Spooq Documentation

Release 3.4.2

David Eigenstuhler

Aug 08, 2024

CONTENTS

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

The main components are: * Extractors * Transformers * Loaders

Those components are independent and can be used separately or be plugged-in into a pipeline instance. You can also use the custom functions from the Mapper transformer directly with PySpark (f.e. `select` or `withColumn`).

CHAPTER

ONE

EXAMPLE OF MAPPER TRANSFORMER

```
from pyspark.sql import Row
from pyspark.sql import functions as F, types as T
from spooq.transformer import Mapper
from spooq.transformer import mapper_transformations as spq

input_df = spark.createDataFrame(
   [
      Row(
         struct_a=Row(idx="000_123_456", sts="enabled", ts="1597069446000"),
         struct_b=Row(itms="1,2,4", sts="whitelisted", ts="2020-08-12T12:43:14+0000"),
         struct_c=Row(email="abc@def.com", gndr="F", dt="2020-08-05", cmt="fine"),
      ),
      Row(
         struct_a=Row(idx="000_654_321", sts="off", ts="1597069500784"),
         struct_b=Row(itms="5", sts="blacklisted", ts="2020-07-01T12:43:14+0000"),
         struct_c=Row(email="", gndr="m", dt="2020-06-27", cmt="faulty"),
      ),
   ],
   schema="""
      a: struct<idx string, sts string, ts string>,
      b: struct<itms string, sts string, ts string>,
      c: struct<email string, gndr string, dt string, cmt string>
   """
)
input_df.printSchema()
root
 |-- a: struct (nullable = true)
 |    |-- idx: string (nullable = true)
 |    |-- sts: string (nullable = true)
 |    |-- ts: string (nullable = true)
 |-- b: struct (nullable = true)
 |    |-- itms: string (nullable = true)
 |    |-- sts: string (nullable = true)
 |    |-- ts: string (nullable = true)
 |-- c: struct (nullable = true)
 |    |-- email: string (nullable = true)
 |    |-- gndr: string (nullable = true)
 |    |-- dt: string (nullable = true)
 |    |-- cmt: string (nullable = true)


mapping = [
    # output_name     # source                # transformation
   ("index",         "a.idx",                spq.to_int),  # removes leading zeros␣
```

(continues on next page)

(continued from previous page)

```
↪and underline characters
    ("is_enabled",      "a.sts",                spq.to_bool),  # recognizes additional␣
↪words like "on", "off", "disabled", "enabled", ...
    ("a_updated_at",    "a.ts",                 spq.to_timestamp),  # supports unix␣
↪timestamps in ms or seconds and strings
    ("items",           "b.itms",               spq.str_to_array(cast="int")),  #␣
↪splits a comma delimited string into an array and casts its elements
    ("block_status",    "b.sts",                spq.map_values(mapping={"whitelisted":
↪"allowed", "blacklisted": "blocked"})),  # applies lookup dictionary
    ("b_updated_at",    "b.ts",                 spq.to_timestamp),  # supports unix␣
↪timestamps in ms or seconds and strings
    ("has_email",       "c.email",              spq.has_value),  # interprets also␣
↪empty strings as no value, although, zeros are values
    ("gender",          "c.gndr",               spq.apply(func=F.lower)),  # applies␣
↪provided function to all values
    ("creation_date",   "c.dt",                 spq.to_timestamp(cast="date")),  #␣
↪explicitly casts result after transformation
    ("processed_at",    F.current_timestamp(),  spq.as_is),  # source column is a␣
↪function, no transformation to the results
    ("comment",         "c.cmt",                "string"),  # no transformation, only␣
↪cast; alternatively: spq.to_str or spq.as_is(cast="string")
]
output_df = Mapper(mapping).transform(input_df)

output_df.show(truncate=False)
+------+----------+----------------------+---------+------------+-------------------
↪+---------+------+-------------+----------------------+-------+
|index |is_enabled|a_updated_at          |items    |block_status|b_updated_at       ␣
↪|has_email|gender|creation_date|processed_at          |comment|
+------+----------+----------------------+---------+------------+-------------------
↪+---------+------+-------------+----------------------+-------+
|123456|true      |2020-08-10 16:24:06   |[1, 2, 4]|allowed     |2020-08-12␣
↪14:43:14|true     |f     |2020-08-05   |2022-08-12 09:17:09.83|fine   |
|654321|false     |2020-08-10 16:25:00.784|[5]      |blocked     |2020-07-01␣
↪14:43:14|false    |m     |2020-06-27   |2022-08-12 09:17:09.83|faulty |
+------+----------+----------------------+---------+------------+-------------------
↪+---------+------+-------------+----------------------+-------+


output_df.printSchema()
root
 |-- index: integer (nullable = true)
 |-- is_enabled: boolean (nullable = true)
 |-- a_updated_at: timestamp (nullable = true)
 |-- items: array (nullable = true)
 |    |-- element: integer (containsNull = true)
 |-- block_status: string (nullable = true)
 |-- b_updated_at: timestamp (nullable = true)
 |-- has_email: boolean (nullable = false)
 |-- gender: string (nullable = true)
 |-- creation_date: date (nullable = true)
 |-- processed_at: timestamp (nullable = false)
 |-- comment: string (nullable = true)
```

CHAPTER

TWO

TABLE OF CONTENT

2.1 Installation

2.1.1 Via Pip

```
$ pip install spooq
```

2.1.2 Build wheel file

```
$ python setup.py sdist bdist_wheel
```

The output is stored as `dist/Spooq-<VERSION_NUMBER>-py3-none-any.whl` and `Spooq-<VERSION_NUMBER>.tar.gz`.

2.1.3 Build egg file

```
$ python setup.py bdist_egg
```

The output is stored as `dist/Spooq-<VERSION_NUMBER>-py3.8.egg`

2.1.4 Build zip file

```
$ zip_file_name="Spooq-$(grep "__version__" spooq/_version.py | cut -d " " -f 3 | tr -
↪d \").zip"
$ zip -r $zip_file_name spooq
```

The output is stored as `Spooq-<VERSION_NUMBER>.zip`.

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

For Submitting or Launching Spark:

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

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

Within Running Spark Session:

```
>>> sc.addFile("Spooq-<VERSION_NUMBER>.egg")
>>> import spooq
```

2.1.6 Install local repository as package

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

2.1.7 Install Spooq directly from git

```
$ pip install git+https://github.com/Breaka84/Spooq@master
```

2.1.8 Development, Testing, and Documenting

Please refer to *Contribute*.

2.2 Get Started

This section will guide you through a simple ETL pipeline built with Spooq to showcase how to use this library.

2.2.1 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"
      },
```

(continues on next page)

(continued from previous page)

```
      {
        "first_name": "Anaëlle",
        "last_name": null,
        "id": "18994"
      }
    ]
  },
  "meta": {
    "created_at_sec": "1547371284",
    "created_at_ms": "1547204429000",
    "version": "24"
  }
}
```

2.2.2 Application Code for Creating a User Table

```
from pyspark.sql import functions as F, types as T
from spooq.extractor import JSONExtractor
from spooq.transformer import Mapper, ThresholdCleaner, NewestByGroup
from spooq.loader import HiveLoader
from spooq.transformer import mapper_transformations as spq

users_mapping = [
    ("id",              "id",                    spq.to_num),
    ("guid",            "guid",                  "string"),
    ("forename",        "attributes.first_name", "string"),
    ("surename",        "attributes.last_name",  "string"),
    ("gender",          "attributes.gender",     spq.apply(func=F.lower)),
    ("has_email",       "attributes.email",      spq.has_value),
    ("has_university",  "attributes.university", spq.has_value),
    ("created_at",      "meta.created_at_ms",    spq.to_timestamp),
]

# Extract
source_df = JSONExtractor(input_path="tests/data/schema_v1/sequenceFiles").extract()

# Transform
mapped_df = Mapper(users_mapping).transform(source_df)
cleaned_df = ThresholdCleaner(
    thresholds={"created_at": {"min": "2019-01-01", "max": F.current_date(), "default
↪": None}},
    column_to_log_cleansed_values="cleansed_values",
    store_as_map=True,
).transform(mapped_df)
deduplicated_df = NewestByGroup(group_by="id", order_by="created_at").
↪transform(cleaned_df)

# Load
HiveLoader(
    db_name="users_and_friends",
    table_name="users",
    partition_definitions=[
        {"column_name": "dt", "column_type": "IntegerType", "default_value": 20200201}
    ],
```

(continues on next page)

(continued from previous page)

```
    repartition_size=10,
)
```

Table 1: Table **"user"**

id	guid	forename	surename	gender	has_email	has_university	create
1	799eb359-2e98-4526-a2ec-455b03e57b5c	Orran	Haug	m	false	false	null
N	...	...	...	...	...	...	...

2.2.3 Application Code for Creating a Friends_Mapping Table

```
from pyspark.sql import functions as F, types as T
from spooq.extractor import JSONExtractor
from spooq.transformer import Mapper, ThresholdCleaner, NewestByGroup, Exploder
from spooq.loader import HiveLoader
from spooq.transformer import mapper_transformations as spq

friends_mapping = [
    ("id",          "id",                  spq.to_num),
    ("guid",        "guid",                "string"),
    ("friend_id",   "friend.id",           spq.to_num),
    ("created_at",  "meta.created_at_ms",  spq.to_timestamp),
]

# Extract
source_df = JSONExtractor(input_path="tests/data/schema_v1/sequenceFiles").extract()

# Transform
deduplicated_friends_df = NewestByGroup(group_by="id", order_by="meta.created_at_ms").
↪transform(source_df)
exploded_friends_df = Exploder(path_to_array="attributes.friends", exploded_elem_name=
↪"friend").transform(deduplicated_friends_df)
mapped_friends_df = Mapper(mapping=friends_mapping).transform(exploded_friends_df)
cleaned_friends_df = ThresholdCleaner(
    thresholds={"created_at": {"min": "2019-01-01", "max": F.current_date(), "default
↪": None}},
    column_to_log_cleansed_values="cleansed_values",
    store_as_map=True,
).transform(mapped_friends_df)

friends_pipeline.set_loader(
    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,
    )
)
```

Table 2: Table **"friends_mapping"**

id	guid	friend_id	created_at
26	bd666d9d-9bb2-494e-8dc3-ab1c29a67ab2	8001	2019-05-29 23:25:27
26	bd666d9d-9bb2-494e-8dc3-ab1c29a67ab2	5623	2019-05-29 23:25:27
26	bd666d9d-9bb2-494e-8dc3-ab1c29a67ab2	17713	2019-05-29 23:25:27
65	2a5fd4e4-2cfa-41c6-9771-46c666e7c2eb	4428	null
65	2a5fd4e4-2cfa-41c6-9771-46c666e7c2eb	13011	null
N	...	...	...

2.2.4 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 Spooq for activities and steps which are not directly supported.

