_df*)

Performs a transformation on a DataFrame.

Parameters **input_df** (`pyspark.sql.DataFrame`) – Input DataFrame

Returns Transformed DataFrame.

Return type `pyspark.sql.DataFrame`

Note: This method does only take the Input DataFrame as a parameters. All other needed parameters are defined in the initialization of the Transformator Object.

1.4.3 Mapper

Class

class Mapper(*mapping*, *ignore_missing_columns=True*)

Bases: *spooq2.transformer.transformer.Transformer*

Constructs and applies a PySpark SQL expression, based on the provided mapping.

Examples

```
>>> mapping = [
>>>     ('id',              'data.relationships.food.data.id',     'StringType
↪'),
>>>     ('message_id',      'data.id',                             'StringType
↪'),
>>>     ('type',            'data.relationships.food.data.type',   'StringType
↪'),
>>>     ('created_at',      'elem.attributes.created_at',          'timestamp_
↪ms_to_s'),
>>>     ('updated_at',      'elem.attributes.updated_at',          'timestamp_
↪ms_to_s'),
>>>     ('deleted_at',      'elem.attributes.deleted_at',          'timestamp_
↪ms_to_s'),
>>>     ('brand',           'elem.attributes.brand',               'StringType
↪')
>>> ]
>>> transformer = Mapper(mapping=mapping)
>>> mapping = [
>>>     ('id',           'data.relationships.food.data.id',   'StringType'),
>>>     ('updated_at',   'elem.attributes.updated_at',        'timestamp_ms_to_s
↪'),
>>>     ('deleted_at',   'elem.attributes.deleted_at',        'timestamp_ms_to_s
↪'),
>>>     ('name',         'elem.attributes.name',              'array')
>>> ]
>>> transformer = Mapper(mapping=mapping)
```

Parameters

- **`mapping`** (`list` of `tuple` containing three `str`) – This is the main parameter for this transformation. It essentially gives information about the column names for the output DataFrame, the column names (paths) from the input DataFrame, and their data types. Custom data types are also supported, which can clean, pivot, anonymize, . . . the data itself. Please have a look at the *`spooq2.transformer.mapper_custom_data_types`* module for more information.
- **`ignore_missing_columns`** (`bool`, Defaults to True) – Specifies if the mapping transformation should raise an exception if a referenced input column is missing in the provided DataFrame.

Note: Let's talk about Mappings:

The mapping should be a list of tuples which are containing all information per column.

- **Column Name** [`str`] Sets the name of the column in the resulting output DataFrame.
- **Source Path / Name** [`str`] Points to the name of the column in the input DataFrame. If the input is a flat DataFrame, it will essentially be the column name. If it is of complex type, it will point to the path of the actual value. For example: `data.relationships.sample.data.id`, where id is the value we want.
- **DataType** [`str`] DataTypes can be types from `pyspark.sql.types`, selected custom datatypes or injected, ad-hoc custom datatypes. The datatype will be interpreted as a PySpark built-in if it is a member of `pyspark.sql.types`. If it is not an importable PySpark data type, a method to construct the statement will be called by the data type's name.

Note: The available input columns can vary from batch to batch if you use schema inference (f.e. on json data) for the extraction. Ignoring missing columns on the input DataFrame is highly encouraged in this case. Although, if you have tight control over the structure of the extracted DataFrame, setting *ignore_missing_columns* to True is advised as it can uncover typos and bugs.

Note: Please see *`spooq2.transformer.mapper_custom_data_types`* for all available custom data types and how to inject your own.

Note: Attention: Decimal is NOT SUPPORTED by Hive! Please use Double instead!

`transform`(*input_df*)

Performs a transformation on a DataFrame.

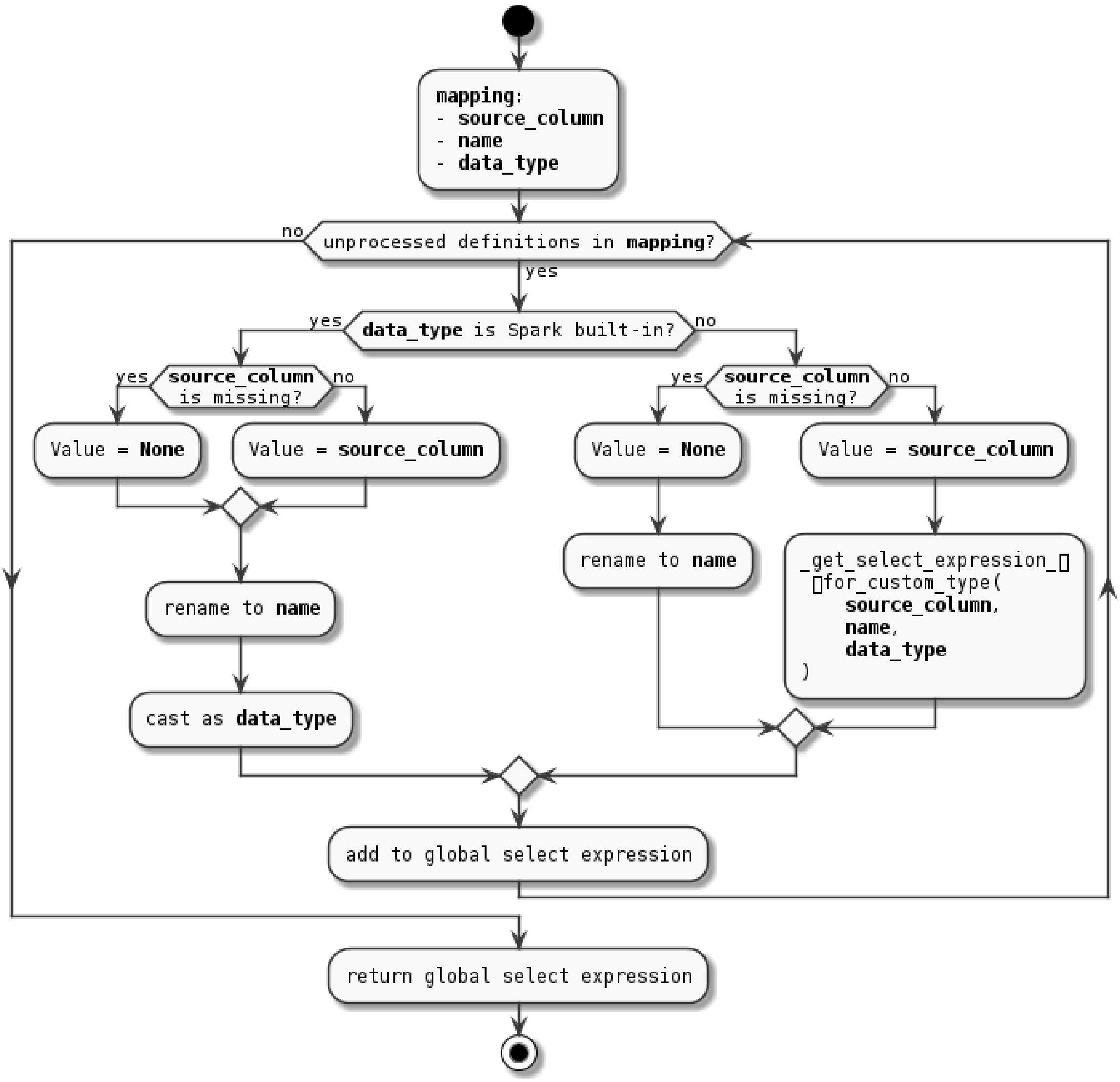
Parameters **`input_df`** (`pyspark.sql.DataFrame`) – Input DataFrame

Returns Transformed DataFrame.

Return type `pyspark.sql.DataFrame`

Note: This method does only take the Input DataFrame as a parameters. All other needed parameters are defined in the initialization of the Transformator Object.

Activity Diagram


mapping:
- source_column
- name
- data_type
unprocessed definitions in mapping?
no
yes
data_type is Spark built-in?
yes
no
source_column is missing?
yes
no
Value = None
Value = source_column
rename to name
cast as data_type
source_column is missing?
yes
no
Value = None
Value = source_column
rename to name
_get_select_expression_□ □for_custom_type(source_column, name, data_type)
add to global select expression
return global select expression

Custom Mapping Methods

This is a collection of module level methods to construct a specific PySpark DataFrame query for custom defined data types.

These methods are not meant to be called directly but via the the *Mapper* transformer. Please see that particular class on how to apply custom data types.

For injecting your **own custom data types**, please have a visit to the *add_custom_data_type()* method!

add_custom_data_type (*function_name*, *func*)
Registers a custom data type in runtime to be used with the *Mapper* transformer.

Example

```
>>> import spooq2.transformer.mapper_custom_data_types as custom_types
>>> import spooq2.transformer as T
>>> from pyspark.sql import Row, functions as F, types as sql_types
```

```
>>> def hello_world(source_column, name):
>>>     "A UDF (User Defined Function) in Python"
>>>     def _to_hello_world(col):
>>>         if not col:
>>>             return None
>>>         else:
>>>             return "Hello World"
>>>
>>>     udf_hello_world = F.udf(_to_hello_world, sql_types.StringType())
>>>     return udf_hello_world(source_column).alias(name)
>>>
>>> input_df = spark.createDataFrame(
>>>     [Row(hello_from=u'tsivorn1@who.int'),
>>>      Row(hello_from=u''),
>>>      Row(hello_from=u'gisaksen4@skype.com')]
>>> )
>>>
>>> custom_types.add_custom_data_type(function_name="hello_world", func=hello_
↪world)
>>> transformer = T.Mapper(mapping=[("hello_who", "hello_from", "hello_world
↪")])
>>> df = transformer.transform(input_df)
>>> df.show()
+-----------+
|  hello_who|
+-----------+
|Hello World|
|       null|
|Hello World|
+-----------+
```

```
>>> def first_and_last_name(source_column, name):
>>>     "A PySpark SQL expression referencing multiple columns"
>>>     return F.concat_ws("_", source_column, F.col("attributes.last_name")).
↪alias(name)
>>>
>>> custom_types.add_custom_data_type(function_name="full_name", func=first_
↪and_last_name)
>>>
>>> transformer = T.Mapper(mapping=[
>>>     ("first_name", "attributes.first_name", "StringType"),
>>>     ("last_name",  "attributes.last_name",  "StringType"),
>>>     ("full_name",  "attributes.first_name", "full_name"),
>>> ])
```

Parameters

- **function_name** (str) – The name of your custom data type
- **func** (*compatible function*) – The PySpark dataframe function which will be called on a column, defined in the mapping of the Mapper class. Required input parameters are source_column and name. Please see the note about required input parameter of custom data types for more information!

Note: Required input parameter of custom data types:

source_column (`pyspark.sql.Column`) **- This is where your logic will be applied.** The mapper transformer takes care of calling this method with the right column so you can just handle it like an object which you would get from `df["some_attribute"]`.

name (`str`) **- The name how the resulting column will be named. Nested attributes are not** supported. The Mapper transformer takes care of calling this method with the right column name.

