
Butterfree Documentation

Release 1.0.2

QuintoAndar

Jun 15, 2021

CONTENTS

1	Navigation	3
1.1	Welcome to the Butterfree Docs!	3
1.2	Getting Started	5
1.3	Source	6
1.4	Feature Sets	7
1.5	Aggregated Feature Sets	9
1.6	Sink	10
1.7	Streaming Feature Sets in Butterfree	11
1.8	Setup Configuration	12
1.9	API Specification	16
1.10	Command-line Interface (CLI)	131
	Python Module Index	133
	Index	135

Made with by the **MLOps** team from [QuintoAndar](#).

The library is centered on the following concepts:

- **ETL**: central framework to create data pipelines. Spark-based Extract, Transform and Load modules ready to use.
- **Declarative Feature Engineering**: care about what you want to compute and not how to code it.
- **Feature Store Modeling**: the library easily provides everything you need to process and load data to your Feature Store.

1.1 Welcome to the Butterfree Docs!

The main idea is for this repository to be a set of tools for easing [ETLs](#). The idea is using Butterfree to upload data to a Feature Store, so data can be provided to your machine learning algorithms.

1.1.1 Table of Contents

- *What is going on here?*
- *Service Architecture*
- *Extract*
- *Transform*
- *Load*
- *Streaming*
- *Setup Configuration*
- *Command-line Interface*

1.1.2 What is going on here

Besides introducing Butterfree itself, it's necessary to define some concepts that will be presented throughout this Docs.

The feature store is where features for machine learning models and pipelines are stored. A feature is an individual property or characteristic of a data-sample, such as the height of a person, the area of a house or an aggregated feature as the average prices of houses seen by a user within the last day. A feature set can be thought of as a set of features. Finally, an entity is a unity representation of a specific business context.

This repository holds all scripts that will extract data from necessary sources (S3 and Kafka, for instance), transform all of this raw data into feature sets and, finally, upload these results in a feature store.

Scripts use Python and [Apache's Spark](#).

It's important to highlight that a feature store incurs some challenging concerns, such as:

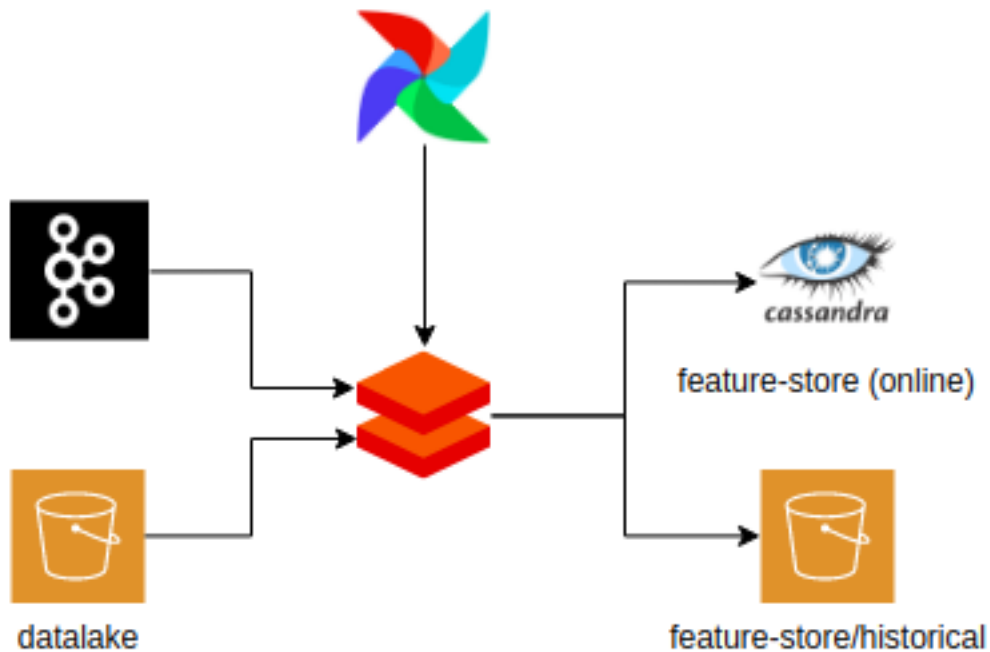
- Dealing with big volumes of data;
- Retroactively computing new features;
- Loading historical and online data for consumption;

- Keeping it simple to add new features and functionality;
- Making it easy to share code.