```
from pyspark.sql import functions as F, types as T
from spooq.extractor import JSONExtractor
from spooq.transformer import Mapper, ThresholdCleaner, NewestByGroup, Exploder
from spooq.loader import HiveLoader
from spooq.transformer import mapper_transformations as spq

mapping = [
    ("id",              "id",                    spq.to_num),
    ("guid",            "guid",                  "string"),
    ("forename",        "attributes.first_name", "string"),
    ("surename",        "attributes.last_name",  "string"),
    ("gender",          "attributes.gender",     spq.apply(func=F.lower)),
    ("has_email",       "attributes.email",      spq.has_value),
    ("has_university",  "attributes.university", spq.has_value),
    ("created_at",      "meta.created_at_ms",    spq.to_timestamp),
    ("friends",         "attributes.friends",    "as_is"),
]

# Transformations used by both output tables
common_df = JSONExtractor(input_path="tests/data/schema_v1/sequenceFiles").extract()
common_df = Mapper(mapping=mapping).transform(common_df)
common_df = ThresholdCleaner(
    thresholds={"created_at": {"min": "2019-01-01", "max": F.current_date(), "default
↪": None}},
    column_to_log_cleansed_values="cleansed_values",
    store_as_map=True,
).transform(common_df)
common_df = NewestByGroup(group_by="id", order_by="created_at").transform(common_df)
common_df.cache()

# Loading of users table
HiveLoader(
    db_name="users_and_friends",
    table_name="users",
    partition_definitions=[
        {"column_name": "dt", "column_type": "IntegerType", "default_value": 20200201}
    ],
    repartition_size=10,
```

(continues on next page)

(continued from previous page)

```
).load(common_df.drop("friends"))

# Transformations for friends_mapping table
friends_df = Exploder(path_to_array="friends", exploded_elem_name="friend").transform(
    common_df
)
friends_df = Mapper(
    mapping=[
        ("id",          "id",          "string"),
        ("guid",        "guid",        "string"),
        ("friend_id",   "friend.id",   spq.to_num),
        ("created_at",  "created_at",  "TimestampType"),
    ]
).transform(friends_df)

# Loading of friends_mapping table
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)
```

Dataflow Chart

Fig. 1: Typical Data Flow of a Spooq Data Pipeline

2.3 ETL Components

Fig. 2: Simplified UML Class Diagram

2.3.1 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**.

class Extractor

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.

Return type
DataFrame

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

JSON Files

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

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 spooq 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
DataFrame

Raises
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 `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
`DataFrame`

JDBC Source

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

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

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

Examples

```
>>> import spooq.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",
>>>     },
>>> )
>>>
>>> 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

DataFrame

class JDBCExtractorIncremental(*partition*, *jdbc_options*, *source_table*, *spooq_values_table*, *spooq_values_db='spooq_values'*, *spooq_values_partition_column='updated_at'*, *cache=True*)

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 spooq_values_table.

Examples

```
>>> import spooq.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",
>>>         "user": "read_only",
>>>         "password": "test123",
>>>     },
>>>     source_table="users",
>>>     spooq_values_table="spooq_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 spooq_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.
- **spooq_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.
- **spooq_values_db** (str, optional) – Defines the Database where the spooq_values_table is stored. Defaults to *'spooq_values'*.
- **spooq_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.

Return type

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

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

2.3.2 Transformers

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

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

Annotator (Load and Update Column Comments)

class AnnotatorMode(*value*)

Possible values: ['insert', 'upsert']

class MissingColumnHandling(*value*)

Possible values: ['raise_error', 'skip']

exception ColumnNotFound

load_comments_from_metastore_table(*sql_source_table_identifier: str*) → Dict[str, str]

Extracts comments from metadata stored in table (loaded by path or metastore).

Parameters

sql_source_table_identifier (`str`) – This is the fully qualified table name used for the `DESCRIBE <sql_source_table_identifier>` SQL command. Some examples:

- delta.`/path/to/table/files.delta`
- my_db.my_table
- my_catalog.my_db.my_table

Returns

Dictionary containing all extracted, non-null comments per column. ({<column_name>: <comment>})

Return type

`dict`

update_comment(*df: DataFrame*, *column_name: str*, *comment: str*, *annotator_mode:* AnnotatorMode = *AnnotatorMode.upsert*, *missing_column_handling:* MissingColumnHandling = *MissingColumnHandling.raise_error*, *logger: Optional[Logger] = None*) → DataFrame

Updates a single column's comment within the provided dataframe.

Parameters

- **df** (`DataFrame`) – The dataframe that contains the column to be updated.
- **column_name** (`str`) – The name of the column to be commented.
- **comment** (`str`) – The comment to apply to the referenced column.
- **annotator_mode** (*AnnotatorMode*, defaults to AnnotatorMode.upsert) – Defines if existing columns get overwritten.
- **missing_column_handling** (*MissingColumnHandling*, defaults to `MissingColumnHandling.raise_error`) – Defines how to behave when the `column_name` is not found in the dataframe.

Raises

ColumnNotFound – If column is missing from the dataframe and missing_column_handling is set to `raise_error`

Returns

Dataframe with updated metadata (comment) for the specified column.

Return type
DataFrame

class Annotator(*comments_mapping: Optional[Dict[str, str]] = None*, *mode:* AnnotatorMode = *AnnotatorMode.upsert*, *missing_column_handling:* MissingColumnHandling = *MissingColumnHandling.raise_error*, *sql_source_table_identifier: Optional[str] = None*)

Inserts or upserts column comments to dataframes. Can also be used just to fetch column comments from an existing table.

Parameters

- **comments_mapping** (dict, Defaults to {}) – Dictionary consisting of column names and comments
- **mode** (*AnnotatorMode*, Defaults to `AnnotatorMode.upsert`) – This mode defines how the transformer should react to existing column comments. `insert` will leave existing untouched while `upsert` overwrites them if a new comment is provided. Existing columns are defined as comments already defined in the dataframe or the sql_source_table!
- **missing_column_handling** (*MissingColumnHandling*, Defaults to `MissingColumnHandling.raise_error`) – This mode defines how the transformer should react to missing columns that are referenced in the `comments_mapping`.
- **sql_source_table_identifier** (str, Defaults to None) – The transformer will load any existing comments from the defined source table to the `comments_mapping`

Example

```
>>> "Fetching comments from an existing source table and applying those plus␣
↪explicitely defined comments"
>>> # Schema of sql_source_table (/tmp/path/to/my/silver_table.delta):
>>> #     col_A string COMMENT "Comment from sql_source_table",
>>> #     col_B string COMMENT "Comment from sql_source_table",
>>> #     col_Z string COMMENT "Comment from sql_source_table",
>>>
>>> # Schema of input dataframe
>>> #     col_A string,
>>> #     col_B string,
>>> #     col_Y string,
>>>
>>> from spooq.transformer import Annotator
>>> from spooq.transformer.annotator import AnnotatorMode, MissingColumnHandling
>>>
>>> spark.createDataFrame(
>>>     [],
>>>     schema='''
>>>         col_A string COMMENT "Comment from sql_source_table",
>>>         col_B string COMMENT "Comment from sql_source_table",
>>>         col_Z string COMMENT "Comment from sql_source_table"
>>>     '''
>>> ).write.format("delta").mode("overwrite").options(mergeSchema=True).save("/
↪tmp/path/to/my/silver_table.delta")
>>> input_df = spark.createDataFrame([], "col_A string, col_B string, col_Y␣
↪string")
>>> comments_mapping = {
>>>     "col_A": "Updated comment from comments_mapping",
>>>     "col_Y": "New comment from comments_mapping",
>>> }
```

(continues on next page)

(continued from previous page)

```
>>>
>>> output_df = Annotator(
>>>     comments_mapping=comments_mapping,
>>>     mode=AnnotatorMode.upsert,
>>>     missing_column_handling=MissingColumnHandling.raise_error,
>>>     sql_source_table_identifier="delta.`/tmp/path/to/my/silver_table.delta`",
>>> ).transform(input_df)
>>>
>>> print(json.dumps({col["name"]: col["metadata"]["comment"] for col in output_
↪df.schema.J["fields"]}, indent=2))
{
"col_A": "Updated comment from comments_mapping",
"col_B": "Comment from sql_source_table",
"col_Y": "New comment from comments_mapping"
}
```

Note:

Here are some use cases to use this transformer:

- Add explicitely defined comments from `comments_mapping` to dataframe within silver pipeline
- Apply comments from existing silver table to dataframe within gold pipeline
- Apply comments from `columns_mapping` within the *Mapper* transformer
- Apply comments from existing silver table to gold dataframe within the *Mapper* transformer
- Combine any of them

Raises

ColumnNotFound – If column is missing from the dataframe and missing_column_handling is set to `raise_error`:

transform(*input_df: DataFrame*) → DataFrame

Performs a transformation on a DataFrame.

Parameters

input_df (DataFrame) – Input DataFrame

Returns

Transformed DataFrame.

Return type

DataFrame

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

Exploder

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

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.
- **drop_rows_with_empty_array** (bool, (Defaults to True)) – By default Spark (and Spooq) drops rows which don't have any elements in the array which is being exploded. To work-around this, set *drop_rows_with_empty_array* to False.

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()` or `explode_outer()` methods of Spark are used internally, depending on the *drop_rows_with_empty_array* parameter.

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 (DataFrame) – Input DataFrame

Returns
Transformed DataFrame.

Return type
DataFrame

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

Sieve (Filter)

class Sieve(*filter_expression*)

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 (DataFrame) – Input DataFrame

Returns
Transformed DataFrame.

Return type
DataFrame

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

Mapper

Class

class MapperMode(*value*)

Possible values: [‘replace’, ‘append’, ‘prepend’, ‘rename_and_validate’]

class MissingColumnHandling(*value*)

Possible values: [‘raise_error’, ‘skip’, ‘nullify’]

exception DataTypeValidationFailed

exception ColumnMappingNotSupported

class Mapper(*mapping: List[Tuple]*, *ignore_ambiguous_columns: bool = False*, *missing_column_handling: Union[MissingColumnHandling, str] = MissingColumnHandling.raise_error*, *mode: Union[MapperMode, str] = MapperMode.replace*, *annotator_options: Optional[dict] = None*, ***kwargs*)

Selects, transforms, comments and casts or validates a DataFrame based on the provided mapping.

Examples