`_get_select_expression_for_custom_type` (*source_column*, *name*, *data_type*)
Internal method for calling functions dynamically

`_generate_select_expression_for_as_is` (*source_column*, *name*)
alias for *_generate_select_expression_without_casting*

`_generate_select_expression_for_keep` (*source_column*, *name*)
alias for *_generate_select_expression_without_casting*

`_generate_select_expression_for_no_change` (*source_column*, *name*)
alias for *_generate_select_expression_without_casting*

`_generate_select_expression_without_casting` (*source_column*, *name*)
Returns a column without casting. This is especially useful if you need to keep a complex data type, like an array, list or a struct.

```
>>> from spooq2.transformer import Mapper
>>>
>>> input_df.head(3)
[Row(friends=[Row(first_name=None, id=3993, last_name=None), Row(first_name=u
↪'Ruò', id=17484, last_name=u'Trank')]),
 Row(friends=[]),
 Row(friends=[Row(first_name=u'Daphnée', id=16707, last_name=u'Lyddiard'),␣
↪Row(first_name=u'Adélaïde', id=17429, last_name=u'Wisdom')])]
>>> mapping = [("my_friends", "friends", "as_is")]
>>> output_df = Mapper(mapping).transform(input_df)
>>> output_df.head(3)
[Row(my_friends=[Row(first_name=None, id=3993, last_name=None), Row(first_
↪name=u'Ruò', id=17484, last_name=u'Trank')]),
 Row(my_friends=[]),
 Row(my_friends=[Row(first_name=u'Daphnée', id=16707, last_name=u'Lyddiard'),␣
↪Row(first_name=u'Adélaïde', id=17429, last_name=u'Wisdom')])]
```

`_generate_select_expression_for_json_string` (*source_column*, *name*)
Returns a column as json compatible string. Nested hierarchies are supported. The unicode representation of a column will be returned if an error occurs.

Example

```
>>> from spooq2.transformer import Mapper
>>>
>>> input_df.head(3)
[Row(friends=[Row(first_name=None, id=3993, last_name=None), Row(first_name=u
↪'Ruò', id=17484, last_name=u'Trank')]),
 Row(friends=[]),
 Row(friends=[Row(first_name=u'Daphnée', id=16707, last_name=u'Lyddiard'),␣
↪Row(first_name=u'Adélaïde', id=17429, last_name=u'Wisdom')])]    >>> mapping␣
↪= [("friends_json", "friends", "json_string")]
>>> mapping = [("friends_json", "friends", "json_string")]
>>> output_df = Mapper(mapping).transform(input_df)
>>> output_df.head(3)
[Row(friends_json=u'[{"first_name": null, "last_name": null, "id": 3993}, {
↪"first_name": "Ru\u00f2", "last_name": "Trank", "id": 17484}]'),
```

(continues on next page)

(continued from previous page)

```
 Row(friends_json=None),
 Row(friends_json=u'[{"first_name": "Daphn\u00e9e", "last_name": "Lyddiard",
↪"id": 16707}, {"first_name": "Ad\u00e91a\u00efde", "last_name": "Wisdom", "id
↪": 17429}]')]
```

_generate_select_expression_for_timestamp_ms_to_ms(*source_column*, *name*)

This Constructor is used for unix timestamps. The values are cleaned next to casting and renaming. If the values are not between *01.01.1970* and *31.12.2099*, NULL will be returned. Cast to `pyspark.sql.types.LongType`

Example

```
>>> from pyspark.sql import Row
>>> from spooq2.transformer import Mapper
>>>
>>> input_df = spark.createDataFrame([
>>>     Row(time_sec=1581540839000),  # 02/12/2020 @ 8:53pm (UTC)
>>>     Row(time_sec=-4887839000),    # Invalid!
>>>     Row(time_sec=4737139200000)   # 02/12/2120 @ 12:00am (UTC)
>>> ])
>>>
>>> mapping = [("unix_ts", "time_sec", "timestamp_ms_to_ms")]
>>> output_df = Mapper(mapping).transform(input_df)
>>> output_df.head(3)
[Row(unix_ts=1581540839000), Row(unix_ts=None), Row(unix_ts=None)]
```

Note: *input* in **milli seconds** *output* in **milli seconds**

_generate_select_expression_for_timestamp_ms_to_s(*source_column*, *name*)

This Constructor is used for unix timestamps. The values are cleaned next to casting and renaming. If the values are not between *01.01.1970* and *31.12.2099*, NULL will be returned. Cast to `pyspark.sql.types.LongType`

Example

```
>>> from pyspark.sql import Row
>>> from spooq2.transformer import Mapper
>>>
>>> input_df = spark.createDataFrame([
>>>     Row(time_sec=1581540839000),  # 02/12/2020 @ 8:53pm (UTC)
>>>     Row(time_sec=-4887839000),    # Invalid!
>>>     Row(time_sec=4737139200000)   # 02/12/2120 @ 12:00am (UTC)
>>> ])
>>>
>>> mapping = [("unix_ts", "time_sec", "timestamp_ms_to_s")]
>>> output_df = Mapper(mapping).transform(input_df)
>>> output_df.head(3)
[Row(unix_ts=1581540839), Row(unix_ts=None), Row(unix_ts=None)]
```

Note: *input* in **milli seconds** *output* in **seconds**

_generate_select_expression_for_timestamp_s_to_ms(*source_column*, *name*)

This Constructor is used for unix timestamps. The values are cleaned next to casting and renaming. If the values are not between *01.01.1970* and *31.12.2099*, NULL will be returned. Cast to `pyspark.sql.types.LongType`

Example

```
>>> from pyspark.sql import Row
>>> from spooq2.transformer import Mapper
>>>
>>> input_df = spark.createDataFrame([
>>>     Row(time_sec=1581540839),   # 02/12/2020 @ 8:53pm (UTC)
>>>     Row(time_sec=-4887839),     # Invalid!
>>>     Row(time_sec=4737139200)    # 02/12/2120 @ 12:00am (UTC)
>>> ])
>>>
>>> mapping = [("unix_ts", "time_sec", "timestamp_s_to_ms")]
>>> output_df = Mapper(mapping).transform(input_df)
>>> output_df.head(3)
[Row(unix_ts=1581540839000), Row(unix_ts=None), Row(unix_ts=None)]
```

Note: *input* in **seconds** *output* in **milli seconds**

`_generate_select_expression_for_timestamp_s_to_s`(*source_column*, *name*)

This Constructor is used for unix timestamps. The values are cleaned next to casting and renaming. If the values are not between *01.01.1970* and *31.12.2099*, NULL will be returned. Cast to `pyspark.sql.types.LongType`

Example

```
>>> from pyspark.sql import Row
>>> from spooq2.transformer import Mapper
>>>
>>> input_df = spark.createDataFrame([
>>>     Row(time_sec=1581540839),   # 02/12/2020 @ 8:53pm (UTC)
>>>     Row(time_sec=-4887839),     # Invalid!
>>>     Row(time_sec=4737139200)    # 02/12/2120 @ 12:00am (UTC)
>>> ])
>>>
>>> mapping = [("unix_ts", "time_sec", "timestamp_s_to_ms")]
>>> output_df = Mapper(mapping).transform(input_df)
>>> output_df.head(3)
[Row(unix_ts=1581540839), Row(unix_ts=None), Row(unix_ts=None)]
```

Note: *input* in **seconds** *output* in **seconds**

`_generate_select_expression_for_StringNull`(*source_column*, *name*)

Used for Anonymizing. Input values will be ignored and replaced by NULL, Cast to `pyspark.sql.types.StringType`

Example

```
>>> from pyspark.sql import Row
>>> from spooq2.transformer import Mapper
>>>
>>> input_df = spark.createDataFrame(
>>>     [Row(email=u'tsivorn1@who.int'),
>>>      Row(email=u''),
>>>      Row(email=u'gisaksen4@skype.com')]
>>> )
```

(continues on next page)

(continued from previous page)

```
>>>
>>> mapping = [("email", "email", "StringNull")]
>>> output_df = Mapper(mapping).transform(input_df)
>>> output_df.head(3)
[Row(email=None), Row(email=None), Row(email=None)]
```

_generate_select_expression_for_IntNull(*source_column*, *name*)

Used for Anonymizing. Input values will be ignored and replaced by NULL, Cast to `pyspark.sql.types.IntegerType`

Example

```
>>> from pyspark.sql import Row
>>> from spooq2.transformer import Mapper
>>>
>>> input_df = spark.createDataFrame(
>>>     [Row(facebook_id=3047288),
>>>      Row(facebook_id=0),
>>>      Row(facebook_id=57815)]
>>> )
>>>
>>> mapping = [("facebook_id", "facebook_id", "IntNull")]
>>> output_df = Mapper(mapping).transform(input_df)
>>> output_df.head(3)
[Row(facebook_id=None), Row(facebook_id=None), Row(facebook_id=None)]
```

_generate_select_expression_for_StringBoolean(*source_column*, *name*)

Used for Anonymizing. The column's value will be replaced by *"1"* if it is:

- not NULL and
- not an empty string

Example

```
>>> from pyspark.sql import Row
>>> from spooq2.transformer import Mapper
>>>
>>> input_df = spark.createDataFrame(
>>>     [Row(email=u'tsivorn1@who.int'),
>>>      Row(email=u''),
>>>      Row(email=u'gisaksen4@skype.com')]
>>> )
>>>
>>> mapping = [("email", "email", "StringBoolean")]
>>> output_df = Mapper(mapping).transform(input_df)
>>> output_df.head(3)
[Row(email=u'1'), Row(email=None), Row(email=u'1')]
```

_generate_select_expression_for_IntBoolean(*source_column*, *name*)

Used for Anonymizing. The column's value will be replaced by *1* if it contains a non-NULL value.

Example

```
>>> from pyspark.sql import Row
>>> from spooq2.transformer import Mapper
>>>
```

(continues on next page)

(continued from previous page)

```
>>> input_df = spark.createDataFrame(
>>>     [Row(facebook_id=3047288),
>>>      Row(facebook_id=0),
>>>      Row(facebook_id=None)]
>>> )
>>>
>>> mapping = [("facebook_id", "facebook_id", "IntBoolean")]
>>> output_df = Mapper(mapping).transform(input_df)
>>> output_df.head(3)
[Row(facebook_id=1), Row(facebook_id=1), Row(facebook_id=None)]
```

Note: *0* (zero) or negative numbers are still considered as valid values and therefore converted to *1*.

`_generate_select_expression_for_TimestampMonth`(*source_column*, *name*)

Used for Anonymizing. Can be used to keep the age but obscure the explicit birthday. This custom datatype requires a `pyspark.sql.types.TimestampType` column as input. The datetime value will be set to the first day of the month.

Example

```
>>> from pyspark.sql import Row
>>> from datetime import datetime
>>> from spooq2.transformer import Mapper
>>>
>>> input_df = spark.createDataFrame(
>>>     [Row(birthday=datetime(2019, 2, 9, 2, 45)),
>>>      Row(birthday=None),
>>>      Row(birthday=datetime(1988, 1, 31, 8))]
>>> )
>>>
>>> mapping = [("birthday", "birthday", "TimestampMonth")]
>>> output_df = Mapper(mapping).transform(input_df)
>>> output_df.head(3)
[Row(birthday=datetime.datetime(2019, 2, 1, 0, 0)),
 Row(birthday=None),
 Row(birthday=datetime.datetime(1988, 1, 1, 0, 0))]
```

`_generate_select_expression_for_meters_to_cm`(*source_column*, *name*)

Convert meters to cm and cast the result to an IntegerType.

Example

```
>>> from pyspark.sql import Row
>>> from spooq2.transformer import Mapper
>>>
>>> input_df = spark.createDataFrame(
>>>     [Row(size_in_m=1.80),
>>>      Row(size_in_m=1.65),
>>>      Row(size_in_m=2.05)]
>>> )
>>>
>>> mapping = [("size_in_cm", "size_in_m", "meters_to_cm")]
>>> output_df = Mapper(mapping).transform(input_df)
>>> output_df.head(3)
[Row(size_in_cm=180),
```

(continues on next page)

(continued from previous page)

```
Row(size_in_cm=165),
Row(size_in_cm=205)]
```

_generate_select_expression_for_unix_timestamp_ms_to_spark_timestamp(*source_column*, *name*)

Convert unix timestamps in milliseconds to a Spark TimeStampType. It is assumed that the timezone is already set to UTC in spark / java to avoid implicit timezone conversions.