1.1.3 Service Architecture

Regarding this architecture, some requisites can be emphasized, for instance:

- We currently support [Amazon S3](#) and [Apache Kafka](#) as data sources;
- A hive metastore is required within your Spark session;
- An [Apache Cassandra](#) database should be up and running.



Some important core concepts regarding this service are features, feature sets and entities, discussed previously. Besides, it's possible to draw attention to other two major ideas:

- Historical Feature Store: all features are calculated over time and stored at S3;
- Online Feature Store: hot/latest data stored at a low latency data storage, such as Apache Cassandra.

1.1.4 Extract

Basically, the extract step is performed with a `Source` object, by defining the desired data sources. You can learn more about it in the [Extract Session](#).

1.1.5 Transform

The transform step is defined within a `FeatureSet`, by explicitly defining the desired transformations. More information about the transformations can be found at the [Transform Session](#).

1.1.6 Load

It's the last step of the ETL process, and it's defined by a `Sink` object. Please, refer to the [Sink Session](#) to know more.

1.1.7 Streaming

We also support streaming pipelines in Butterfree. More information is available at the [Streaming Session](#).

1.1.8 Setup Configuration

Some configurations are needed to run your ETL pipelines. Detailed information is provided at the [Configuration Section](#)

1.1.9 Command-line Interface

Butterfree has its own command-line interface, to manage your feature sets. Detailed information provided by the [Command-line Interface](#) section.

1.2 Getting Started

Butterfree depends on **Python 3.7+** and it is **Spark 3.0 ready**.

[Python Package Index](#) hosts reference to a pip-installable module of this library, using it is as straightforward as including it on your project's requirements.

```
pip install butterfree
```

Or after listing `butterfree` in your `requirements.txt` file:

```
pip install -r requirements.txt
```

1.2.1 Discovering Butterfree

Welcome to **Discovering Butterfree** tutorial series! Click on the following links to open the tutorials:

[#1 Feature Set Basics](#)

[#2 Spark Functions and Window](#)

[#3 Aggregated Feature Set](#)

[#4 Streaming Feature Set](#)

1.3 Source

Regarding the extract step, we can define a `Source` as a set of data sources in order to join your raw data for the transform step.

Currently, we support three different data sources or, as it's called within Butterfree, `readers`:

- `FileReader`: this reader loads data from a file, as the name suggests, and returns a dataframe. It can be instantiated as:

```
file_reader = FileReader(  
    id="file_reader_id",  
    path="data_path",  
    format="json"  
)
```

- `TableReader`: this reader loads data from a table registered in spark metastore and returns a dataframe. It can be instantiated as:

```
table_reader = TableReader(  
    id="table_reader_id",  
    database="table_reader_db",  
    table="table_reader_table"  
)
```

- `KafkaReader`: this reader loads data from a kafka topic and returns a dataframe. It can be instantiated as:

```
kafka_reader = KafkaReader(  
    id="kafka_reader_id",  
    topic="topic",  
    value_schema=value_schema  
    connection_string="host1:port,host2:port",  
)
```

After defining all your data sources, it's important to write a query in order to define the relation between them, something like this:

```
source = Source(  
    readers=[  
        TableReader(  
            id="table_reader_id",  
            database="table_reader_db",  
            table="table_reader_table",  
        ),  
        FileReader(id="file_reader_id", path="data_sample_path", format="json"),  
    ],  
    query=f"select a.*, b.feature2 "  
    f"from table_reader_id a "  
    f"inner join file_reader_id b on a.id = b.id ",  
)
```

It's important to state that we have some pre-processing methods as well, such as `filter` and `pivot`. Feel free to check them [here](#).

1.4 Feature Sets

Regarding the transform step, we can define a `FeatureSet` as a set of features in order to apply the suggested transformations to your data.