```
>>> from pyspark.sql import functions as F, types as T
>>> from spooq.transformer import Mapper
>>> from spooq.transformer import mapper_transformations as spq
>>>
>>> cmt_id = "Identifier for entity (UUID4)"
>>> cmt_type = "Type of entity ('type_a', 'type_b' or 'type_c')"
>>>
>>> mapping = [
>>>     ("id",            "data.relationships.food.data.id",  spq.to_str,       ␣
↪            cmt_id),
>>>     ("version",       "data.version",                     spq.to_int),
>>>     ("type",          "elem.attributes.type",             "string",         ␣
↪            cmt_type),
>>>     ("created_at",    "elem.attributes.created_at",       spq.to_timestamp),
>>>     ("created_on",    "elem.attributes.created_at",       spq.to_
↪timestamp(cast="date")),
>>>     ("processed_at",  F.current_timestamp(),              spq.as_is,
>>> ]
>>> mapper = Mapper(mapping=mapping)
>>> mapper.transform(input_df).printSchema()
root
 |-- id: string (nullable = true)
 |-- version: integer (nullable = true)
 |-- type: string (nullable = true)
 |-- created_at: timestamp (nullable = true)
 |-- created_on: date (nullable = true)
 |-- processed_at: timestamp (nullable = false)
```

Parameters

- **mapping** (`list` of `tuple` containing three or four elements, respectively) – This is the main parameter for this transformation. It gives information about the column names for the output DataFrame, the column names (paths) from the input DataFrame, their data types and optionally a column comment. Custom data types are also supported, which can clean, pivot, anonymize, … the data itself. Please have a look at the *spooq.transformer.mapper_transformations* module for more information.

- **missing_column_handling** (*MissingColumnHandling*, Defaults to `MissingColumnHandling.raise_error`) –

  Specifies how to proceed in case a source column does not exist in the source DataFrame:

  - **raise_error (default)**
    Raise an exception
  - **nullify**
    Create source column filled with null
  - **skip**
    Skip the mapping transformation

- **ignore_ambiguous_columns** (`bool`, Defaults to False) – It can happen that the input DataFrame has ambiguous column names (like "Key" vs "key") which will raise an exception with Spark when reading. This flag surpresses this exception and skips those affected columns.
- **mode** (*MapperMode*, Defaults to `MapperMode.replace`) – Defines whether the mapping should fully replace the schema of the input DataFrame or just add to it. Following modes are supported:
  - **replace**
    The output schema is the same as the provided mapping. => output schema: columns from mapping
  - **append**
    The columns provided in the mapping are added at the end of the input schema. If a column already exists in the input DataFrame, its position is kept. => output schema: input columns + columns from mapping
  - **prepend**
    The columns provided in the mapping are added at the beginning of the input schema. If a column already exists in the input DataFrame, its position is kept. => output schema: columns from mapping + input columns
  - **rename_and_validate**
    All built-in, custom transformations (except renaming) and casts are disabled. The Mapper only renames the columns and validates that the output data type is the same as the input data type. The transformation will fail if any spooq / custom transformations (except *as_is*) are defined! => output schema: columns from mapping
- **annotator_option** (`dict`, Defaults to {}) – Options that are passed as parameters to the Annotator instance used by the Mapper Transformer if comments are provided.

Keyword Arguments
ignore_missing_columns (`bool`, Defaults to False) – DEPRECATED: please use missing_column_handling instead!

Note: Let's talk about Mappings:

The mapping should be a list of tuples that contain all necessary information per column.

- **Column Name:** `str`
  Sets the name of the column in the resulting output DataFrame.
- **Source Path / Name / Column / Function:** `str`, `Column` **or** `functions`
  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. It is also possible to directly pass a PySpark Column which will get evaluated. This can

contain arbitrary logic supported by Spark. For example: `F.current_date()` or `F.when(F.col("size") == 180, F.lit("tall")).otherwise(F.lit("tiny"))`.

- **DataType: `str`, `DataType` or *`mapper_transformations`***
  DataTypes can be types from `pyspark.sql.types` (like T.StringType()), simple strings supported by PySpark (like "string") and custom transformations provided by spooq (like spq.to_timestamp). You can find more information about the transformations at https://spooq.rtfd.io/en/latest/transformer/mapper.html#module-spooq.transformer.mapper_transformations.
- **Comment: `str` (optional)**
  Applies a comment to the respective column, if provided.

Note: The available input columns can vary from batch to batch if you use schema inference (f.e. on json data) for the extraction. Via the parameter *missing_column_handling* you can specify a strategy on how to handle missing columns on the input DataFrame. It is advised to use the 'raise_error' option as it can uncover typos and bugs.

`transform`(*input_df: DataFrame*) → DataFrame

Performs a transformation on a DataFrame.

Parameters
`input_df` (`DataFrame`) – Input DataFrame

Returns
Transformed DataFrame.

Return type
`DataFrame`

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

Custom Transformations

This is a collection of module level functions to be applied to a DataFrame. These methods can be used with the *`Mapper`* transformer or directly within a `select` or `withColumn` statement.

All functions support following generic functionalities:

- `alt_src_cols`: Alternative source columns that will be used within a coalesce function if provided
- `cast`: Explicit casting after the transformation (sane defaults are set for each function)

`to_str`, `to_int`, `to_long`, `to_float`, `to_double` are convenience methods with a hardcoded cast that cannot be changed.

All examples assume following code has been executed before:

```
>>> from pyspark.sql import Row
>>> from pyspark.sql import functions as F, types as T
>>> from spooq.transformer import Mapper
>>> from spooq.transformer import mapper_transformations as spq
```

spooq.transformer.mapper_transformations.as_is([...])	Returns a renamed column without any casting.
spooq.transformer.mapper_transformations.to_num([...])	More robust conversion to number data types (Default: LongType).
spooq.transformer.mapper_transformations.to_bool([...])	More robust conversion to BooleanType.
spooq.transformer.mapper_transformations.to_timestamp([...])	More robust conversion to TimestampType (or as a formatted string).
spooq.transformer.mapper_transformations.str_to_array([...])	Splits a string into a list (ArrayType).
spooq.transformer.mapper_transformations.map_values([...])	Maps input values to specified output values.
spooq.transformer.mapper_transformations.meters_to_cm([...])	Converts meters to cm and casts the result to an IntegerType.
spooq.transformer.mapper_transformations.has_value([...])	Returns True if the source_column is
spooq.transformer.mapper_transformations.apply([...])	Applies a function / partial
spooq.transformer.mapper_transformations.to_json_string([...])	Returns a column as json compatible string.
spooq.transformer.mapper_transformations.to_str([...])	Convenience transformation that only casts to string.
spooq.transformer.mapper_transformations.to_int([...])	Syntactic sugar for calling `to_num(cast="int")`
spooq.transformer.mapper_transformations.to_long([...])	Syntactic sugar for calling `to_num(cast="long")`
spooq.transformer.mapper_transformations.to_float([...])	Syntactic sugar for calling `to_num(cast="float")`
spooq.transformer.mapper_transformations.to_double([...])	Syntactic sugar for calling `to_num(cast="double")`

spooq.transformer.mapper_transformations.as_is

as_is(*source_column: Optional[Union[str, Column]] = None*, *name: Optional[str] = None*, ***kwargs*) → Union[partial, Column]

Returns a renamed column without any casting. This is especially useful if you need to keep a complex data type (f.e. array, list or struct).

Parameters

- **source_column** (*str or Column*) – Input column. Can be a name, pyspark column or pyspark function
- **name** (*str, default -> derived from input column*) – Name of the output column. (`.alias(name)`)

Keyword Arguments

- **alt_src_cols** (*str, default -> no coalescing, only source_column*) – Coalesce with source_column and columns from this parameter.
- **cast** (*T.DataType(), default -> no casting, same return data type as input data type*) – Applies provided datatype on output column (`.cast(cast)`)

Examples

```
>>> input_df = spark.createDataFrame([
...     Row(friends=[Row(first_name="Gianni", id=3993, last_name="Weber"),
...                  Row(first_name="Arielle", id=17484, last_name="Greaves")]),
... ])
>>>
>>> input_df.select(spq.as_is("friends.first_name")).show(truncate=False)
+-----------------+
|[Gianni, Arielle]|
+-----------------+
>>>
>>> mapping = [("my_friends", "friends", spq.as_is)]
>>> output_df = Mapper(mapping).transform(input_df)
>>> output_df.show(truncate=False)
+-------------------------------------------------+
|my_friends                                       |
+-------------------------------------------------+
|[[Gianni, 3993, Weber], [Arielle, 17484, Greaves]]|
+-------------------------------------------------+
```

Returns

This method returns a suitable type depending on how it was called. This ensures compability with Spooq's mapper transformer - with or without explicit parameters - as well as direct calls via select, withColumn, where, ...

Return type

partial or Column

spooq.transformer.mapper_transformations.to_num

to_num(*source_column=None*, *name=None*, ***kwargs: Any*) → Union[partial, Column]

More robust conversion to number data types (Default: LongType). This method is able to additionally handle (compared to implicit Spark conversion):

- Preceding and/or trailing whitespace
- underscores as 'thousand' separators

Parameters

- **source_column** (*str or Column*) – Input column. Can be a name, pyspark column or pyspark function
- **name** (*str, default -> derived from input column*) – Name of the output column. (`.alias(name)`)

Keyword Arguments

- **alt_src_cols** (*str, default -> no coalescing, only source_column*) – Coalesce with source_column and columns from this parameter.
- **cast** (*T.DataType(), default -> T.LongType()*) – Applies provided datatype on output column (`.cast(cast)`)

Examples

```
>>> input_df = spark.createDataFrame(
...     [
...         Row(input_string="  123456 "),
...         Row(input_string="Hello"),
...         Row(input_string="123_456")
...     ], schema="input_key string"
... )
>>>
>>> input_df.select(spq.to_num("input_key")).show(truncate=False)
+---------+
|123456   |
|null     |
|123456   |
+---------+
>>>
>>> mapping = [
...     ("original_value",    "input_key", spq.as_is),
...     ("transformed_value", "input_key", spq.to_num)
... ]
>>> output_df = Mapper(mapping).transform(input_df)
>>> output_df.show(truncate=False)
+--------------+-----------------+
|original_value|transformed_value|
+--------------+-----------------+
|  123456      |123456           |
|Hello         |null             |
|123_456       |123456           |
+--------------+-----------------+
```

Returns

This method returns a suitable type depending on how it was called. This ensures compability with Spooq's mapper transformer - with or without explicit parameters - as well as direct calls via select, withColumn, where, ...

Return type

partial or Column

spooq.transformer.mapper_transformations.to_bool

to_bool(*source_column=None*, *name=None*, ***kwargs: Any*) → partial

More robust conversion to BooleanType. This method is able to additionally handle (compared to implicit Spark conversion):

- Preceding and/or trailing whitespace
- Define additional strings for true/false values ("on"/"off", "enabled"/"disabled" are added by default)

Parameters

- **source_column** (*str or Column*) – Input column. Can be a name, pyspark column or pyspark function
- **name** (*str, default -> derived from input column*) – Name of the output column. (`.alias(name)`)

Keyword Arguments

- **case_sensitive** (*Bool, default -> False*) – Defines whether the case for the additional true/false lookup values is considered
- **true_values** (*list, default -> ["on", "enabled"]*) – A list of values that should result in a `True` value if they are found in the source column
- **false_values** (*list, default -> ["off", "disabled"]*) – A list of values that should result in a `False` value if they are found in the source column
- **replace_default_values** (*Bool, default -> False*) – Defines whether additionally provided true/false values replace or extend the default list
- **alt_src_cols** (*str, default -> no coalescing, only source_column*) – Coalesce with source_column and columns from this parameter.
- **cast** (*T.DataType(), default -> T.BooleanType()*) – Applies provided datatype on output column (`.cast(cast)`)

Warning: Spark (and Spooq) handles number to boolean conversions depending on the input datatype! Please see this table for clarification:

Input		Result	
Value	Datatype	Cast to Boolean	spq.to_bool
-1	int	True	NULL
-1	str	NULL	NULL
0	int	False	False
0	str	False	False
1	int	True	True
1	str	True	True
100	int	True	NULL
100	str	NULL	NULL

Examples