Example

```
>>> from pyspark.sql import Row
>>> from spooq2.transformer import Mapper
>>>
>>> input_df = spark.createDataFrame(
>>>     [Row(unix_timestamp_in_ms=1591627696951),
>>>      Row(unix_timestamp_in_ms=1596812952000),
>>>      Row(unix_timestamp_in_ms=946672200000)]
>>> )
>>>
>>> mapping = [("spark_timestamp", "unix_timestamp_in_ms", "unix_timestamp_ms_
↪to_spark_timestamp")]
>>> output_df = Mapper(mapping).transform(input_df)
>>> output_df.head(3)
[Row(spark_timestamp=datetime.datetime(2020, 6, 8, 16, 48, 16, 951000)),
 Row(spark_timestamp=datetime.datetime(2020, 8, 7, 17, 9, 12)),
 Row(spark_timestamp=datetime.datetime(1999, 12, 31, 21, 30))]
```

_generate_select_expression_for_extended_string_to_int(*source_column*, *name*)

More robust conversion from StringType to IntegerType. Is able to additionally handle (compared to implicit Spark conversion):

- Preceding whitespace
- Trailing whitespace
- Preceeding and trailing whitespace
- underscores as thousand separators

Hint: Please have a look at the tests to get a better feeling how it behaves under tests/unit/transformer/test_mapper_custom_data_types.py::TestConversionsFromString

Example

```
>>> from spooq2.transformer import Mapper
>>>
>>> input_df.head(3)
[Row(input_string="  123456 "),
 Row(input_string="Hello"),
 Row(input_string="123_456")]
>>> mapping = [("output_value", "input_string", "extended_string_to_int")]
>>> output_df = Mapper(mapping).transform(input_df)
>>> output_df.head(3)
[Row(input_string=123456),
 Row(input_string=None),
 Row(input_string=123456)]
```

_generate_select_expression_for_extended_string_to_long(*source_column*, *name*)

More robust conversion from StringType to LongType. Is able to additionally handle (compared to implicit Spark conversion):

- Preceding whitespace
- Trailing whitespace
- Preceeding and trailing whitespace
- underscores as thousand separators

Hint: Please have a look at the tests to get a better feeling how it behaves under tests/unit/transformer/test_mapper_custom_data_types.py::TestConversionsFromString

Example

```
>>> from spooq2.transformer import Mapper
>>>
>>> input_df.head(3)
[Row(input_string="  21474836470 "),
 Row(input_string="Hello"),
 Row(input_string="21_474_836_470")]
>>> mapping = [("output_value", "input_string", "extended_string_to_long")]
>>> output_df = Mapper(mapping).transform(input_df)
>>> output_df.head(3)
[Row(input_string=21474836470),
 Row(input_string=None),
 Row(input_string=21474836470)]
```

_generate_select_expression_for_extended_string_to_float(*source_column*, *name*)

More robust conversion from StringType to FloatType. Is able to additionally handle (compared to implicit Spark conversion):

- Preceding whitespace
- Trailing whitespace
- Preceeding and trailing whitespace
- underscores as thousand separators

Hint: Please have a look at the tests to get a better feeling how it behaves under tests/unit/transformer/test_mapper_custom_data_types.py::TestConversionsFromString

Example

```
>>> from spooq2.transformer import Mapper
>>>
>>> input_df.head(3)
[Row(input_string="  836470.819 "),
 Row(input_string="Hello"),
 Row(input_string="836_470.819")]
>>> mapping = [("output_value", "input_string", "extended_string_to_float")]
>>> output_df = Mapper(mapping).transform(input_df)
>>> output_df.head(3)
[Row(input_string=836470.819),
```

(continues on next page)

(continued from previous page)

```
 Row(input_string=None),
 Row(input_string=836470.819)]
```

_generate_select_expression_for_extended_string_to_double(*source_column*, *name*)

More robust conversion from StringType to DoubleType. Is able to additionally handle (compared to implicit Spark conversion):

- Preceding whitespace
- Trailing whitespace
- Preceeding and trailing whitespace
- underscores as thousand separators

Hint: Please have a look at the tests to get a better feeling how it behaves under tests/unit/transformer/test_mapper_custom_data_types.py::TestConversionsFromString

Example

```
>>> from spooq2.transformer import Mapper
>>>
>>> input_df.head(3)
[Row(input_string="  21474838464.70 "),
 Row(input_string="Hello"),
 Row(input_string="21_474_838_464.70")]
>>> mapping = [("output_value", "input_string", "extended_string_to_double")]
>>> output_df = Mapper(mapping).transform(input_df)
>>> output_df.head(3)
[Row(input_string=21474838464.7),
 Row(input_string=None),
 Row(input_string=21474838464.70)]
```

_generate_select_expression_for_extended_string_to_boolean(*source_column*, *name*)

More robust conversion from StringType to BooleanType. Is able to additionally handle (compared to implicit Spark conversion):

- Preceding whitespace
- Trailing whitespace
- Preceeding and trailing whitespace

Hint: Please have a look at the tests to get a better feeling how it behaves under tests/unit/transformer/test_mapper_custom_data_types.py::TestConversionsFromString

Example

```
>>> from spooq2.transformer import Mapper
>>>
>>> input_df.head(3)
[Row(input_string="  true "),
 Row(input_string="0"),
 Row(input_string="y")]
>>> mapping = [("output_value", "input_string", "extended_string_to_boolean")]
```

(continues on next page)

(continued from previous page)

```
>>> output_df = Mapper(mapping).transform(input_df)
>>> output_df.head(3)
[Row(input_string=True),
 Row(input_string=False),
 Row(input_string=True)]
```

_generate_select_expression_for_extended_string_to_timestamp(*source_column*, *name*)

More robust conversion from StringType to TimestampsType. It is assumed that the timezone is already set to UTC in spark / java to avoid implicit timezone conversions. Is able to additionally handle (compared to implicit Spark conversion):

- Unix timestamps in seconds
- Preceding whitespace
- Trailing whitespace
- Preceeding and trailing whitespace

Hint: Please have a look at the tests to get a better feeling how it behaves under tests/unit/transformer/test_mapper_custom_data_types.py::TestConversionsFromString

Example

```
>>> from spooq2.transformer import Mapper
>>>
>>> input_df.head(3)
[Row(input_string="2020-08-12T12:43:14+0000"),
 Row(input_string="1597069446"),
 Row(input_string="2020-08-12")]
>>> mapping = [("output_value", "input_string", "extended_string_to_timestamp
↪")]
>>> output_df = Mapper(mapping).transform(input_df)
>>> output_df.head(3)
[Row(input_string=datetime.datetime(2020, 8, 12, 12, 43, 14)),
 Row(input_string=datetime.datetime(2020, 8, 10, 14, 24, 6)),
 Row(input_string=datetime.datetime(2020, 8, 12, 0, 0, 0))]
```

1.4.4 Threshold-based Cleaner

class ThresholdCleaner(*thresholds={}*)

Bases: *spooq2.transformer.transformer.Transformer*

Sets outiers within a DataFrame to a default value. Takes a dictionary with valid value ranges for each column to be cleaned.

Example

```
>>> transformer = ThresholdCleaner(
>>>     thresholds={
>>>         "created_at": {
>>>             "min": 0,
>>>             "max": 1580737513,
>>>             "default": None
>>>         },
```

(continues on next page)

(continued from previous page)

```
>>>         "size_cm": {
>>>             "min": 70,
>>>             "max": 250,
>>>             "default": None
>>>         },
>>>     }
>>> )
```

Parameters **thresholds** (dict) – Dictionary containing column names and respective valid ranges

Returns The transformed DataFrame

Return type pyspark.sql.DataFrame

Raises exceptions.ValueError – Threshold-based cleaning only supports Numeric, Date and Timestamp Types! Column with name: {col_name} and type of: {col_type} was provided

Warning: Only Numeric, TimestampType, and DateType data types are supported!

transform (*input_df*)

Performs a transformation on a DataFrame.

Parameters **input_df** (pyspark.sql.DataFrame) – Input DataFrame

Returns Transformed DataFrame.

Return type pyspark.sql.DataFrame

Note: This method does only take the Input DataFrame as a parameters. All other needed parameters are defined in the initialization of the Transformator Object.

1.4.5 Newest by Group (Most current record per ID)

class NewestByGroup (*group_by=['id'], order_by=['updated_at', 'deleted_at']*)

Bases: *spooq2.transformer.transformer.Transformer*

Groups, orders and selects first element per group.

Example

```
>>> transformer = NewestByGroup(
>>>     group_by=["first_name", "last_name"],
>>>     order_by=["created_at_ms", "version"]
>>> )
```

Parameters

- **group_by** (str or list of str, (Defaults to ['id'])) – List of attributes to be used within the Window Function as Grouping Arguments.
- **order_by** (str or list of str, (Defaults to ['updated_at', 'deleted_at'])) – List of attributes to be used within the Window Function as Ordering Arguments. All columns will be sorted in **descending** order.

Raises `exceptions.AttributeError` – If any Attribute in `group_by` or `order_by` is not contained in the input DataFrame.

Note: PySpark's `Window` function is used internally The first row (`row_number()`) per window will be selected and returned.

transform (*input_df*)
Performs a transformation on a DataFrame.

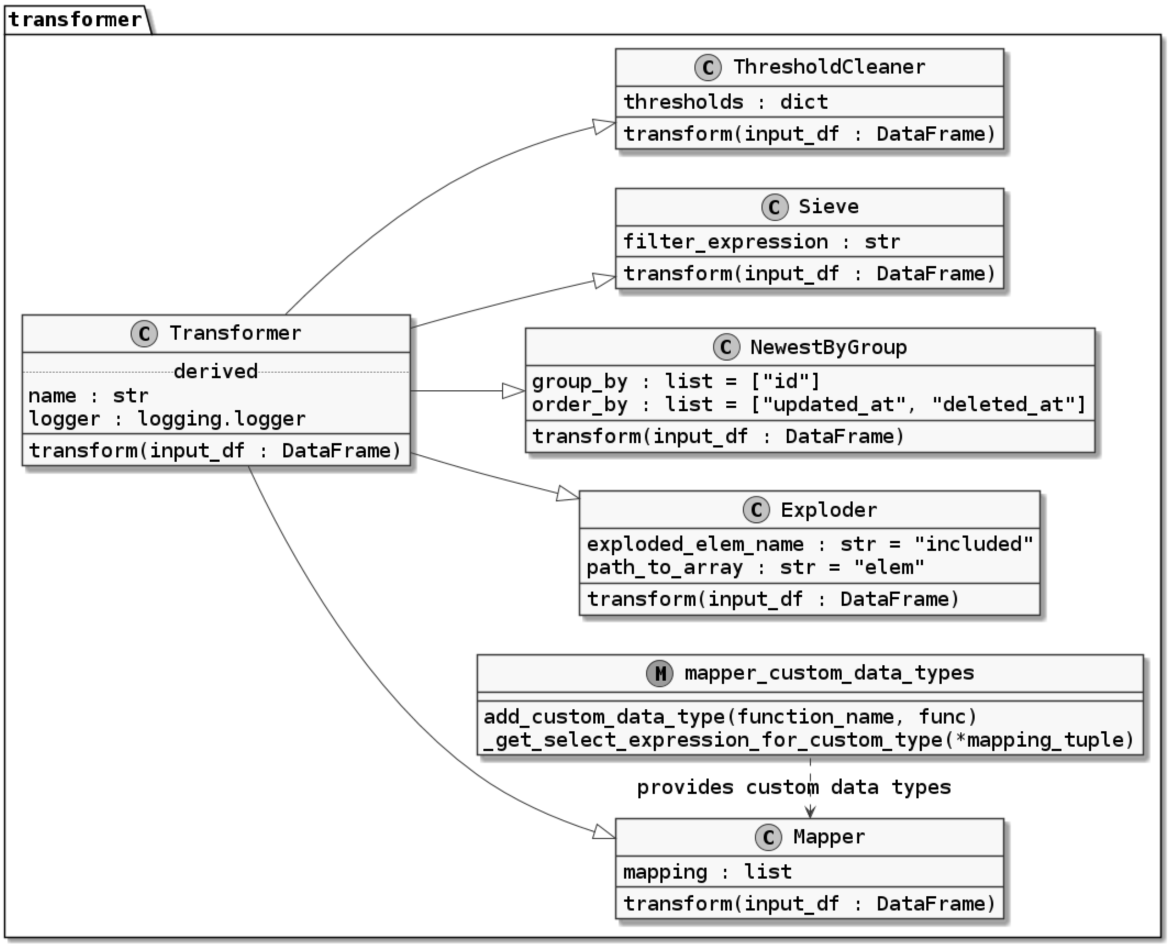
Parameters **input_df** (`pyspark.sql.DataFrame`) – Input DataFrame

Returns Transformed DataFrame.