It's possible to use three different kinds of features:

- `KeyFeature`: a `FeatureSet` must contain one or more `KeyFeatures`, which will be used as keys when storing the feature set dataframe as tables. The `FeatureSet` may validate keys that are unique for the latest state of a feature set.
- `TimestampFeature`: a `FeatureSet` must contain one `TimestampFeature`, which will be used as a time tag for the state of all features. By containing a timestamp feature, users may time travel over their features. The `FeatureSet` may validate that the set of keys and timestamp are unique for a feature set. By defining a timestamp column, the feature set will always contain a data column called "timestamp" of `TimestampType` (spark dtype).
- `Feature`: a `Feature` is the result of a transformation over one (or more) data columns over an input dataframe. Transformations can be as simple as renaming, casting types, mathematical expressions or complex functions/models. It can be instantiated as:

```
feature = Feature(
    name="new_feature",
    description="simple feature renaming",
    from_column="feature"
)
```

It's possible to define the desired transformation to be applied over your input data by defining the `transformation` parameter within the `Feature` instantiation. The following transform components can be used in Butterfree:

- `SparkFunctionTransform`: this component can be used when a `pyspark sql function` needs to be used. It can be instantiated as:

```
feature = Feature(
    name="feature",
    description="Spark function transform usage example",
    transformation=SparkFunctionTransform(
        functions=[Function(functions.cos, DataType.DOUBLE)])
)
```

Feel free to check more [here](#).

- `SQLExpressionTransform`: as the name suggests, this component can be used when a user wants to define the feature's transformation using SQL. It can be instantiated as:

```
feature = Feature(
    name="feature",
    description="SQL expression transform usage example",
    transformation=SQLExpressionTransform(expression="feature1/feature2"),
)
```

You can find more info [here](#).

- `CustomTransform`: this component can be used when an user wants to define a custom transformation by defining a method. For instance:

```
def divide(df, parent_feature, column1, column2):
    name = parent_feature.get_output_columns()[0]
```

(continues on next page)

(continued from previous page)

```

df = df.withColumn(name, F.col(column1) / F.col(column2))
return df

feature = Feature(
  name="feature",
  description="custom transform usage example",
  transformation=CustomTransform(
    transformer=divide, column1="feature1", column2="feature2",
  )
)

```

You can take a look at the code [here](#).

- H3HashTransform: this component can be used when an user wants to convert latitude and longitude values into a hash. You can learn more about H3 [here](#). For instance:

```

feature = Feature(
  name="feature",
  description="h3 hash transform usage example",
  transformation=H3HashTransform(
    h3_resolutions=[6, 7, 8, 9, 10, 11, 12],
    lat_column="lat",
    lng_column="lng",
  )
)

```

You can read the code [here](#).

- StackTransform: this component can be used when an user wants to create a stack column based on some given values. For instance:

```

feature = Feature(
  name="stack_ids",
  description="id_a and id_b stacked in a single column.",
  transformation=StackTransform("id_a", "id_b"),
)

```

Feel free to check it out [here](#).

Finally, an example of a FeatureSet is provided:

```

feature_set = FeatureSet(
  name="feature_set",
  entity="entity",
  description="description",
  features=[
    Feature(
      name="feature1",
      description="test",
      transformation=SparkFunctionTransform(
        functions=[
          Function(F.avg, DataType.DOUBLE),
          Function(F.stddev_pop, DataType.DOUBLE)]
      ).with_window(
        partition_by="id",
        order_by=TIMESTAMP_COLUMN,
        mode="fixed_windows",
      )
    )
  ]
)

```

(continues on next page)

(continued from previous page)

```

        window_definition=["2 minutes", "15 minutes"],
    ),
),
Feature(
    name="divided_feature",
    description="unit test",
    transformation=CustomTransform(
        transformer=divide, column1="feature1", column2="feature2",
    ),
),
],
keys=[KeyFeature(name="id", description="The user's Main ID or device ID")],
timestamp=TimestampFeature(),
)

```

1.5 Aggregated Feature Sets

When an user desires to use aggregated features, an `AggregatedFeatureSet` can be used. Besides `KeyFeature`, `TimestampFeature` and `Feature` classes, we're going to use a new component regarding the transformations, called `AggregatedTransform`.