```
>>> input_df = spark.createDataFrame(
...     [
...         Row(input_string="  false "),
...         Row(input_string="123"),
...         Row(input_string="1"),
...         Row(input_string="Enabled"),
...         Row(input_string="?"),
...         Row(input_string="n")
...     ], schema="input_key string"
... )
>>>
>>> input_df.select(spq.to_bool("input_key", false_values=["?"])).
↪show(truncate=False)
+---------+
|false    |
|null     |
|true     |
|false    |
|false    |
+---------+
>>>
>>> mapping = [
```

(continues on next page)

(continued from previous page)

```
...     ("original_value",    "input_key", spq.as_is),
...     ("transformed_value", "input_key", spq.to_bool(false_values=["?"]))
... ]
>>> output_df = Mapper(mapping).transform(input_df)
>>> output_df.show(truncate=False)
+--------------+-----------------+
|original_value|transformed_value|
+--------------+-----------------+
|  false       |false            |
|123           |null             |
|1             |true             |
|Enabled       |true             |
|?             |false            |
|n             |false            |
+--------------+-----------------+
```

Returns

This method returns a suitable type depending on how it was called. This ensures compability with Spooq's mapper transformer - with or without explicit parameters - as well as direct calls via select, withColumn, where, ...

Return type

partial or Column

spooq.transformer.mapper_transformations.to_timestamp

to_timestamp(*source_column=None*, *name=None*, ***kwargs: Any*) → partial

More robust conversion to TimestampType (or as a formatted string). This method supports following input types:

- Unix timestamps in seconds
- Unix timestamps in milliseconds
- Timestamps in any format supported by Spark
- Timestamps in any custom format (via `input_format`)
- Preceding and/or trailing whitespace

Parameters

- **source_column** (*str or Column*) – Input column. Can be a name, pyspark column or pyspark function
- **name** (*str, default -> derived from input column*) – Name of the output column. (`.alias(name)`)

Keyword Arguments

- **max_timestamp_sec** (*int, default -> 4102358400 (=> 2099-12-31 01:00:00)*) – Defines the range in which unix timestamps are still considered as seconds (compared to milliseconds)
- **input_format** (*[str, Bool], default -> False*) – Spooq tries to convert the input string with the provided pattern (via `F.unix_timestamp()`)
- **output_format** (*[str, Bool], default -> False*) – The output can be formatted according to the provided pattern (via `F.date_format()`)

- **min_timestamp_ms** (*int, default -> -62135514321000 (=> Year 1)*) – Defines the overall allowed range to keep the timestamps within Python's `datetime` library limits
- **max_timestamp_ms** (*int, default -> 253402210800000 (=> Year 9999)*) – Defines the overall allowed range to keep the timestamps within Python's `datetime` library limits
- **alt_src_cols** (*str, default -> no coalescing, only source_column*) – Coalesce with source_column and columns from this parameter.
- **cast** (*T.DataType(), default -> T.TimestampType()*) – Applies provided datatype on output column (`.cast(cast)`)

Warning:

- Timestamps in the range (-inf, -max_timestamp_sec) and (max_timestamp_sec, inf) are treated as milliseconds
- There is a time interval (1970-01-01 +- ~2.5 months) where we can not distinguish correctly between s and ms (e.g. 3974400000 would be treated as seconds (2095-12-11T00:00:00) as the value is smaller than MAX_TIMESTAMP_S, but it could also be a valid date in Milliseconds (1970-02-16T00:00:00)

Examples

```
>>> input_df = spark.createDataFrame(
...     [
...         Row(input_key="2020-08-12T12:43:14+0000"),
...         Row(input_key="1597069446"),
...         Row(input_key="1597069446000"),
...         Row(input_key="2020-08-12"),
...     ], schema="input_key string"
... )
>>>
>>> input_df.select(spq.to_timestamp("input_key")).show(truncate=False)
+-------------------+
|2020-08-12 14:43:14|
|2020-08-10 16:24:06|
|2020-08-10 16:24:06|
|2020-08-12 00:00:00|
+-------------------+
>>>
>>> mapping = [
...     ("original_value",    "input_key", spq.as_is),
...     ("transformed_value", "input_key", spq.to_timestamp)
... ]
>>> output_df = Mapper(mapping).transform(input_df)
>>> output_df.show(truncate=False)
+------------------------+-------------------+
|original_value          |transformed_value  |
+------------------------+-------------------+
|2020-08-12T12:43:14+0000|2020-08-12 14:43:14|
|1597069446              |2020-08-10 16:24:06|
|1597069446000           |2020-08-10 16:24:06|
|2020-08-12              |2020-08-12 00:00:00|
+------------------------+-------------------+
```

Returns

This method returns a suitable type depending on how it was called. This ensures compability with Spooq's mapper transformer - with or without explicit parameters - as well as direct calls via select, withColumn, where, ...

Return type

partial or Column

spooq.transformer.mapper_transformations.str_to_array

str_to_array(*source_column=None*, *name=None*, ***kwargs: Any*) → partial

Splits a string into a list (ArrayType).

Parameters

- **source_column** (*str or Column*) – Input column. Can be a name, pyspark column or pyspark function
- **name** (*str, default -> derived from input column*) – Name of the output column. (`.alias(name)`)

Keyword Arguments

- **alt_src_cols** (*str, default -> no coalescing, only source_column*) – Coalesce with source_column and columns from this parameter.
- **cast** (*T.DataType(), default -> T.StringType()*) – Applies provided datatype on the elements of the output array (`.cast(T.ArrayType(cast))`)

Examples

```
>>> input_df = spark.createDataFrame(
...     [
...         Row(input_key="[item1,item2,3]"),
...         Row(input_key="item1,it[e]m2,it em3"),
...         Row(input_key="    item1,   item2    ,   item3")
...     ], schema="input_key string"
... )
>>>
>>> input_df.select(spq.str_to_array("input_key")).show(truncate=False)
+------------------------+
|[item1, item2, 3]       |
|[item1, it[e]m2, it em3]|
|[item1, item2, item3]   |
+------------------------+
>>>
>>> mapping = [
...     ("original_value",    "input_key", spq.as_is),
...     ("transformed_value", "input_key", spq.str_to_array)
... ]
>>> output_df = Mapper(mapping).transform(input_df)
>>> output_df.printSchema()
root
 |-- original_value: string (nullable = true)
 |-- transformed_value: array (nullable = true)
 |    |-- element: string (containsNull = true)
>>> output_df.show(truncate=False)
+------------------------------+------------------------+
|original_value                |transformed_value       |
```

(continues on next page)

(continued from previous page)

```
+------------------------------+-----------------------+
|[item1,item2,3]               |[item1, item2, 3]      |
|item1,it[e]m2,it em3          |[item1, it[e]m2, it em3]|
|    item1,   item2    ,   item3|[item1, item2, item3]  |
+------------------------------+-----------------------+
```

Returns

This method returns a suitable type depending on how it was called. This ensures compability with Spooq's mapper transformer - with or without explicit parameters - as well as direct calls via select, withColumn, where, ...

Return type

partial or Column

spooq.transformer.mapper_transformations.map_values

map_values(*source_column=None*, *name=None*, ***kwargs: Any*) → partial

Maps input values to specified output values.

Parameters

- **source_column** (*str or Column*) – Input column. Can be a name, pyspark column or pyspark function
- **name** (*str, default -> derived from input column*) – Name of the output column. (`.alias(name)`)

Keyword Arguments

- **mapping** (*dict*) – Dictionary containing lookup / substitute value pairs.
- **default** (*[str, Column, Any], default -> "source_column"*) – Defines what will be returned if no matching lookup value was found.
- **ignore_case** (*bool, default -> True*) – Only relevant for "equals" and "sql_like" comparison operators.
- **pattern_type** (*str, default -> "equals"*) – Please choose among ['equals', 'regex' and 'sql_like'] for the comparison of input value and mapping key.
- **alt_src_cols** (*str, default -> no coalescing, only source_column*) – Coalesce with source_column and columns from this parameter.
- **cast** (*T.DataType(), default -> T.StringType()*) – Applies provided datatype on output column (`.cast(cast)`)

Hint: Maybe this table helps you to better understand what happens behind the curtains:

lookup	substitute	mode | Internal Spark Logic
whitelist	allowlist	**“equals” | F.when(** F.col(“input_column”) == “whitelist”, F.lit(“allowlist”)).otherwise(F.col(“input_column”))
%whitelist%	allowlist	**“sql_like” | F.when(** F.col(“input_column”).like(“%whitelist%”, F.lit(“allowlist”)).otherwise(F.col(“input_column”))
.*whitelist.*	allowlist	**“regex ” | F.when(** F.col(“input_column”).rlike(“.*whitelist.*”, F.lit(“allowlist”)).otherwise(F.col(“input_column”))

Examples

```
>>> input_df = spark.createDataFrame(
...     [
...         ("allowlist", ),
...         ("WhiteList", ),
...         ("blocklist", ),
...         ("blacklist", ),
...         ("Blacklist", ),
...         ("Shoppinglist", ),
...     ], schema="input_key string"
... )
>>> substitute_mapping = {"whitelist": "allowlist", "blacklist": "blocklist"}
>>>
>>> input_df.select(spq.map_values("input_key", mapping=substitute_mapping)).
↪show(truncate=False)
+------------+
|allowlist   |
|allowlist   |
|blocklist   |
|blocklist   |
|blocklist   |
|Shoppinglist|
+------------+
>>>
>>> mapping = [
...     ("original_value",    "input_key", spq.as_is),
...     ("transformed_value", "input_key", spq.map_values(mapping=substitute_
```

(continues on next page)

(continued from previous page)

```
↪mapping))
... ]
>>> output_df = Mapper(mapping).transform(input_df)
>>> output_df.show(truncate=False)
+--------------+----------------+
|original_value|transformed_value|
+--------------+----------------+
|allowlist     |allowlist       |
|WhiteList     |allowlist       |
|blocklist     |blocklist       |
|blacklist     |blocklist       |
|Blacklist     |blocklist       |
|Shoppinglist  |Shoppinglist    |
+--------------+----------------+
```

Returns

This method returns a suitable type depending on how it was called. This ensures compability with Spooq's mapper transformer - with or without explicit parameters - as well as direct calls via select, withColumn, where, ...

Return type

partial or Column

spooq.transformer.mapper_transformations.meters_to_cm

meters_to_cm(*source_column=None*, *name=None*, ***kwargs: Any*) → partial

Converts meters to cm and casts the result to an IntegerType.

Parameters

- **source_column** (*str or Column*) – Input column. Can be a name, pyspark column or pyspark function
- **name** (*str, default -> derived from input column*) – Name of the output column. (`.alias(name)`)

Keyword Arguments

- **alt_src_cols** (*str, default -> no coalescing, only source_column*) – Coalesce with source_column and columns from this parameter.
- **cast** (*T.DataType(), default -> T.IntegerType()*) – Applies provided datatype on output column (`.cast(cast)`)

Examples

```
>>> input_df = spark.createDataFrame([
...     Row(size_in_m=1.80),
...     Row(size_in_m=1.65),
...     Row(size_in_m=2.05)
... ])
>>>
>>> input_df.select(spq.meters_to_cm("size_in_m")).show(truncate=False)
+---------+
|180      |
|165      |
|204      |
```

(continues on next page)

(continued from previous page)