Return type `pyspark.sql.DataFrame`

Note: This method does only take the Input DataFrame as a parameters. All other needed parameters are defined in the initialization of the Transformator Object.

1.4.6 Class Diagram of Transformer Subpackage



transformer
C ThresholdCleaner
thresholds : dict
transform(input_df : DataFrame)
C Sieve
filter_expression : str
transform(input_df : DataFrame)
C Transformer
derived
name : str
logger : logging.logger
transform(input_df : DataFrame)
C NewestByGroup
group_by : list = ["id"]
order_by : list = ["updated_at", "deleted_at"]
transform(input_df : DataFrame)
C Exploder
exploded_elem_name : str = "included"
path_to_array : str = "elem"
transform(input_df : DataFrame)
M mapper_custom_data_types
add_custom_data_type(function_name, func)
_get_select_expression_for_custom_type(*mapping_tuple)
provides custom data types
C Mapper
mapping : list
transform(input_df : DataFrame)

1.4.7 Create your own Transformer

Please see the *Create your own Transformer* for further details.

1.5 Loaders

Loaders take a `pyspark.sql.DataFrame` as an input and save it to a sink.

Each Loader class has to have a *load* method which takes a DataFrame as single paremter.

Possible Loader sinks can be **Hive Tables**, **Kudu Tables**, **HBase Tables**, **JDBC Sinks** or **ParquetFiles**.

1.5.1 Hive Database

class HiveLoader(*db_name, table_name, partition_definitions=[{'column_name': 'dt', 'column_type': 'IntegerType', 'default_value': None}], clear_partition=True, repartition_size=40, auto_create_table=True, overwrite_partition_value=True*)

Bases: *spooq2.loader.loader.Loader*

Persists a PySpark DataFrame into a Hive Table.

Examples

```
>>> HiveLoader(
>>>     db_name="users_and_friends",
>>>     table_name="friends_partitioned",
>>>     partition_definitions=[{
>>>         "column_name": "dt",
>>>         "column_type": "IntegerType",
>>>         "default_value": 20200201}],
>>>     clear_partition=True,
>>>     repartition_size=10,
>>>     overwrite_partition_value=False,
>>>     auto_create_table=False,
>>> ).load(input_df)
```

```
>>> HiveLoader(
>>>     db_name="users_and_friends",
>>>     table_name="all_friends",
>>>     partition_definitions=[],
>>>     repartition_size=200,
>>>     auto_create_table=True,
>>> ).load(input_df)
```

Parameters

- **db_name** (`str`) – The database name to load the data into.
- **table_name** (`str`) – The table name to load the data into. The database name must not be included in this parameter as it is already defined in the *db_name* parameter.
- **partition_definitions** (`list` of `dict`) – (Defaults to *[{"column_name": "dt", "column_type": "IntegerType", "default_value": None}]*).
  - **column_name** (`str`) - The Column's Name to partition by.
  - **column_type** (`str`) - The PySpark SQL DataType for the Partition Value as a String. This should normally either be 'IntegerType()' or 'StringType()'
  - **default_value** (`str` or `int`) - If *column_name* does not contain a value or *overwrite_partition_value* is set, this value will be used for the partitioning
- **clear_partition** (`bool`, (Defaults to True)) – This flag tells the Loader to delete the defined partitions before inserting the input DataFrame into the target table. Has no effect if no partitions are defined.

- **repartition_size** (`int`, (Defaults to 40)) – The DataFrame will be repartitioned on Spark level before inserting into the table. This effects the number of output files on which the Hive table is based.
- **auto_create_table** (`bool`, (Defaults to True)) – Whether the target table will be created if it does not yet exist.
- **overwrite_partition_value** (`bool`, (Defaults to True)) – Defines whether the values of columns defined in *partition_definitions* should explicitly set by default_values.

Raises

- `exceptions.AssertionError:` – partition_definitions has to be a list containing dicts. Expected dict content: 'column_name', 'column_type', 'default_value' per partition_definitions item.
- `exceptions.AssertionError:` – Items of partition_definitions have to be dictionaries.
- `exceptions.AssertionError:` – No column name set!
- `exceptions.AssertionError:` – Not a valid (PySpark) datatype for the partition column {name} | {type}.
- `exceptions.AssertionError:` – *clear_partition* is only supported if *overwrite_partition_value* is also enabled. This would otherwise result in clearing partitions on basis of dynamically values (from DataFrame) instead of explicitly defining the partition(s) to clear.

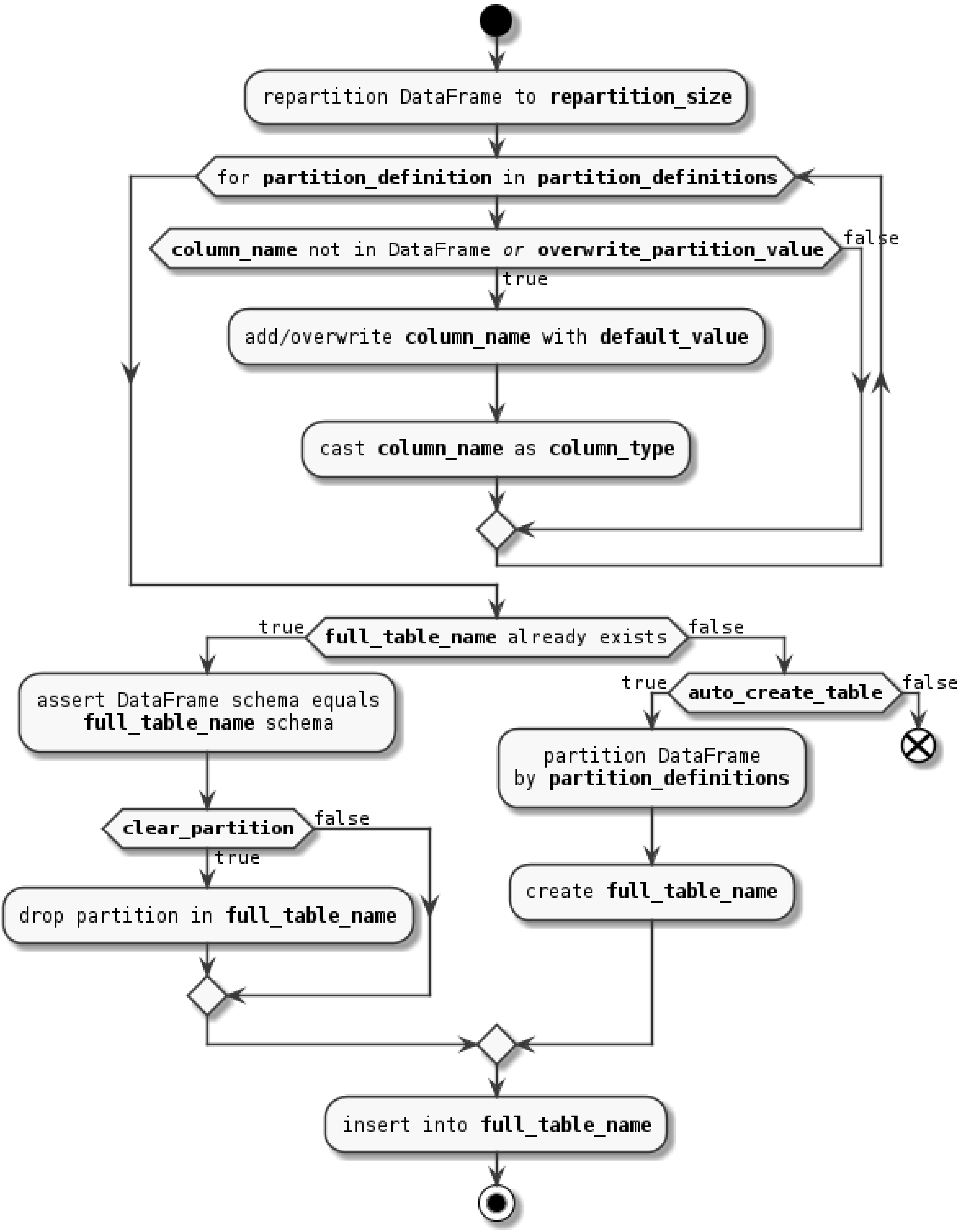
load(*input_df*)

Persists data from a PySpark DataFrame to a target table.

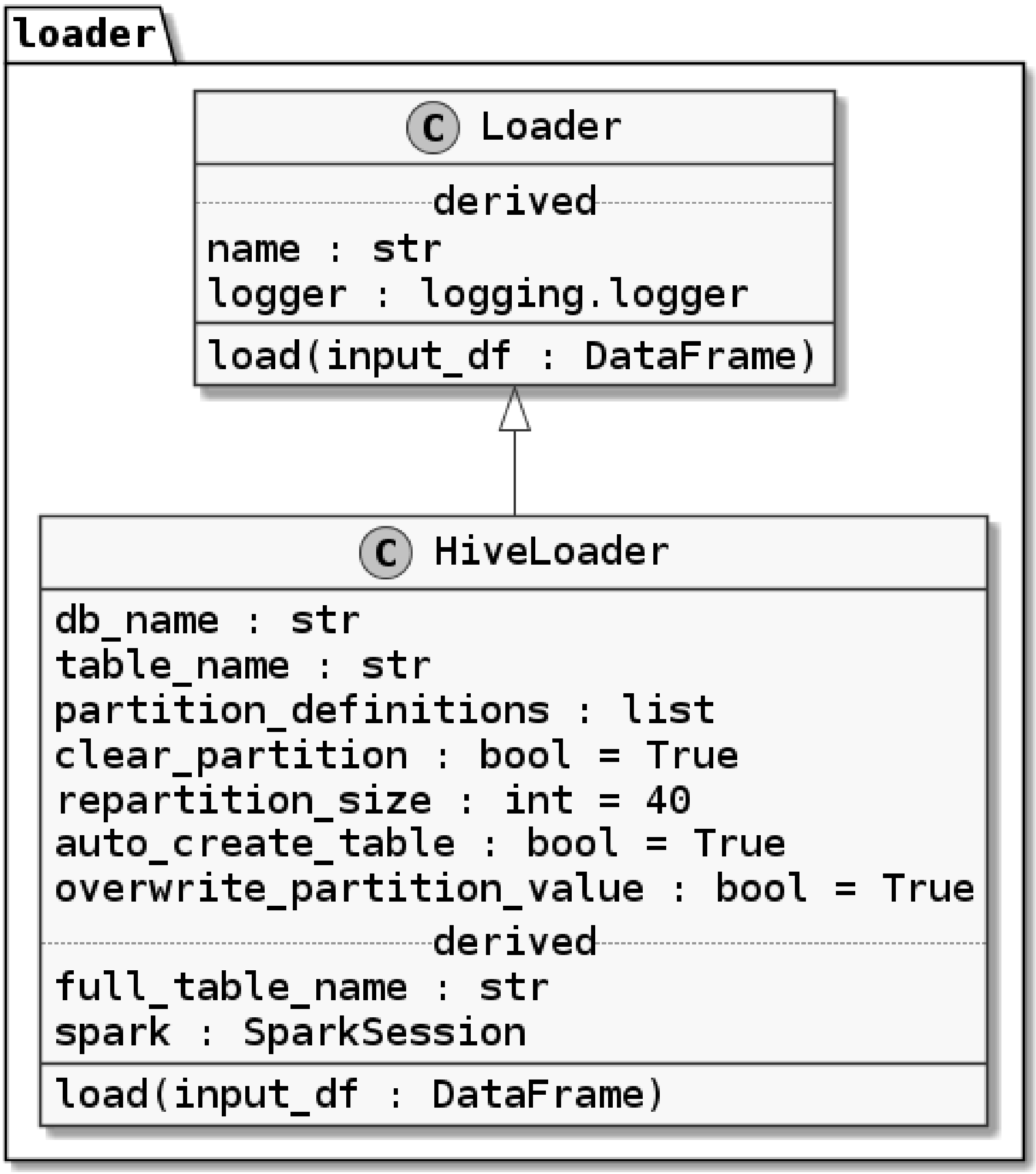
Parameters **input_df** (`pyspark.sql.DataFrame`) – Input DataFrame which has to be loaded to a target destination.