```
+---------+
>>>
>>> mapping = [
...     ("original_value",  "size_in_m",  spq.as_is),
...     ("size_in_cm",      "size_in_m",  spq.meters_to_cm),
... ]
>>> output_df = Mapper(mapping).transform(input_df)
>>> output_df.show(truncate=False)
+--------------+----------+
|original_value|size_in_cm|
+--------------+----------+
|1.8           |180       |
|1.65          |165       |
|2.05          |204       |
+--------------+----------+
```

Returns

This method returns a suitable type depending on how it was called. This ensures compability with Spooq's mapper transformer - with or without explicit parameters - as well as direct calls via select, withColumn, where, ...

Return type

partial or Column

spooq.transformer.mapper_transformations.has_value

has_value(*source_column=None*, *name=None*, ***kwargs: Any*) → partial

Returns True if the source_column is

- not NULL and
- not "" (empty string)
- otherwise it returns False

Warning: This means that it will return True for values which would indicate a False value. Like "false" or 0!!!

Parameters

- **source_column** (*str or Column*) – Input column. Can be a name, pyspark column or pyspark function
- **name** (*str, default -> derived from input column*) – Name of the output column. (`.alias(name)`)

Keyword Arguments

- **alt_src_cols** (*str, default -> no coalescing, only source_column*) – Coalesce with source_column and columns from this parameter.
- **cast** (*T.DataType(), default -> T.BooleanType()*) – Applies provided datatype on output column (`.cast(cast)`)

Examples

```
>>> input_df = spark.createDataFrame(
...     [
...         Row(input_key=False),
...         Row(input_key=None),
...         Row(input_key="some text"),
...         Row(input_key="")
...     ], schema="input_key string"
... )
>>>
>>> input_df.select(spq.has_value("input_key")).show(truncate=False)
+---------+
|true     |
|false    |
|true     |
|false    |
+---------+
>>>
>>> mapping = [
...     ("original_value",      "input_key",  spq.as_is),
...     ("has_value",  "input_key",  spq.has_value)
... ]
>>> output_df = Mapper(mapping).transform(input_df)
>>> output_df.show(truncate=False)
+--------------+---------+
|original_value|has_value|
+--------------+---------+
|false         |true     |
|null          |false    |
|some text     |true     |
|              |false    |
+--------------+---------+
```

Returns

This method returns a suitable type depending on how it was called. This ensures compability with Spooq's mapper transformer - with or without explicit parameters - as well as direct calls via select, withColumn, where, ...

Return type

partial or Column

spooq.transformer.mapper_transformations.apply

apply(*source_column=None*, *name=None*, ***kwargs: Any*) → partial

Applies a function / partial

Parameters

- **source_column** (*str or Column*) – Input column. Can be a name, pyspark column or pyspark function
- **name** (*str, default -> derived from input column*) – Name of the output column. (`.alias(name)`)

Keyword Arguments

- **func** (*Callable*) – Function that takes the source column as single argument

- **alt_src_cols** (*str, default -> no coalescing, only source_column*) – Coalesce with source_column and columns from this parameter.
- **cast** (*T.DataType(), default -> no casting*) – Applies provided datatype on output column (`.cast(cast)`)

Examples

```
>>> input_df = spark.createDataFrame(
...     [
...         ("F", ),
...         ("f", ),
...         ("x", ),
...         ("X", ),
...         ("m", ),
...         ("M", ),
...     ], schema="input_key string"
... )
>>>
>>> input_df.select(spq.apply("input_key", func=F.lower)).show(truncate=False)
+---------+
|f        |
|f        |
|x        |
|x        |
|m        |
|m        |
+---------+
>>>
>>> mapping = [
...     ("original_value",    "input_key", spq.as_is),
...     ("transformed_value", "input_key", spq.apply(func=F.lower))
... ]
>>> output_df = Mapper(mapping).transform(input_df)
>>> output_df.show(truncate=False)
+--------------+-----------------+
|original_value|transformed_value|
+--------------+-----------------+
|F             |f                |
|f             |f                |
|x             |x                |
|X             |x                |
|m             |m                |
|M             |m                |
+--------------+-----------------+
```

```
>>> input_df = spark.createDataFrame(
...     [
...         ("sarajishvilileqso@gmx.at", ),
...         ("jnnqn@astrinurdin.art", ),
...         ("321aw@hotmail.com", ),
...         ("techbrenda@hotmail.com", ),
...         ("sdsxcx@gmail.com", ),
...     ], schema="input_key string"
... )
...
>>> def _has_hotmail(source_column):
```

(continues on next page)

(continued from previous page)

```
...     return F.when(
...         source_column.cast(T.StringType()).endswith("@hotmail.com"),
...         F.lit(True)
...     ).otherwise(F.lit(False))
...
>>> mapping = [
...     ("original_value", "input_key", spq.as_is),
...     ("over_sixty",     "input_key", spq.apply(func=_has_hotmail))
... ]
>>> output_df = Mapper(mapping).transform(input_df)
>>> output_df.show(truncate=False)
+-----------------------+----------+
|original_value         |over_sixty|
+-----------------------+----------+
|sarajishvilileqso@gmx.at|false     |
|jnnqn@astrinurdin.art  |false     |
|321aw@hotmail.com      |true      |
|techbrenda@hotmail.com |true      |
|sdsxcx@gmail.com       |false     |
+-----------------------+----------+
```

Returns

This method returns a suitable type depending on how it was called. This ensures compability with Spooq's mapper transformer - with or without explicit parameters - as well as direct calls via select, withColumn, where, ...

Return type

partial or Column

spooq.transformer.mapper_transformations.to_json_string

`to_json_string`(*source_column=None*, *name=None*, ***kwargs: Any*) → partial

Returns a column as json compatible string. Nested hierarchies are supported. This function also supports NULL and strings as input in comparison to Spark's built-in `to_json`. The unicode representation of a column will be returned if an error occurs.

Parameters

- **source_column** (*str or Column*) – Input column. Can be a name, pyspark column or pyspark function
- **name** (*str, default -> derived from input column*) – Name of the output column. (`.alias(name)`)

Keyword Arguments

- **alt_src_cols** (*str, default -> no coalescing, only source_column*) – Coalesce with source_column and columns from this parameter.
- **cast** (*T.DataType(), default -> no casting, same return data type as input data type*) – Applies provided datatype on output column (`.cast(cast)`)

Examples

```
>>> input_df = spark.createDataFrame([
...     Row(friends=[Row(first_name="Gianni", id=3993, last_name="Weber"),
...                  Row(first_name="Arielle", id=17484, last_name="Greaves")]),
... ])
>>>
>>> input_df.select(spq.to_json_string("friends")).show(truncate=False)
+-------------------------------------------------------------------------------
↪-----------------------------------------+
|[{"first_name": "Gianni", "id": 3993, "last_name": "Weber"}, {"first_name":
↪"Arielle", "id": 17484, "last_name": "Greaves"}]|
+-------------------------------------------------------------------------------
↪-----------------------------------------+
>>>
>>> mapping = [("friends_json", "friends", spq.to_json_string)]
>>> output_df = Mapper(mapping).transform(input_df)
>>> output_df.show(truncate=False)
+-------------------------------------------------------------------------------
↪-----------------------------------------+
|friends_json
↪                                          |
+-------------------------------------------------------------------------------
↪-----------------------------------------+
|[{"first_name": "Gianni", "id": 3993, "last_name": "Weber"}, {"first_name":
↪"Arielle", "id": 17484, "last_name": "Greaves"}]|
+-------------------------------------------------------------------------------
↪-----------------------------------------+
>>> input_df.select(spq.to_json_string("friends.first_name")).
↪show(truncate=False)
+--------------------+
|first_name          |
+--------------------+
|['Gianni', 'Arielle']|
+--------------------+
```

Returns

This method returns a suitable type depending on how you called it. This ensures compability with Spooq's mapper transformer - with or without explicit parameters as well as direct calls via select, withColumn, where, ...

Return type

partial or Column

spooq.transformer.mapper_transformations.to_str

to_str(*source_column=None*, *name=None*, ***kwargs: Any*) → Union[partial, Column]

Convenience transformation that only casts to string.

Parameters

- **source_column** (*str or Column*) – Input column. Can be a name, pyspark column or pyspark function
- **name** (*str, default -> derived from input column*) – Name of the output column. (`.alias(name)`)

Keyword Arguments
alt_src_cols (*str, default -> no coalescing, only source_column*) – Coalesce with source_column and columns from this parameter.

Examples