Note: This method takes only a single DataFrame as an input parameter. All other needed parameters are defined in the initialization of the Loader object.

Activity Diagram


repartition DataFrame to repartition_size
for partition_definition in partition_definitions
column_name not in DataFrame or overwrite_partition_value
false
true
add/overwrite column_name with default_value
cast column_name as column_type
true
full_table_name already exists
false
assert DataFrame schema equals full_table_name schema
true
auto_create_table
false
partition DataFrame by partition_definitions
clear_partition
false
true
drop partition in full_table_name
create full_table_name
insert into full_table_name

1.5.2 Class Diagram of Loader Subpackage


loader
Loader
derived
name : str
logger : logging.logger
load(input_df : DataFrame)
HiveLoader
db_name : str
table_name : str
partition_definitions : list
clear_partition : bool = True
repartition_size : int = 40
auto_create_table : bool = True
overwrite_partition_value : bool = True
derived
full_table_name : str
spark : SparkSession
load(input_df : DataFrame)

1.5.3 Create your own Loader

Please see the *Create your own Loader* for further details.

1.6 Pipeline

1.6.1 Pipeline

This type of object glues the aforementioned processes together and extracts, transforms (Transformer chain possible) and loads the data from start to end.

class Pipeline(*input_df=None*, *bypass_loader=False*)

Bases: `object`

Represents a Pipeline of an Extractor, (multiple) Transformers and a Loader Object.

extractor

The entry point of the Pipeline. Extracts a DataFrame from a Source.

Type Subclass of `spooq2.extractor.Extractor`

transformers

The Data Wrangling Part of the Pipeline. A chain of Transformers, a single Transformer or a PassThrough Transformer can be set and used.

Type List of Subclasses of `spooq2.transformer.Transformer` Objects

loader

The exit point of the Pipeline. Loads a DataFrame to a target Sink.

Type Subclass of `spooq2.loader.Loader`

name

Sets the __*name*__ of the class' type as *name*, which is essentially the Class' Name.

Type `str`

logger

Shared, class level logger for all instances.

Type `logging.Logger`

Example

```
>>> from spooq2.pipeline import Pipeline
>>> import spooq2.extractor as   E
>>> import spooq2.transformer as T
>>> import spooq2.loader as      L
>>>
>>> #  Definition how the output table should look like and where the␣
↪attributes come from:
>>> users_mapping = [
>>>     ("id",              "id",                     "IntegerType"),
>>>     ("guid",            "guid",                   "StringType"),
>>>     ("forename",        "attributes.first_name",  "StringType"),
>>>     ("surename",        "attributes.last_name",   "StringType"),
>>>     ("gender",          "attributes.gender",      "StringType"),
>>>     ("has_email",       "attributes.email",       "StringBoolean"),
>>>     ("has_university",  "attributes.university",  "StringBoolean"),
>>>     ("created_at",      "meta.created_at_ms",     "timestamp_ms_to_s"),
>>> ]
>>>
>>> #  The main object where all steps are defined:
>>> users_pipeline = Pipeline()
>>>
>>> #  Defining the EXTRACTION:
>>> users_pipeline.set_extractor(E.JSONExtractor(
>>>     input_path="tests/data/schema_v1/sequenceFiles"
>>> ))
>>>
>>> #  Defining the TRANSFORMATION:
>>> users_pipeline.add_transformers([
>>>     T.Mapper(mapping=users_mapping),
>>>     T.ThresholdCleaner(thresholds={"created_at": {
>>>                                                  "min": 0,
```

(continues on next page)

(continued from previous page)

```
>>>                                                 "max": 1580737513,
>>>                                                 "default": None}}),
>>>     T.NewestByGroup(group_by="id", order_by="created_at")
>>> ])
>>>
>>> #  Defining the LOAD:
>>> users_pipeline.set_loader(L.HiveLoader(
>>>     db_name="users_and_friends",
>>>     table_name="users",
>>>     partition_definitions=[{
>>>         "column_name": "dt",
>>>         "column_type": "IntegerType",
>>>         "default_value": 20200201}],
>>>     repartition_size=10,
>>> ))
>>>
>>> #  Executing the whole ETL pipeline
>>> users_pipeline.execute()
```

execute()

Executes the whole Pipeline at once.

Extracts from the Source, transformes the DataFrame and loads it into a target Sink.

Returns **input_df – If** the `bypass_loader` attribute was set to True in the Pipeline class, the output DataFrame from the Transformer(s) will be directly returned.

Return type `pyspark.sql.DataFrame`

Note: This method does not take ANY input parameters. All needed parameters are defined at the initialization phase.

extract()

Calls the extract Method on the Extractor Object.

Returns The output_df from the Extractor used as the input for the Transformer (chain).

Return type `pyspark.sql.DataFrame`

transform(*input_df*)

Calls the transform Method on the Transformer Object(s) in the order of importing the Objects while passing the DataFrame.

Parameters **input_df** (`pyspark.sql.DataFrame`) – The output DataFrame of the Extractor Object.

Returns The input DataFrame for the Loader.

Return type `pyspark.sql.DataFrame`

load(*input_df*)

Calls the load Method on the Loader Object.

Parameters **input_df** (`pyspark.sql.DataFrame`) – The output DataFrame from the Transformer(s).

Returns **input_df – If** the `bypass_loader` attribute was set to True in the Pipeline class, the output DataFrame from the Transformer(s) will be directly returned.

Return type `pyspark.sql.DataFrame`

set_extractor(*extractor*)

Sets an Extractor Object to be used within the Pipeline.

Parameters **extractor** (Subclass of `spooq2.extractor.Extractor`) – An already initialized Object of any Subclass of spooq2.extractor.Extractor.

Raises `exceptions.AssertionError:` – An input_df was already provided which bypasses the extraction action

add_transformers(*transformers*)
Adds a list of Transformer Objects to be used within the Pipeline.

Parameters **transformer** (`list` of Subclass of `spooq2.transformer.Transformer`) – Already initialized Object of any Subclass of spooq2.transformer.Transformer.

clear_transformers()
Clears the list of already added Transformers.

set_loader(*loader*)
Sets an Loader Object to be used within the Pipeline.

Parameters **loader** (Subclass of `spooq2.loader.Loader`) – An already initialized Object of any Subclass of spooq2.loader.Loader.

Raises `exceptions.AssertionError:` – You can not set a loader if the *bypass_loader* parameter is set.

1.6.2 Pipeline Factory

To decrease the complexity of building data pipelines for data engineers, an expert system or business rules engine can be used to automatically build and configure a data pipeline based on context variables, groomed metadata, and relevant rules.

class PipelineFactory(*url='http://localhost:5000/pipeline/get'*)
Bases: `object`

Provides an interface to automatically construct pipelines for Spooq.

Example

```
>>> pipeline_factory = PipelineFactory()
>>>
>>> #  Fetch user data set with applied mapping, filtering,
>>> #  and cleaning transformers
>>> df = pipeline_factory.execute({
>>>      "entity_type": "user",
>>>      "date": "2018-10-20",
>>>      "time_range": "last_day"})
>>>
>>> #  Load user data partition with applied mapping, filtering,
>>> #  and cleaning transformers to a hive database
>>> pipeline_factory.execute({
>>>      "entity_type": "user",
>>>      "date": "2018-10-20",
>>>      "batch_size": "daily"})
```

url
The end point of an expert system which will be called to infer names and parameters.

Type `str`, (Defaults to "http://localhost:5000/pipeline/get")

Note: PipelineFactory is only responsible for querying an expert system with provided parameters and constructing a Spooq pipeline out of the response. It does not have any reasoning capabilities itself! It requires therefore a HTTP service responding with a JSON object containing following structure:

```
{
    "extractor": {"name": "Type1Extractor", "params": {"key 1": "val 1", "key N
↪": "val N"}},
    "transformers": [
        {"name": "Type1Transformer", "params": {"key 1": "val 1", "key N":
↪"val N"}},
        {"name": "Type2Transformer", "params": {"key 1": "val 1", "key N":
↪"val N"}},
        {"name": "Type3Transformer", "params": {"key 1": "val 1", "key N":
↪"val N"}},
        {"name": "Type4Transformer", "params": {"key 1": "val 1", "key N":
↪"val N"}},
        {"name": "Type5Transformer", "params": {"key 1": "val 1", "key N":
↪"val N"}},
    ],
    "loader": {"name": "Type1Loader", "params": {"key 1": "val 1", "key N":
↪"val N"}}
}
```

Hint: There is an experimental implementation of an expert system which complies with the requirements of PipelineFactory called *spooq_rules*. If you are interested, please ask the author of Spooq about it.

`execute`(*context_variables*)

Fetches a ready-to-go pipeline instance via *`get_pipeline()`* and executes it.

Parameters **`context_variables`** (`dict`) – These collection of parameters should describe the current context about the use case of the pipeline. Please see the examples of the PipelineFactory class' documentation.

Returns

- `pyspark.sql.DataFrame` – If the loader component is by-passed (in the case of ad_hoc use cases).
- `None` – If the loader component does not return a value (in the case of persisting data).

`get_metadata`(*context_variables*)

Sends a POST request to the defined endpoint (*url*) containing the supplied context variables.

Parameters **`context_variables`** (`dict`) – These collection of parameters should describe the current context about the use case of the pipeline. Please see the examples of the PipelineFactory class' documentation.

Returns Names and parameters of each ETL component to construct a Spooq pipeline

Return type `dict`

`get_pipeline`(*context_variables*)

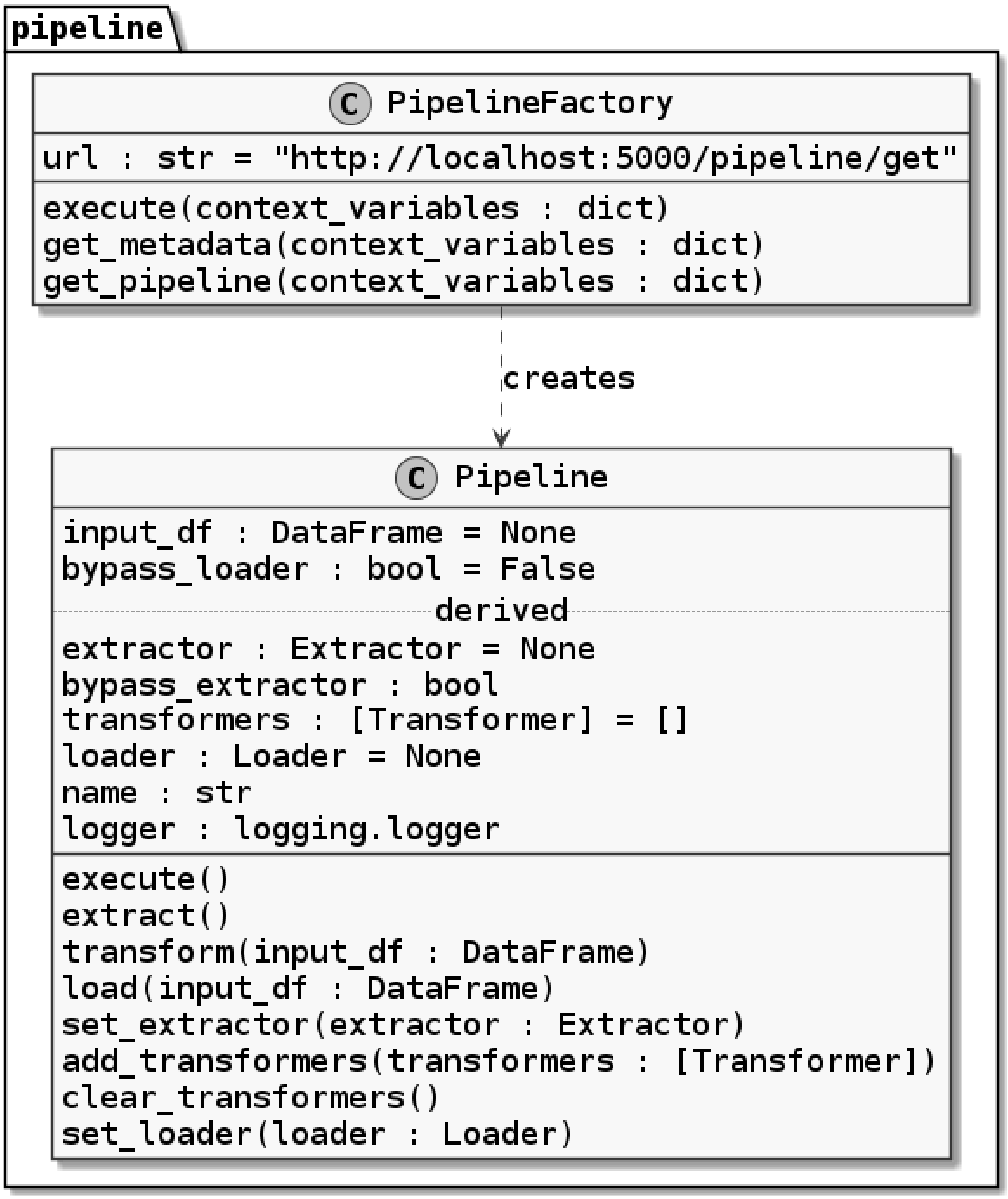
Fetches the necessary metadata via *`get_metadata()`* and returns a ready-to-go pipeline instance.

Parameters **`context_variables`** (`dict`) – These collection of parameters should describe the current context about the use case of the pipeline. Please see the examples of the PipelineFactory class' documentation.

Returns A Spooq pipeline instance which is fully configured and can still be adapted and consequently executed.

Return type `Pipeline`

1.6.3 Class Diagram of Pipeline Subpackage


pipeline
C PipelineFactory
url : str = "http://localhost:5000/pipeline/get"
execute(context_variables : dict)
get_metadata(context_variables : dict)
get_pipeline(context_variables : dict)
creates
C Pipeline
input_df : DataFrame = None
bypass_loader : bool = False
derived
extractor : Extractor = None
bypass_extractor : bool
transformers : [Transformer] = []
loader : Loader = None
name : str
logger : logging.logger
execute()
extract()
transform(input_df : DataFrame)
load(input_df : DataFrame)
set_extractor(extractor : Extractor)
add_transformers(transformers : [Transformer])
clear_transformers()
set_loader(loader : Loader)

1.7 Spooq Base

1.7.1 Global Logger

Global Logger instance used by Spooq2.

Example

```
>>> import logging
>>> logga = logging.getLogger("spooq2")
<logging.Logger at 0x7f5dc8eb2890>
>>> logga.info("Hello World")
[spooq2] 2020-03-21 23:55:48,253 INFO logging_example::<module>::4: Hello World
```

initialize()

Initializes the global logger for Spooq with pre-defined levels for `stdout` and `stderr`. No input parameters are needed, as the configuration is received via *get_logging_level()*.

Note:

The output format is defined as:

> "[%(name)s] %(asctime)s %(levelname)s %(module)s::%(funcName)s::%(lineno)d: %(message)s"
>
> For example "[spooq2] 2020-03-11 15:40:59,313 DEBUG newest_by_group::__init__::53: group by columns: [u'user_id']"

Warning: The `root` logger of python is also affected as it has to have a level at least as fine grained as the logger of Spooq, to be able to produce an output.

get_logging_level()

Returns the logging level depending on the environment variable *SPOOQ_ENV*.

Note:

If SPOOQ_ENV is

- **dev** -> "DEBUG"
- **test** -> "ERROR"
- something else -> "INFO"

Returns Logging level

Return type `str`

1.7.2 Extractor Base Class

Extractors are used to fetch, extract and convert a source data set into a PySpark DataFrame. Exemplary extraction sources are **JSON Files** on file systems like HDFS, DBFS or EXT4 and relational database systems via **JDBC**.

class Extractor

Bases: `object`

Base Class of Extractor Classes.

name

Sets the *__name__* of the class' type as *name*, which is essentially the Class' Name.

Type `str`

logger

Shared, class level logger for all instances.

Type `logging.Logger`

extract()

Extracts Data from a Source and converts it into a PySpark DataFrame.

Returns

Return type pyspark.sql.DataFrame

Note: This method does not take ANY input parameters. All needed parameters are defined in the initialization of the Extractor Object.

Create your own Extractor

Let your extractor class inherit from the extractor base class. This includes the name, string representation and logger attributes from the superclass.

The only mandatory thing is to provide an *extract()* method which
takes
=> *no input parameters*
and **returns** a
=> *PySpark DataFrame!*

All configuration and parameterization should be done while initializing the class instance.

Here would be a simple example for a CSV Extractor:

Exemplary Sample Code

Listing 1: src/spooq2/extractor/csv_extractor.py:

```
from pyspark.sql import SparkSession

from extractor import Extractor

class CSVExtractor(Extractor):
    """
    This is a simplified example on how to implement a new extractor class.
    Please take your time to write proper docstrings as they are automatically
    parsed via Sphinx to build the HTML and PDF documentation.
    Docstrings use the style of Numpy (via the napoleon plug-in).

    This class uses the :meth:`pyspark.sql.DataFrameReader.csv` method internally.

    Examples
    --------
    extracted_df = CSVExtractor(
        input_file='data/input_data.csv'
    ).extract()

    Parameters
    ----------
    input_file: :any:`str`
        The explicit file path for the input data set. Globbing support depends
        on implementation of Spark's csv reader!

    Raises
```

(continues on next page)

(continued from previous page)

```
    ------
    :any:`exceptions.TypeError`:
        path can be only string, list or RDD
    """

    def __init__(self, input_file):
        super(CSVExtractor, self).__init__()
        self.input_file = input_file
        self.spark = SparkSession.Builder()\
            .enableHiveSupport()\
            .appName('spooq2.extractor: {nm}'.format(nm=self.name))\
            .getOrCreate()

    def extract(self):
        self.logger.info('Loading Raw CSV Files from: ' + self.input_file)
        output_df = self.spark.read.load(
            input_file,
            format="csv",
            sep=";",
            inferSchema="true",
            header="true"
        )

        return output_df
```

References to include

Listing 2: src/spooq2/extractor/__init__.py:

```
--- original
+++ adapted
@@ -1,8 +1,10 @@
 from jdbc import JDBCExtractorIncremental, JDBCExtractorFullLoad
 from json_files import JSONExtractor
+from csv_extractor import CSVExtractor

 __all__ = [
     "JDBCExtractorIncremental",
     "JDBCExtractorFullLoad",
     "JSONExtractor",
+    "CSVExtractor",
 ]
```

Tests

One of Spooq2's features is to provide tested code for multiple data pipelines. Please take your time to write sufficient unit tests! You can reuse test data from *tests/data* or create a new schema / data set if needed. A SparkSession is provided as a global fixture called *spark_session*.

Listing 3: tests/unit/extractor/test_csv.py:

```
import pytest

from spooq2.extractor import CSVExtractor

@pytest.fixture()
def default_extractor():
```

(continues on next page)

(continued from previous page)

```
    return CSVExtractor(input_path="data/input_data.csv")


class TestBasicAttributes(object):

    def test_logger_should_be_accessible(self, default_extractor):
        assert hasattr(default_extractor, "logger")

    def test_name_is_set(self, default_extractor):
        assert default_extractor.name == "CSVExtractor"

    def test_str_representation_is_correct(self, default_extractor):
        assert unicode(default_extractor) == "Extractor Object of Class␣
↪CSVExtractor"

class TestCSVExtraction(object):

    def test_count(default_extractor):
        """Converted DataFrame has the same count as the input data"""
        expected_count = 312
        actual_count = default_extractor.extract().count()
        assert expected_count == actual_count

    def test_schema(default_extractor):
        """Converted DataFrame has the expected schema"""
        do_some_stuff()
        assert expected == actual
```

Documentation

You need to create a *rst* for your extractor which needs to contain at minimum the *automodule* or the *autoclass* directive.

Listing 4: docs/source/extractor/csv.rst:

```
CSV Extractor
=============

Some text if you like...

.. automodule:: spooq2.extractor.csv_extractor
```

To automatically include your new extractor in the HTML documentation you need to add it to a *toctree* directive. Just refer to your newly created *csv.rst* file within the extractor overview page.

Listing 5: docs/source/extractor/overview.rst:

```
--- original
+++ adapted
@@ -7,8 +7,9 @@
 .. toctree::

     json
     jdbc
+    csv

 Class Diagram of Extractor Subpackage
 ------------------------------------------------
 .. uml:: ../diagrams/from_thesis/class_diagram/extractors.puml
```

That should be all!

1.7.3 Transformer Base Class

Transformers take a `pyspark.sql.DataFrame` as an input, transform it accordingly and return a PySpark DataFrame.

Each Transformer class has to have a *transform* method which takes no arguments and returns a PySpark DataFrame.

Possible transformation methods can be **Selecting the most up to date record by id**, **Exploding an array**, **Filter (on an exploded array)**, **Apply basic threshold cleansing** or **Map the incoming DataFrame to at provided structure**.

class Transformer

Bases: `object`

Base Class of Transformer Classes.

name

Sets the *__name__* of the class' type as *name*, which is essentially the Class' Name.

Type `str`

logger

Shared, class level logger for all instances.

Type `logging.Logger`

transform(*input_df*)

Performs a transformation on a DataFrame.

Parameters **input_df** (`pyspark.sql.DataFrame`) – Input DataFrame

Returns Transformed DataFrame.

Return type `pyspark.sql.DataFrame`

Note: This method does only take the Input DataFrame as a parameters. All other needed parameters are defined in the initialization of the Transformator Object.

Create your own Transformer

Let your transformer class inherit from the transformer base class. This includes the name, string representation and logger attributes from the superclass.

The only mandatory thing is to provide a *transform()* method which
takes a
=> *PySpark DataFrame!*
and **returns** a
=> *PySpark DataFrame!*

All configuration and parameterization should be done while initializing the class instance.