```
>>> input_df = spark.createDataFrame(
...     [
...         Row(input_key=123456),
...         Row(input_key=-123456),
...     ], schema="input_key int"
... )
>>>
>>> input_df.select(spq.to_str("input_key")).show(truncate=False)
+---------+
|123456   |
|-123456  |
+---------+
>>>
>>> mapping = [
...     ("original_value",    "input_key", spq.as_is),
...     ("transformed_value", "input_key", spq.to_str)
... ]
>>> output_df = Mapper(mapping).transform(input_df)
>>> output_df.show(truncate=False)
+--------------+-----------------+
|original_value|transformed_value|
+--------------+-----------------+
|123456        |123456           |
|-123456       |-123456          |
+--------------+-----------------+
```

Returns
This method returns a suitable type depending on how it was called. This ensures compability with Spooq's mapper transformer - with or without explicit parameters - as well as direct calls via select, withColumn, where, ...

Return type
partial or Column

spooq.transformer.mapper_transformations.to_int

to_int(*source_column=None*, *name=None*, ***kwargs: Any*) → Union[partial, Column]

Syntactic sugar for calling `to_num(cast="int")`

Parameters

- **source_column** (*str or Column*) – Input column. Can be a name, pyspark column or pyspark function
- **name** (*str, default -> derived from input column*) – Name of the output column. (`.alias(name)`)

Keyword Arguments
alt_src_cols (*str, default -> no coalescing, only source_column*) – Coalesce with source_column and columns from this parameter.

Returns
This method returns a suitable type depending on how it was called. This ensures compability with Spooq's mapper transformer - with or without explicit parameters - as well as direct calls via select, withColumn, where, ...

Return type
partial or Column

spooq.transformer.mapper_transformations.to_long

`to_long`(*source_column=None*, *name=None*, ***kwargs: Any*) → Union[partial, Column]

Syntactic sugar for calling `to_num(cast="long")`

Parameters

- **source_column** (*str or Column*) – Input column. Can be a name, pyspark column or pyspark function
- **name** (*str, default -> derived from input column*) – Name of the output column. (`.alias(name)`)

Keyword Arguments
alt_src_cols (*str, default -> no coalescing, only source_column*) – Coalesce with source_column and columns from this parameter.

Returns
This method returns a suitable type depending on how it was called. This ensures compability with Spooq's mapper transformer - with or without explicit parameters - as well as direct calls via select, withColumn, where, ...

Return type
partial or Column

spooq.transformer.mapper_transformations.to_float

`to_float`(*source_column=None*, *name=None*, ***kwargs: Any*) → Union[partial, Column]

Syntactic sugar for calling `to_num(cast="float")`

Parameters

- **source_column** (*str or Column*) – Input column. Can be a name, pyspark column or pyspark function
- **name** (*str, default -> derived from input column*) – Name of the output column. (`.alias(name)`)

Keyword Arguments
alt_src_cols (*str, default -> no coalescing, only source_column*) – Coalesce with source_column and columns from this parameter.

Returns
This method returns a suitable type depending on how it was called. This ensures compability with Spooq's mapper transformer - with or without explicit parameters - as well as direct calls via select, withColumn, where, ...

Return type
partial or Column

spooq.transformer.mapper_transformations.to_double

to_double(*source_column=None*, *name=None*, ***kwargs: Any*) → Union[partial, Column]

Syntactic sugar for calling `to_num(cast="double")`

Parameters

- **source_column** (*str or Column*) – Input column. Can be a name, pyspark column or pyspark function
- **name** (*str, default -> derived from input column*) – Name of the output column. (`.alias(name)`)

Keyword Arguments

alt_src_cols (*str, default -> no coalescing, only source_column*) – Coalesce with source_column and columns from this parameter.

Returns

This method returns a suitable type depending on how it was called. This ensures compability with Spooq's mapper transformer - with or without explicit parameters - as well as direct calls via select, withColumn, where, ...

Return type

partial or Column

Custom Mapping Functions as Strings [DEPRECATED]

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!

spooq.transformer.mapper_custom_data_types._generate_select_expression_for_IntBoolean(...)	Deprecated!
spooq.transformer.mapper_custom_data_types._generate_select_expression_for_IntNull(...)	Deprecated!
spooq.transformer.mapper_custom_data_types._generate_select_expression_for_StringBoolean(...)	Deprecated!
spooq.transformer.mapper_custom_data_types._generate_select_expression_for_StringNull(...)	Deprecated!
spooq.transformer.mapper_custom_data_types._generate_select_expression_for_TimestampMonth(...)	Deprecated!
spooq.transformer.mapper_custom_data_types._generate_select_expression_for_as_is(...)	Deprecated!
spooq.transformer.mapper_custom_data_types._generate_select_expression_for_extended_string_to_boolean(...)	Deprecated!
spooq.transformer.mapper_custom_data_types._generate_select_expression_for_extended_string_to_date(...)	Deprecated!
spooq.transformer.mapper_custom_data_types._generate_select_expression_for_extended_string_to_double(...)	Deprecated!
spooq.transformer.mapper_custom_data_types._generate_select_expression_for_extended_string_to_float(...)	Deprecated!
spooq.transformer.mapper_custom_data_types._generate_select_expression_for_extended_string_to_int(...)	Deprecated!
spooq.transformer.mapper_custom_data_types._generate_select_expression_for_extended_string_to_long(...)	Deprecated!
spooq.transformer.mapper_custom_data_types._generate_select_expression_for_extended_string_to_timestamp(...)	Deprecated!
spooq.transformer.mapper_custom_data_types._generate_select_expression_for_extended_string_unix_timestamp_ms_to_date(...)	Deprecated!
spooq.transformer.mapper_custom_data_types._generate_select_expression_for_extended_string_unix_timestamp_ms_to_timestamp(...)	Deprecated!
spooq.transformer.mapper_custom_data_types._generate_select_expression_for_has_value(...)	Deprecated!
spooq.transformer.mapper_custom_data_types._generate_select_expression_for_json_string(...)	Deprecated!
spooq.transformer.mapper_custom_data_types._generate_select_expression_for_keep(...)	Deprecated!
spooq.transformer.mapper_custom_data_types._generate_select_expression_for_meters_to_cm(...)	Deprecated!
spooq.transformer.mapper_custom_data_types._generate_select_expression_for_no_change(...)	Deprecated!
spooq.transformer.mapper_custom_data_types._generate_select_expression_for_timestamp_ms_to_ms(...)	Deprecated!

spooq.transformer.mapper_custom_data_types._generate_select_expression_for_IntBoolean

_generate_select_expression_for_IntBoolean(*source_column*, *name*)

Deprecated!

Please use :dt.:~*spooq.transformer.mapper_transformations.has_value* instead.

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 spooq.transformer import Mapper
>>>
>>> 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`.

spooq.transformer.mapper_custom_data_types._generate_select_expression_for_IntNull

_generate_select_expression_for_IntNull(*source_column*, *name*)

Deprecated!

Please just use `F.lit(None)` as source_column and `"int"` as data_type instead!

spooq.transformer.mapper_custom_data_types._generate_select_expression_for_StringBoolean

_generate_select_expression_for_StringBoolean(*source_column*, *name*)

Deprecated!

Please use :dt.:~*spooq.transformer.mapper_transformations.has_value* instead.

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 spooq.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')]
```

spooq.transformer.mapper_custom_data_types._generate_select_expression_for_StringNull

_generate_select_expression_for_StringNull(*source_column*, *name*)

Deprecated!

Please just use `F.lit(None)` as source_column and `"string"` as data_type instead!

spooq.transformer.mapper_custom_data_types._generate_select_expression_for_TimestampMonth

_generate_select_expression_for_TimestampMonth(*source_column*, *name*)

Deprecated!

Please use :dt.:~*spooq.transformer.mapper_transformations.apply* instead.

spooq.transformer.mapper_custom_data_types._generate_select_expression_for_as_is

_generate_select_expression_for_as_is(*source_column*, *name*)

Deprecated!

Please use :dt.:~*spooq.transformer.mapper_transformations.as_is* directly instead.

spooq.transformer.mapper_custom_data_types._generate_select_expression_for_extended_string_to_boolean

_generate_select_expression_for_extended_string_to_boolean(*source_column*, *name*)

Deprecated!

Please use :dt.:~*spooq.transformer.mapper_transformations.to_bool* directly instead.

spooq.transformer.mapper_custom_data_types._generate_select_expression_for_extended_string_to_date

_generate_select_expression_for_extended_string_to_date(*source_column*, *name*)

Deprecated!

Please use :dt.:~*spooq.transformer.mapper_transformations.to_timestamp* and cast to date instead.

spooq.transformer.mapper_custom_data_types._generate_select_expression_for_extended_string_to_double

_generate_select_expression_for_extended_string_to_double(*source_column*, *name*)

Deprecated!

Please use :dt.:~*spooq.transformer.mapper_transformations.to_num* directly instead and define the `cast` as `"double"`.

spooq.transformer.mapper_custom_data_types._generate_select_expression_for_extended_string_to_float

_generate_select_expression_for_extended_string_to_float(*source_column*, *name*)

Deprecated!

Please use :dt.:~*spooq.transformer.mapper_transformations.to_num* directly instead and define the `cast` as `"float"`.

spooq.transformer.mapper_custom_data_types._generate_select_expression_for_extended_string_to_int

_generate_select_expression_for_extended_string_to_int(*source_column*, *name*)

Deprecated!

Please use :dt.:~*spooq.transformer.mapper_transformations.to_num* directly instead and define the `cast` as `"int"`.

spooq.transformer.mapper_custom_data_types._generate_select_expression_for_extended_string_to_long

_generate_select_expression_for_extended_string_to_long(*source_column*, *name*)

Deprecated!

Please use :dt.:~*spooq.transformer.mapper_transformations.to_num* directly instead and define the `cast` as `"long"`.

spooq.transformer.mapper_custom_data_types._generate_select_expression_for_extended_string_to_timesta

_generate_select_expression_for_extended_string_to_timestamp(*source_column*, *name*)

Deprecated!

Please use :dt.:~*spooq.transformer.mapper_transformations.to_timestamp* instead.

spooq.transformer.mapper_custom_data_types._generate_select_expression_for_extended_string_unix_times

_generate_select_expression_for_extended_string_unix_timestamp_ms_to_date(*source_column*, *name*)

Deprecated!

Please use :dt.:~*spooq.transformer.mapper_transformations.to_timestamp* and cast to date instead.

spooq.transformer.mapper_custom_data_types._generate_select_expression_for_extended_string_unix_times

_generate_select_expression_for_extended_string_unix_timestamp_ms_to_timestamp(*source_column*, *name*)

Deprecated!

Please use :dt.:~*spooq.transformer.mapper_transformations.to_timestamp* instead.

spooq.transformer.mapper_custom_data_types._generate_select_expression_for_has_value

_generate_select_expression_for_has_value(*source_column*, *name*)

Deprecated!

Please use :dt.:~*spooq.transformer.mapper_transformations.has_value* directly instead.

spooq.transformer.mapper_custom_data_types._generate_select_expression_for_json_string

_generate_select_expression_for_json_string(*source_column*, *name*)

Deprecated!

Please use :dt.:~*spooq.transformer.mapper_transformations.to_json_string* directly instead.

spooq.transformer.mapper_custom_data_types._generate_select_expression_for_keep

_generate_select_expression_for_keep(*source_column*, *name*)

Deprecated!

Please use :dt.:~*spooq.transformer.mapper_transformations.as_is* directly instead.

spooq.transformer.mapper_custom_data_types._generate_select_expression_for_meters_to_cm

_generate_select_expression_for_meters_to_cm(*source_column*, *name*)

Deprecated!

Please use :dt.:~*spooq.transformer.mapper_transformations.meters_to_cm* directly instead.

spooq.transformer.mapper_custom_data_types._generate_select_expression_for_no_change

_generate_select_expression_for_no_change(*source_column*, *name*)

Deprecated!

Please use :dt.:~*spooq.transformer.mapper_transformations.as_is* directly instead.

spooq.transformer.mapper_custom_data_types._generate_select_expression_for_timestamp_ms_to_ms

_generate_select_expression_for_timestamp_ms_to_ms(*source_column*, *name*)

Deprecated!

Please just use :dt.:~*spooq.transformer.mapper_transformations.as_is* with `"long"` as data_type instead. This method doesn't do any cleansing anymore!

spooq.transformer.mapper_custom_data_types._generate_select_expression_for_timestamp_ms_to_s

_generate_select_expression_for_timestamp_ms_to_s(*source_column*, *name*)

Deprecated!

Please use :dt.:~*spooq.transformer.mapper_transformations.apply* with a lambda instead.

spooq.transformer.mapper_custom_data_types._generate_select_expression_for_timestamp_s_to_ms

_generate_select_expression_for_timestamp_s_to_ms(*source_column*, *name*)

Deprecated!

Please use :dt.:~*spooq.transformer.mapper_transformations.apply* with a lambda instead.

spooq.transformer.mapper_custom_data_types._generate_select_expression_for_timestamp_s_to_s

_generate_select_expression_for_timestamp_s_to_s(*source_column*, *name*)

Deprecated!

Please just use :dt.:~*spooq.transformer.mapper_transformations.as_is* with `"long"` as data_type instead. This method doesn't do any cleansing anymore!

spooq.transformer.mapper_custom_data_types._generate_select_expression_for_unix_timestamp_ms_to_spa

_generate_select_expression_for_unix_timestamp_ms_to_spark_timestamp(*source_column*, *name*)

Deprecated!

Please use :dt.:~*spooq.transformer.mapper_transformations.to_timestamp* directly instead.

spooq.transformer.mapper_custom_data_types._generate_select_expression_without_casting

_generate_select_expression_without_casting(*source_column*, *name*)

Deprecated!

Please use :dt.:~*spooq.transformer.mapper_transformations.as_is* directly instead.

Threshold-based Cleaner

class ThresholdCleaner(*thresholds={}*, *column_to_log_cleansed_values=None*, *store_as_map=False*)

Cleanes values based on defined boundaries and optionally logs the original values. The valid value ranges are provided via dictionary for each column to be cleaned. Boundaries can either be of static or dynamic nature that resolve to numbers, timetamps or dates.

Examples

```
>>> from pyspark.sql import functions as F
>>> from spooq.transformer import ThresholdCleaner
>>> transformer = ThresholdCleaner(
>>>     thresholds={
>>>         "created_at": {
>>>             "min": F.date_sub(F.current_date(), 365 * 10),  # Ten years ago
>>>             "max": F.current_date()
>>>         },
>>>         "size_cm": {
>>>             "min": 70,
>>>             "max": 250,
>>>             "default": None
>>>         },
>>>     }
>>> )
```

```
>>> from spooq.transformer import ThresholdCleaner
>>> from pyspark.sql import Row
>>>
>>> input_df = spark.createDataFrame([
>>>     Row(id=0, integers=-5, doubles=-0.75),
>>>     Row(id=1, integers= 5, doubles= 1.25),
>>>     Row(id=2, integers=15, doubles= 0.67),
>>> ])
>>> transformer = ThresholdCleaner(
>>>     thresholds={
>>>         "integers": {"min":  0, "max": 10},
>>>         "doubles":  {"min": -1, "max":  1}
>>>     },
>>>     column_to_log_cleansed_values="cleansed_values_threshold",
>>>     store_as_map=True,
>>> )
>>> output_df = transformer.transform(input_df)
>>> output_df.show(truncate=False)
+---+--------+-----------+--------------------+
|id |integers|doubles|cleansed_values_threshold|
+---+--------+-------+------------------------+
|0  |null    |-0.75  |{integers -> -5}        |
|1  |5       |null   |{doubles -> 1.25}       |
```

(continues on next page)

(continued from previous page)

```
|2  |null    |0.67   |{integers -> 15}         |
+---+--------+-------+-------------------------+
>>> output_df.printSchema()
 |-- id: long (nullable = true)
 |-- integers: long (nullable = true)
 |-- doubles: double (nullable = true)
 |-- cleansed_values_threshold: map (nullable = false)
 |    |-- key: string
 |    |-- value: string (valueContainsNull = true)
```

Parameters

- **thresholds** (dict) – Dictionary containing column names and respective valid ranges
- **column_to_log_cleansed_values** (str, Defaults to None) – Defines a column in which the original (uncleansed) value will be stored in case of cleansing. If no column name is given, nothing will be logged.
- **store_as_map** (bool, Defaults to False) – Specifies if the logged cleansed values should be stored in a column as MapType with stringified values or as StructType with the original respective data types.

Note: Following cleansing rule attributes per column are supported:

- **min, mandatory: str, int, Column, pyspark.sql.functions**
  A number or timestamp/date which serves as the lower limit for allowed values. Values below this threshold will be cleansed. Supports literals, Spark functions and Columns.
- **max, mandatory: str, int, Column, pyspark.sql.functions**
  A number or timestamp/date which serves as the upper limit for allowed values. Values above this threshold will be cleansed. Supports literals, Spark functions and Columns.
- **default, defaults to None: str, int, Column, pyspark.sql.functions**
  If a value gets cleansed it gets replaced with the provided default value. Supports literals, Spark functions and Columns.

The between() method is used internally.

Returns
The transformed DataFrame

Return type
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 (DataFrame) – Input DataFrame

Returns
Transformed DataFrame.

Return type
DataFrame

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

Base Class

This abstract class provides the functionality to log any cleansed values into a separate column that contains a struct with a sub column per cleansed column (according to the *cleaning_definition*). If a value was cleansed, the original value will be stored in its respective sub column. If a value was not cleansed, the sub column will be empty (None).

class BaseCleaner(*cleaning_definitions*, *column_to_log_cleansed_values*, *store_as_map=False*, *temporary_columns_prefix='1b75cdd2e2356a35486230c69cfac5493488a919'*)

Enumeration-based Cleaner

class EnumCleaner(*cleaning_definitions={}*, *column_to_log_cleansed_values=None*, *store_as_map=False*)

Cleanses a dataframe based on lists of allowed|disallowed values.

Examples

```
>>> from spooq.transformer import EnumCleaner
>>>
>>> transformer = EnumCleaner(
>>>     cleaning_definitions={
>>>         "status": {
>>>             "elements": ["active", "inactive"],
>>>         },
>>>         "version": {
>>>             "elements": ["", "None", "none", "null", "NULL"],
>>>             "mode": "disallow",
>>>             "default": None,
>>>         },
>>>     }
>>> )
```

```
>>> from spooq.transformer import EnumCleaner
>>> from pyspark.sql import Row
>>>
>>> input_df = spark.createDataFrame([
>>>     Row(a="stay", b="positive"),
>>>     Row(a="stay", b="negative"),
>>>     Row(a="stay", b="positive"),
>>> ])
>>> transformer = EnumCleaner(
>>>     cleaning_definitions={
>>>         "b": {
>>>             "elements": ["positive"],
>>>             "mode": "allow",
```

(continues on next page)

(continued from previous page)

```
>>>         }
>>>     },
>>>     column_to_log_cleansed_values="cleansed_values_enum",
>>>     store_as_map=True,
>>> )
>>> output_df = transformer.transform(input_df)
>>> output_df.show()
+----+--------+--------------------+
|   a|       b|cleansed_values_enum|
+----+--------+--------------------+
|stay|positive|                  []|
|stay|    null|     [b -> negative]|
|stay|positive|                  []|
+----+--------+--------------------+
>>> output_df.printSchema()
root
 |-- a: string (nullable = true)
 |-- b: string (nullable = true)
 |-- cleansed_values_enum: map (nullable = false)
 |    |-- key: string
 |    |-- value: string (valueContainsNull = true)
```

Parameters

- **cleaning_definitions** (dict) – Dictionary containing column names and respective cleansing rules
- **column_to_log_cleansed_values** (str, Defaults to None) – Defines a column in which the original (uncleansed) value will be stored in case of cleansing. If no column name is given, nothing will be logged.
- **store_as_map** (bool, Defaults to False) – Specifies if the logged cleansed values should be stored in a column as pyspark.sql.types.MapType with stringified values or as pyspark.sql.types.StructType with the original respective data types.

Note: Following cleansing rule attributes per column are supported:

- **elements, mandatory - list**
  A list of elements which will be used to allow or reject (based on mode) values from the input DataFrame.
- **mode, allow|disallow, defaults to 'allow' - str**
  "allow" will set all values which are NOT in the list (ignoring NULL) to the default value. "disallow" will set all values which ARE in the list (ignoring NULL) to the default value.
- **default, defaults to None - Column or any primitive Python value**
  If a value gets cleansed it gets replaced with the provided default value.

Returns
The transformed DataFrame

Return type
DataFrame

Raises

- **exceptions.ValueError** – Enumeration-based cleaning requires a non-empty list of elements per cleaning rule! Spooq did not find such a list for column: {column_name}

- **exceptions.ValueError** – Only the following modes are supported by EnumCleaner: 'allow' and 'disallow'.

Warning: None values are explicitly ignored as input values because *F.lit(None).isin(["elem1", "elem2"])* will neither return True nor False but None. If you want to replace Null values you should use the transformer `spooq.transformer.NullCleaner`

transform(*input_df*)

Performs a transformation on a DataFrame.

Parameters
input_df (DataFrame) – Input DataFrame

Returns
Transformed DataFrame.

Return type
DataFrame

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

Base Class

This abstract class provides the functionality to log any cleansed values into a separate column that contains a struct with a sub column per cleansed column (according to the *cleaning_definition*). If a value was cleansed, the original value will be stored in its respective sub column. If a value was not cleansed, the sub column will be empty (None).

class BaseCleaner(*cleaning_definitions*, *column_to_log_cleansed_values*, *store_as_map=False*, *temporary_columns_prefix='1b75cdd2e2356a35486230c69cfac5493488a919'*)

Null Cleaner

class NullCleaner(*cleaning_definitions=None*, *column_to_log_cleansed_values=None*, *store_as_map=False*)

Fills Null values of the specifield fields. Takes a dictionary with the fields to be cleaned and the default value to be set when the field is null.

Examples

```
>>> from pyspark.sql import functions as F
>>> from spooq.transformer import NullCleaner
>>> transformer = NullCleaner(
>>>     cleaning_definitions={
>>>         "points": {
>>>             "default": 0
>>>         }
>>>     }
>>> )
```

```
>>> from spooq.transformer import NullCleaner
>>> from pyspark.sql import Row
>>>
>>> input_df = spark.createDataFrame([
>>>     Row(id=0, points=5),
>>>     Row(id=1, points= None),
>>>     Row(id=2, points=15),
>>> ])
>>> transformer = NullCleaner(
>>>     cleaning_definitions={
>>>         "points":    {"default":  0},
>>>     },
>>>     column_to_log_cleansed_values="cleansed_values_null",
>>>     store_as_map=True,
>>> )
>>> output_df = transformer.transform(input_df)
>>> output_df.show()
+---+------+--------------------+
| id|points|cleansed_values_null|
+---+------+--------------------+
|  0|     5|                null|
|  1|     0|    [points -> null]|
|  2|    15|                null|
+---+------+--------------------+
>>> output_df.printSchema()
 |-- id: long (nullable = true)
 |-- points: long (nullable = true)
 |-- cleansed_values_null: map (nullable = true)
 |    |-- key: string
 |    |-- value: string (valueContainsNull = true)
```

Parameters

- **cleaning_definitions** (dict) – Dictionary containing column names and respective default values
- **column_to_log_cleansed_values** (str, Defaults to None) – Defines a column in which the original (uncleansed) value will be stored in case of cleansing. If no column name is given, nothing will be logged.
- **store_as_map** (bool, Defaults to False) – Specifies if the logged cleansed values should be stored in a column as pyspark.sql.types.MapType or as pyspark.sql.types.StructType with stringified values.

Note: The following cleaning_definitions attributes per column are mandatory:

- **default - Column or any primitive Python value**
  If a value gets cleansed it gets replaced with the provided default value.

Returns
The transformed DataFrame

Return type
DataFrame

Raises
exceptions.ValueError – Null-based cleaning requires the field default. Default parameter is not specified for column with name: {column_name}

transform(*input_df*)

Performs a transformation on a DataFrame.

Parameters
input_df (DataFrame) – Input DataFrame

Returns
Transformed DataFrame.

Return type
DataFrame

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

Base Class

This abstract class provides the functionality to log any cleansed values into a separate column that contains a struct with a sub column per cleansed column (according to the *cleaning_definition*). If a value was cleansed, the original value will be stored in its respective sub column. If a value was not cleansed, the sub column will be empty (None).

class BaseCleaner(*cleaning_definitions*, *column_to_log_cleansed_values*, *store_as_map=False*, *temporary_columns_prefix='1b75cdd2e2356a35486230c69cfac5493488a919'*)

Newest by Group (Most current record per ID)

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

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 (DataFrame) – Input DataFrame

Returns
Transformed DataFrame.