Here would be a simple example for a transformer which drops records without an Id:

Exemplary Sample Code

Listing 6: src/spooq2/transformer/no_id_dropper.py:

```python
from transformer import Transformer

class NoIdDropper(Transformer):
    """
    This is a simplified example on how to implement a new transformer class.
    Please take your time to write proper docstrings as they are automatically
    parsed via Sphinx to build the HTML and PDF documentation.
    Docstrings use the style of Numpy (via the napoleon plug-in).

    This class uses the :meth:`pyspark.sql.DataFrame.dropna` method internally.

    Examples
    --------
    input_df = some_extractor_instance.extract()
    transformed_df = NoIdDropper(
        id_columns='user_id'
    ).transform(input_df)

    Parameters
    ----------
    id_columns: :any:`str` or :any:`list`
        The name of the column containing the identifying Id values.
        Defaults to "id"

    Raises
    ------
    :any:`exceptions.ValueError`:
        "how ('" + how + "') should be 'any' or 'all'"
    :any:`exceptions.ValueError`:
        "subset should be a list or tuple of column names"
    """

    def __init__(self, id_columns='id'):
        super(NoIdDropper, self).__init__()
        self.id_columns = id_columns


    def transform(self, input_df):
        self.logger.info("Dropping records without an Id (columns to consider: {col})"
            .format(col=self.id_columns))
        output_df = input_df.dropna(
            how='all',
            thresh=None,
            subset=self.id_columns
        )

        return output_df
```

References to include

This makes it possible to import the new transformer class directly from *spooq2.transformer* instead of *spooq2.transformer.no_id_dropper*. It will also be imported if you use *from spooq2.transformer import **.

Listing 7: src/spooq2/transformer/__init__.py:

```
--- original
+++ adapted
@@ -1,13 +1,15 @@
 from newest_by_group import NewestByGroup
 from mapper import Mapper
 from exploder import Exploder
 from threshold_cleaner import ThresholdCleaner
 from sieve import Sieve
+from no_id_dropper import NoIdDropper

 __all__ = [
     "NewestByGroup",
     "Mapper",
     "Exploder",
     "ThresholdCleaner",
     "Sieve",
+    "NoIdDropper",
 ]
```

Tests

One of Spooq2's features is to provide tested code for multiple data pipelines. Please take your time to write sufficient unit tests! You can reuse test data from *tests/data* or create a new schema / data set if needed. A SparkSession is provided as a global fixture called *spark_session*.

Listing 8: tests/unit/transformer/test_no_id_dropper.py:

```
import pytest
from pyspark.sql.dataframe import DataFrame

from spooq2.transformer import NoIdDropper


@pytest.fixture()
def default_transformer():
    return NoIdDropper(id_columns=["first_name", "last_name"])


@pytest.fixture()
def input_df(spark_session):
    return spark_session.read.parquet("../data/schema_v1/parquetFiles")


@pytest.fixture()
def transformed_df(default_transformer, input_df):
    return default_transformer.transform(input_df)


class TestBasicAttributes(object):

    def test_logger_should_be_accessible(self, default_transformer):
        assert hasattr(default_transformer, "logger")

    def test_name_is_set(self, default_transformer):
        assert default_transformer.name == "NoIdDropper"

    def test_str_representation_is_correct(self, default_transformer):
        assert unicode(default_transformer) == "Transformer Object of Class␣
↪NoIdDropper"
```

(continues on next page)

(continued from previous page)

```
class TestNoIdDropper(object):

    def test_records_are_dropped(transformed_df, input_df):
        """Transformed DataFrame has no records with missing first_name and last_
↪name"""
        assert input_df.where("first_name is null or last_name is null").count() >␣
↪0
        assert transformed_df.where("first_name is null or last_name is null").
↪count() == 0

    def test_schema_is_unchanged(transformed_df, input_df):
        """Converted DataFrame has the expected schema"""
        assert transformed_df.schema == input_df.schema
```

Documentation

You need to create a *rst* for your transformer which needs to contain at minimum the *automodule* or the *autoclass* directive.

Listing 9: docs/source/transformer/no_id_dropper.rst:

```
Record Dropper if Id is missing
===============================

Some text if you like...

.. automodule:: spooq2.transformer.no_id_dropper
```

To automatically include your new transformer in the HTML / PDF documentation you need to add it to a *toctree* directive. Just refer to your newly created *no_id_dropper.rst* file within the transformer overview page.

Listing 10: docs/source/transformer/overview.rst:

```
--- original
+++ adapted
@@ -7,14 +7,15 @@
 .. toctree::

     exploder
     sieve
     mapper
     threshold_cleaner
     newest_by_group
+    no_id_dropper

 Class Diagram of Transformer Subpackage
 ------------------------------------------------
 .. uml:: ../diagrams/from_thesis/class_diagram/transformers.puml
```

That should be it!

1.7.4 Loader Base Class

Loaders take a `pyspark.sql.DataFrame` as an input and save it to a sink.

Each Loader class has to have a *load* method which takes a DataFrame as single paremter.

Possible Loader sinks can be **Hive Tables**, **Kudu Tables**, **HBase Tables**, **JDBC Sinks** or **ParquetFiles**.

class Loader

Bases: `object`

Base Class of Loader Objects.

name

Sets the *__name__* of the class' type as *name*, which is essentially the Class' Name.

Type `str`

logger

Shared, class level logger for all instances.

Type `logging.Logger`

load(*input_df*)

Persists data from a PySpark DataFrame to a target table.

Parameters **input_df** (`pyspark.sql.DataFrame`) – Input DataFrame which has to be loaded to a target destination.

Note: This method takes only a single DataFrame as an input parameter. All other needed parameters are defined in the initialization of the Loader object.

Create your own Loader

Let your loader class inherit from the loader base class. This includes the name, string representation and logger attributes from the superclass.

The only mandatory thing is to provide a *load()* method which
takes a
=> *PySpark DataFrame!*
and **returns**
nothing (or at least the API does not expect anything)

All configuration and parameterization should be done while initializing the class instance.

Here would be a simple example for a loader which save a DataFrame to parquet files:

Exemplary Sample Code

Listing 11: src/spooq2/loader/parquet.py:

```
from pyspark.sql import functions as F

from loader import Loader

class ParquetLoader(loader):
    """
    This is a simplified example on how to implement a new loader class.
    Please take your time to write proper docstrings as they are automatically
    parsed via Sphinx to build the HTML and PDF documentation.
    Docstrings use the style of Numpy (via the napoleon plug-in).

    This class uses the :meth:`pyspark.sql.DataFrameWriter.parquet` method␣
↪internally.
```

(continues on next page)

(continued from previous page)

```
    Examples
    ---------
    input_df = some_extractor_instance.extract()
    output_df = some_transformer_instance.transform(input_df)
    ParquetLoader(
        path="data/parquet_files",
        partition_by="dt",
        explicit_partition_values=20200201,
        compression=""gzip""
    ).load(output_df)

    Parameters
    ----------
    path: :any:`str`
        The path to where the loader persists the output parquet files.
        If partitioning is set, this will be the base path where the partitions
        are stored.

    partition_by: :any:`str` or :any:`list` of (:any:`str`)
        The column name or names by which the output should be partitioned.
        If the partition_by parameter is set to None, no partitioning will be
        performed.
        Defaults to "dt"

    explicit_partition_values: :any:`str` or :any:`int`
                                or :any:`list` of (:any:`str` and :any:`int`)
        Only allowed if partition_by is not None.
        If explicit_partition_values is not None, the dataframe will
            * overwrite the partition_by columns values if it already exists or
            * create and fill the partition_by columns if they do not yet exist
        Defaults to None

    compression: :any:`str`
        The compression codec used for the parquet output files.
        Defaults to "snappy"


    Raises
    ------
    :any:`exceptions.AssertionError`:
        explicit_partition_values can only be used when partition_by is not None
    :any:`exceptions.AssertionError`:
        explicit_partition_values and partition_by must have the same length
    """

    def __init__(self, path, partition_by="dt", explicit_partition_values=None,␣
↪compression_codec="snappy"):
        super(ParquetLoader, self).__init__()
        self.path = path
        self.partition_by = partition_by
        self.explicit_partition_values = explicit_partition_values
        self.compression_codec = compression_codec
        if explicit_partition_values is not None:
            assert (partition_by is not None,
                "explicit_partition_values can only be used when partition_by is␣
↪not None")
            assert (len(partition_by) == len(explicit_partition_values),
                "explicit_partition_values and partition_by must have the same␣
↪length")

    def load(self, input_df):
```

(continues on next page)

(continued from previous page)

```
        self.logger.info("Persisting DataFrame as Parquet Files to " + self.path)

        if isinstance(self.explicit_partition_values, list):
            for (k, v) in zip(self.partition_by, self.explicit_partition_values):
                input_df = input_df.withColumn(k, F.lit(v))
        elif isinstance(self.explicit_partition_values, basestring):
            input_df = input_df.withColumn(self.partition_by, F.lit(self.explicit_
↪partition_values))

        input_df.write.parquet(
            path=self.path,
            partitionBy=self.partition_by,
            compression=self.compression_codec
        )
```

References to include

This makes it possible to import the new loader class directly from *spooq2.loader* instead of *spooq2.loader.parquet*. It will also be imported if you use *from spooq2.loader import **.

Listing 12: src/spooq2/loader/__init__.py:

```
--- original
+++ adapted
@@ -1,7 +1,9 @@
 from loader import Loader
 from hive_loader import HiveLoader
+from parquet import ParquetLoader

 __all__ = [
     "Loader",
     "HiveLoader",
+    "ParquetLoader",
 ]
```

Tests

One of Spooq2's features is to provide tested code for multiple data pipelines. Please take your time to write sufficient unit tests! You can reuse test data from *tests/data* or create a new schema / data set if needed. A SparkSession is provided as a global fixture called *spark_session*.

Listing 13: tests/unit/loader/test_parquet.py:

```
import pytest
from pyspark.sql.dataframe import DataFrame

from spooq2.loader import ParquetLoader


@pytest.fixture(scope="module")
def output_path(tmpdir_factory):
    return str(tmpdir_factory.mktemp("parquet_output"))


@pytest.fixture(scope="module")
def default_loader(output_path):
    return ParquetLoader(
```

(continues on next page)

(continued from previous page)

```
        path=output_path,
        partition_by="attributes.gender",
        explicit_partition_values=None,
        compression_codec=None
    )


@pytest.fixture(scope="module")
def input_df(spark_session):
    return spark_session.read.parquet("../data/schema_v1/parquetFiles")


@pytest.fixture(scope="module")
def loaded_df(default_loader, input_df, spark_session, output_path):
    default_loader.load(input_df)
    return spark_session.read.parquet(output_path)


class TestBasicAttributes(object):

    def test_logger_should_be_accessible(self, default_loader):
        assert hasattr(default_loader, "logger")

    def test_name_is_set(self, default_loader):
        assert default_loader.name == "ParquetLoader"

    def test_str_representation_is_correct(self, default_loader):
        assert unicode(default_loader) == "loader Object of Class ParquetLoader"

class TestParquetLoader(object):

    def test_count_did_not_change(loaded_df, input_df):
        """Persisted DataFrame has the same number of records than the input␣
↪DataFrame"""
        assert input_df.count() == output_df.count() and input_df.count() > 0

    def test_schema_is_unchanged(loaded_df, input_df):
        """Loaded DataFrame has the same schema as the input DataFrame"""
        assert loaded.schema == input_df.schema
```

Documentation

You need to create a *rst* for your loader which needs to contain at minimum the *automodule* or the *autoclass* directive.

Listing 14: docs/source/loader/parquet.rst:

```
Parquet Loader
===============================

Some text if you like...

.. automodule:: spooq2.loader.parquet
```

To automatically include your new loader in the HTML / PDF documentation you need to add it to a *toctree* directive. Just refer to your newly created *parquet.rst* file within the loader overview page.

Listing 15: docs/source/loader/overview.rst:

```
--- original
+++ adapted
@@ -7,4 +7,5 @@
 .. toctree::
     hive_loader
+    parquet

 Class Diagram of Loader Subpackage
```

That should be it!

1.8 Setup for Development, Testing, Documenting

Attention: The current version of Spooq is designed (and tested) for Python 2.7/3.7/3.8 on ubuntu, manjaro linux and WSL2 (Windows Subsystem Linux).

1.8.1 Prerequisites

- python 2.7 or python 3.7/3.8
- Java 8 (jdk8-openjdk)
- pipenv
- Latex (for PDF documentation)

1.8.2 Setting up the Environment

The requirements are stored in the file *Pipfile* separated for production and development packages.

To install the packages needed for development and testing run the following command:

```
$ pipenv install --dev
```

This will create a virtual environment in *~/.local/share/virtualenvs*.

If you want to have your virtual environment installed as a sub-folder (.venv) you have to set the environment variable *PIPENV_VENV_IN_PROJECT* to 1.

To remove a virtual environment created with pipenv just change in the folder where you created it and execute *pipenv –rm*.

1.8.3 Activate the Virtual Environment

Listing 16: To activate the virtual environment enter:

```
$ pipenv shell
```

Listing 17: To deactivate the virtual environment simply enter:

```
$ exit
# or close the shell
```

For more commands of pipenv call *pipenv -h*.

1.8.4 Creating Your Own Components

Implementing new extractors, transformers, or loaders is fairly straightforward. Please refer to following descriptions and examples to get an idea:

- *Create your own Extractor*
- *Create your own Transformer*
- *Create your own Loader*

1.8.5 Running Tests

The tests are implemented with the pytest framework.

Listing 18: Start all tests:

```
$ pipenv shell
$ cd tests
$ pytest
```

Test Plugins

Those are the most useful plugins automatically used:

html

Listing 19: Generate an HTML report for the test results:

```
$ pytest --html=report.html
```

random-order

Shuffles the order of execution for the tests to avoid / discover dependencies of the tests.

Randomization is set by a seed number. To re-test the same order of execution where you found an error, just set the seed value to the same as for the failing test. To temporarily disable this feature run with *pytest -p no:random-order -v*

cov

Generates an HTML for the test coverage

Listing 20: Get a test coverage report in the terminal:

```
$ pytest --cov-report term --cov=spooq2
```

Listing 21: Get the test coverage report as HTML

```
$ pytest --cov-report html:cov_html --cov=spooq2
```

ipdb

To use ipdb (IPython Debugger) add following code at your breakpoint::

```
>>> import ipdb
>>> ipdb.set_trace()
```

You have to start pytest with *-s* if you want to use interactive debugger.

```
$ pytest -s
```

1.8.6 Generate Documentation

This project uses Sphinx for creating its documentation. Graphs and diagrams are produced with PlantUML.

The main documentation content is defined as docstrings within the source code. To view the current documentation open *docs/build/html/index.html* or *docs/build/latex/spooq2.pdf* in your application of choice. There are symlinks in the root folder for symplicity:

- Documentation.html
- Documentation.pdf

Although, if you are reading this, you have probably already found the documentation…

Diagrams

For generating the graphs and diagrams, you need a working plantuml installation on your computer! Please refer to sphinxcontrib-plantuml.

HTML

```
$ cd docs
$ make html
$ chromium build/html/index.html
```

PDF

For generating documentation in the PDF format you need to have a working (pdf)latex installation on your computer! Please refer to TexLive on how to install TeX Live - a compatible latex distribution. But beware, the download size is huge!

```
$ cd docs
$ make latexpdf
$ evince build/latex/Spooq2.pdf
```

Configuration

Themes, plugins, settings, … are defined in *docs/source/conf.py*.

napoleon

Enables support for parsing docstrings in NumPy / Google Style

intersphinx

Allows linking to other projects' documentation. E.g., PySpark, Python2 To add an external project, at the documentation link to *intersphinx_mapping* in *conf.py*

recommonmark

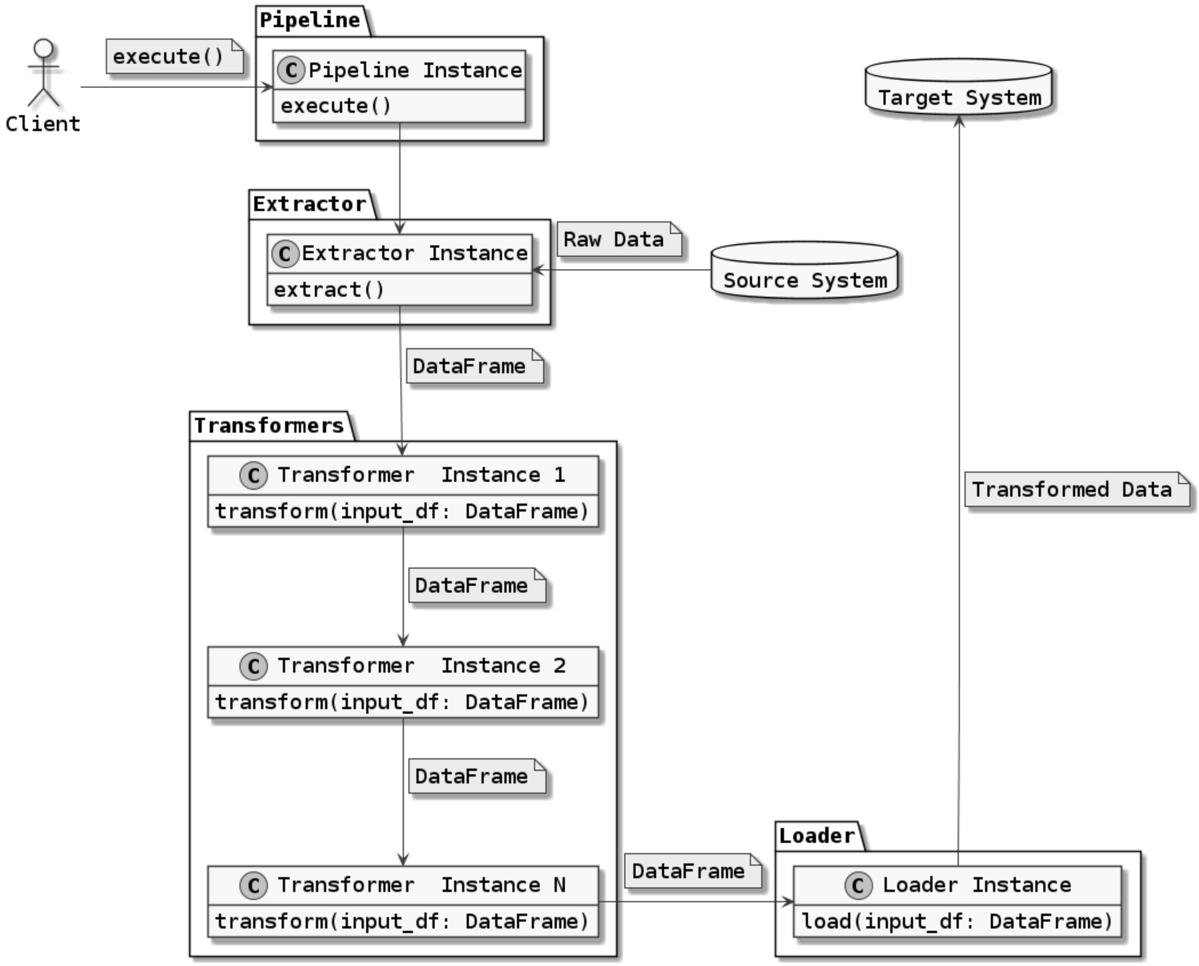
This allows you to write CommonMark (Markdown) inside of Docutils & Sphinx projects instead of rst.

plantuml

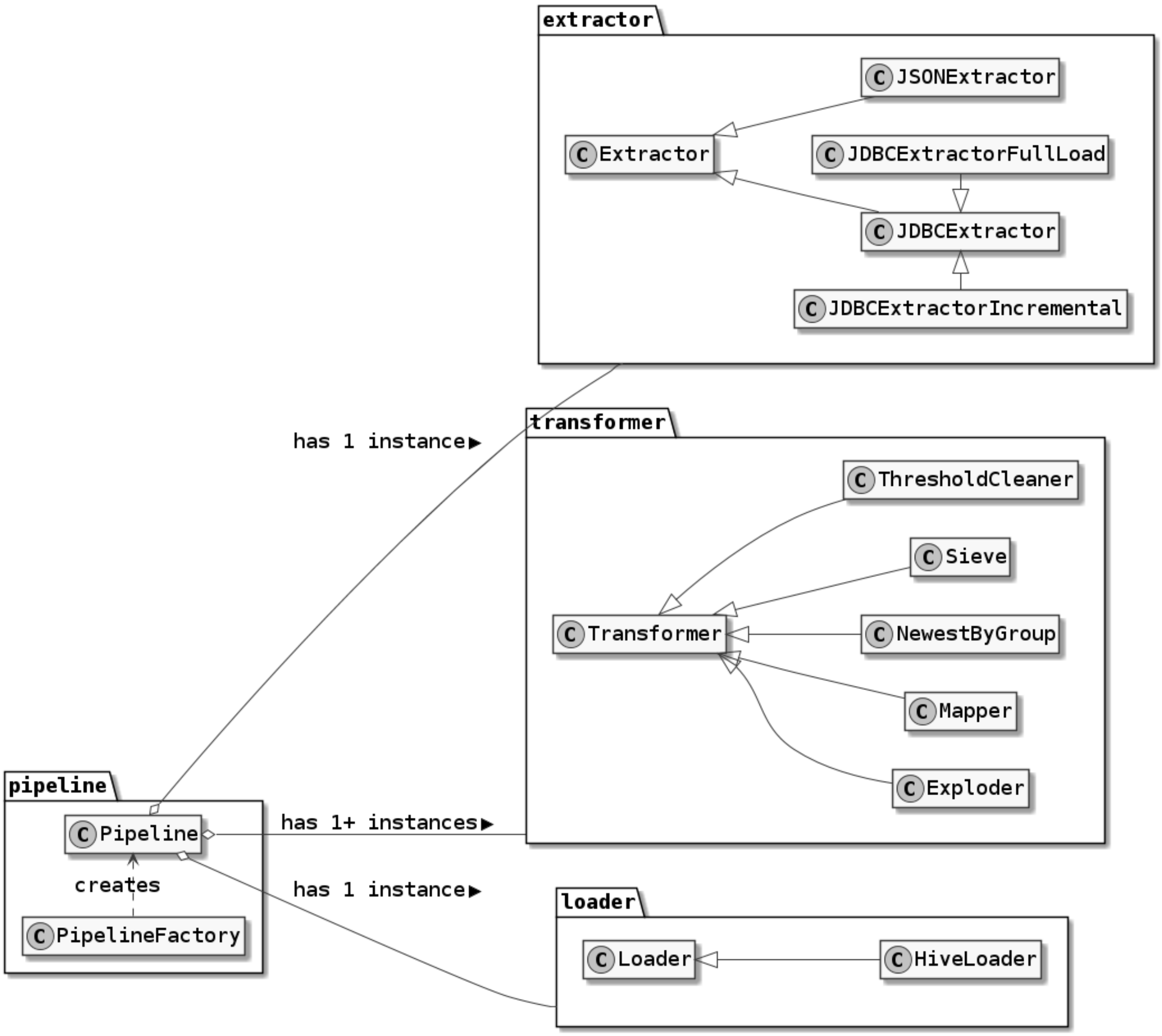
Allows for inline Plant UML code (uml directive) which is automatically rendered into an svg image and placed in the document. Allows also to source puml-files. See *Architecture Overview* for an example.

1.9 Architecture Overview

1.9.1 Typical Data Flow of a Spooq Data Pipeline



1.9.2 Simplified Class Diagram



CHAPTER 2

Indices and tables

- modindex
- search

Python Module Index

Index