Return type
DataFrame

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

Create your own Transformer

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

2.3.3 Loaders

Loaders take a 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**.

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*)

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,
```

(continues on next page)

(continued from previous page)

```
>>>     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 (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

Create your own Loader

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

2.3.4 Pipeline

Pipeline

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

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'*)

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

- |*SPARK_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`

Class Diagram of Pipeline Subpackage

2.3.5 Spooq Base

Global Logger

Global Logger instance used by spooq.

Example

```
>>> import logging
>>> logga = logging.getLogger("spooq")
<logging.Logger at 0x7f5dc8eb2890>
>>> logga.info("Hello World")
[spooq] 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 "[spooq] 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

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**.

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: spooq/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!
```

(continues on next page)

(continued from previous page)

```
    Raises
    ------
    :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('spooq.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: spooq/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 Spooq'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 spooq.extractor import CSVExtractor

@pytest.fixture()
def default_extractor():
    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:: spooq.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!

Transformer Base Class

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

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

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: spooq/transformer/no_id_dropper.py:

```
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).
```

(continues on next page)

(continued from previous page)

```
    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 *spooq.transformer* instead of *spooq.transformer.no_id_dropper*. It will also be imported if you use *from spooq.transformer import **.

Listing 7: spooq/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
```

(continues on next page)

(continued from previous page)

```
+from no_id_dropper import NoIdDropper

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

Tests

One of Spooq'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 spooq.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"

class TestNoIdDropper(object):

    def test_records_are_dropped(transformed_df, input_df):
```

(continues on next page)

(continued from previous page)

```
        """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:: spooq.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!

Loader Base Class

Loaders take a `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**.

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: spooq/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.

    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`
```

(continues on next page)

(continued from previous page)

```
        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):
        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))
```

(continues on next page)

(continued from previous page)

```
        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 *spooq.loader* instead of *spooq.loader.parquet*. It will also be imported if you use *from spooq.loader import **.

Listing 12: spooq/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 Spooq'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 spooq.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(
        path=output_path,
        partition_by="attributes.gender",
        explicit_partition_values=None,
        compression_codec=None
    )
```

(continues on next page)

(continued from previous page)

```
@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:: spooq.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
```

(continues on next page)

(continued from previous page)

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

 Class Diagram of Loader Subpackage
```

That should be it!

2.4 Contribute

You can either fork Spooq and create a PR on github or get in contact with the authors to get access to the repo.

Spooq was built with extensibility in mind which results in clearly separated and independent modules and classes.

2.4.1 Prerequisites

- python 3.8
- Java 8+ (jdk8-openjdk)
- pipenv
- Latex (for PDF documentation)

2.4.2 Setting up the Environment

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

Run the following command to install the packages needed for development and testing:

```
$ 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`.

2.4.3 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` or got to their documentation

2.4.4 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*

2.4.5 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=spooq
```

Listing 21: Get the test coverage report as HTML

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

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
```

2.4.6 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/spooq.pdf` in your application of choice.

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/Spooq.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, Python3 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. for an example.

2.4.7 Release a new Version on PyPi

Things to consider

Version Bump

For any update on PyPi we need a new version number. You can manually edit the file *spooq/_version.py* to change the version number. This is reflected in the *setup.py* and consequently in the release version number.

Documentation

Please don't forget to also update the documentation accordingly. This is either done directly in the source code as docstrings or for more overview-centered topics in the rst file under *docs/source*.

Changelog

Please add your changes to the CHANGELOG.rst

Automatic Publishing via Github Action

The current Spooq version is automatically published on PyPi after a release on github is created.

Manual Publishing from Command Line

Create the Distribution Files

```
$ python setup.py sdist bdist_wheel
```

Upload to Test-PyPi

```
$ pipenv shell
$ twine upload --repository-url https://test.pypi.org/legacy/ dist/
```

Your new version is available at https://test.pypi.org/project/Spooq/. Beware, that the test PyPi uses different credentials than the real PyPi. You can get the credentials from your favourite collaborator.

Upload to Real PyPi

```
$ pipenv shell
$ twine upload dist/
```

Your new version is available at https://pypi.org/project/Spooq/. You can get the credentials from your favourite collaborator.

2.5 Changelog

2.5.1 3.4.2 (2024-08-08)

- [ADD] Annotator: New transformer to load and apply comments to dataframes
- [MOD] Mapper: Change mode and missing_column_handling from strings to Enums
- [MOD] Mapper: Add support for column comments (via annotator)

2.5.2 3.4.1 (2024-06-05)

- [MOD] Mapper: Add validation mode

2.5.3 3.4.0 (2024-03-15)

- [MOD] Mapper: Custom transformations can now also be used with `select`, `withColumn` or `where` clauses
- [MOD] Mapper: Custom transformations can now be passed as python objects with or without parameters
- [MOD] Mapper: Spark's built-in data types can now be passed as simple strings (f.e. "string")
- [MOD] Mapper: Renaming (shortening) of most custom Mapper transformations (https://spooq.rtfd.io/en/latest/transformer/mapper_transformations.html)
- [ADD] Mapper: `str_to_array` transformation
- [ADD] Mapper: `map_values` transformation
- [ADD] Mapper: `apply` transformation
- [MOD] Tests use now Python 3.8
- [MOD] Spark 3.3.0 compatibility (Tested for all Spark version from 3.0 to 3.3)

- [MOD] Clean up documentation
- [FIX] Tests with github actions

2.5.4 3.3.9 (2022-08-16)

- [MOD] Mapper: Replace missing column parameters (*nullify_missing_columns*, *skip_missing_columns*, *ignore_missing_columns*) with one single parameter: *missing_column_handling*.

2.5.5 3.3.8 (2022-08-11)

- [MOD] Mapper: Add additional parameter allowing skipping of transformations in case the source column is missing:
  - *nullify_missing_columns*: set source column to null in case it does not exist
  - *skip_missing_columns*: skip transformation in case the source column does not exist
  - *ignore_missing_columns*: DEPRECATED -> use *nullify_missing_columns* instead

2.5.6 3.3.7 (2022-03-15)

- [FIX] Fix long overflow in extended_string_to_timestamp

2.5.7 3.3.6 (2021-11-19)

- [FIX] Fix Cleaners logs in case of field type different than string

2.5.8 3.3.5 (2021-11-16)

- [ADD] Add Null Cleaner spooq.transformer.NullCleaner

2.5.9 3.3.4 (2021-07-21)

- [MOD] Store null value instead of an empty Map in case no cleansing was necessary

2.5.10 3.3.3 (2021-06-30)

- [MOD] Change logic for storing cleansed values as MapType Column to not break Spark (logical plan got to big)
- [MOD] Add streaming tests (parquet & delta) for EnumCleaner unit tests.

2.5.11 3.3.2

- Left out intentionally as there is already a yanked version 3.3.2 on PyPi

2.5.12 3.3.1 (2021-06-22)

- [MOD] Add option to store logged cleansed values as MapType (Enum & Threshold based cleansers)
- [FIX] Fix TOC on PyPi, add more links to PyPi

2.5.13 3.3.0 (2021-04-22)

- [MOD] (BREAKING CHANGE!) rename package back to Spooq (dropping "2"). This means you need to update all imports from spooq2.xxx.yyy to spooq.xxx.yyy in your code!
- [MOD] Prepare for PyPi release
- [MOD] Drop official support for Spark 2.x (it most probably still works without issues, but some tests fail on Spark2 due to different columns ordering and the effort is too high to maintain both versions with respect to tests)

2.5.14 3.2.0 (2021-04-13)

- [MOD] Add functionality to log cleansed values into separate struct column (column_to_log_cleansed_values)
- [MOD] Add ignore_ambiguous_columns to Mapper
- [MOD] Log spooq version when importing
- [REM] Drop separate spark package (bin-folder) as pip package can now handle all tests as well
- [ADD] Github action to test on label (test-it) or merge into master

2.5.15 3.1.0 (2021-01-27)

- [ADD] EnumCleaner Transformer
- [MOD] Add support for dynamic default values with the ThresholdCleaner

2.5.16 3.0.1 (2021-01-22)

- [MOD] extended_string_to_timestamp: now keeps milli seconds (no more cast to LongType) for conversion to Timestamp

2.5.17 3.0.0b (2020-12-09)

- [ADD] Spark 3 support (different handling in tests via *only_sparkX* decorators)
- [FIX] Fix null types in schema for custom transformations on missing columns
- [MOD] (BREAKING CHANGE!) set default for *ignore_missing_columns* of Mapper to False (fails on missing input columns)

2.5.18 2.3.0 (2020-11-23)

- [MOD] extended_string_to_timestamp: it can now handle unix timestamps in seconds and in milliseconds
- [MOD] extended_string_to_date: it can now handle unix timestamps in seconds and in milliseconds

2.5.19 2.2.0 (2020-10-02)

- [MOD] Add support for prepending and appending mappings on input dataframe (Mapper)
- [MOD] Add support for custom spark sql functions in mapper without injecting methods
- [MOD] Add support for "on"/"off" and "enabled"/"disabled" in extended_string_to_boolean custom mapper transformations
- [ADD] New custom mapper transformations:
  - extended_string_to_date
  - extended_string_unix_timestamp_ms_to_date
  - has_value

2.5.20 2.1.1 (2020-09-04)

- [MOD] *drop_rows_with_empty_array* flag to allow keeping rows with empty array after explosion
- [MOD] Additional test-cases for extended_string mappings (non string inputs)
- [FIX] Remove STDERR logging, don't touch root logging level anymore (needs to be done outside spooq to see some lower log levels)
- [ADD] New custom mapper transformations:
  - extended_string_unix_timestamp_ms_to_timestamp

2.5.21 2.1.0 (2020-08-17)

- [ADD] Python 3 support
- [MOD] *ignore_missing_columns* flag to fail on missing input columns with Mapper transformer (https://github.com/Breaka84/Spooq/pull/6)
- [MOD] Timestamp support for threshold cleaner
- [ADD] New custom mapper transformations:
  - meters_to_cm
  - unix_timestamp_ms_to_spark_timestamp
  - extended_string_to_int
  - extended_string_to_long
  - extended_string_to_float
  - extended_string_to_double
  - extended_string_to_boolean
  - extended_string_to_timestamp

2.5.22 2.0.0 (2020-05-22)

- [UPDATE] Upgrade to use Spark 2 (tested for 2.4.3) -> will no longer work for spark 1
- Breaking changes (severe refactoring)

2.5.23 0.6.2 (2019-05-13)

- [FIX] Logger writes now to std_out and std_err & logger instance is shared across all spooq instances
- [FIX] PyTest version locked to 3.10.1 as 4+ broke the tests
- [MOD] Removes id_function to create names for parameters in test methods (fallback to built-in)
- [ADD] Change SelectNewestByGroup from string eval to pyspark objects
- [FIX] json_string is now able to None values

2.5.24 0.6.1 (2019-03-26)

- [FIX] PassThrough Extractor (input df now defined at instantiation time)
- [ADD] json_string new custom data type

CHAPTER

THREE

INDICES AND TABLES

- modindex
- search

PYTHON MODULE INDEX

S

INDEX

Symbols

A

C

D

E

G

H

I

J

L

M

N

P

S

T

U