- `AggregatedTransform`: this transformation needs to be used within an `AggregatedFeatureSet`. Unlike the other transformations, this class won't have a `transform` method implemented. The idea behind aggregating is that, in spark, we should execute all aggregation functions after in a single group by. So an `AggregatedFeatureSet` will have many features with an associated `AggregatedTransform`. If each one of them needs to apply a `groupby.agg()`, then we must join all the results in the end, making this computation extremely slow. It's important to say that the `AggregatedFeatureSet` can be used with both time and row windows.

Some examples are provided below:

```

feature_set = AggregatedFeatureSet(
    name="aggregated_feature_set",
    entity="entity",
    description="description",
    features=[
        Feature(
            name="feature1",
            description="test",
            transformation=AggregatedTransform(
                functions=[
                    Function(F.avg, DataType.DOUBLE),
                    Function(F.stddev_pop, DataType.DOUBLE)],
            ),
            from_column="rent",
        ),
    ],
    keys=[KeyFeature(name="id", description="some key")],
    timestamp=TimestampFeature(from_column="ts"),
)

```

You could also add a time window, like this:

```
feature_set.with_windows(definitions=["3 days"])
```

Also, you can add a `pivot` operation:

```
feature_set.with_pivot(  
    column="status",  
    values=["publicado", "despublicado", "alugado"]  
) .with_windows(definitions=["1 day", "2 weeks"])
```

Please refer to [this](#) code in order to understand more. Also, check our [examples](#) section.

1.6 Sink

The Load step is the `Sink` method, where we define the destinations for the feature set pipeline, that is, it is the process of recording the transformed data after the transformation step.

Declaring the sink:

```
sink = Sink(  
    writers = [HistoricalFeatureStoreWriter(), OnlineFeatureStoreWriter()]  
)
```

Currently, you can write your data into two types of writers:

- `HistoricalFeatureStoreWriter`: The Historical Feature Store will write the data to an AWS S3 bucket.
- `OnlineFeatureStoreWriter`: The Online Feature Store will write the data to a Cassandra database.

If you declare your writers without a database configuration, they will use their default settings. But we can also define this configuration, such as:

- `HistoricalFeatureStoreWriter`:

```
config = S3Config(bucket="my_bucket", mode="append", format_="parquet")  
writers = [HistoricalFeatureStoreWriter(db_config=config)]
```

- `OnlineFeatureStoreWriter`:

```
config = CassandraConfig(  
    mode="overwrite",  
    format_="org.apache.spark.sql.cassandra",  
    keyspace="keyspace_name"  
)  
writers = [OnlineFeatureStoreWriter(db_config=config)]
```

You can see the writers [here](#) and database configuration [here](#).

It's also important to highlight that our writers support a `debug_mode` option:

```
writers = [HistoricalFeatureStoreWriter(debug_mode=True),  
↳ OnlineFeatureStoreWriter(debug_mode=True)]  
sink = Sink(writers=writers)
```

When `debug_mode` assumes `True`, then a temporary view will be created, therefore no data will be actually saved to both historical and online feature store. Feel free to check our [examples](#) section, in order to learn more about how to use this mode.

1.7 Streaming Feature Sets in Butterfree

1.7.1 Introduction

Spark enables us to deal with streaming processing in a very powerful way. For an introduction of all Spark's capabilities in streaming you can read more in this [link](#). As the core Spark, Butterfree also let you declare pipelines to deal with streaming data. The best is that the pipeline declaration is almost the same as dealing with batch use-cases, so there isn't too complex to deal with this type of challenge using Butterfree tools.

Streaming feature sets are the ones that have at least one streaming source of data declared in the `Readers` of a `FeatureSetPipeline`. The pipeline is considered a streaming job if it has at least one reader in streaming mode (`stream=True`).

1.7.2 Readers

Using readers in streaming mode will make use of Spark's `readStream` API instead of the normal `read`. That means it will produce a stream dataframe (`df.isStreaming == True`) instead of a normal Spark's dataframe.

The currently supported readers in stream mode are `FileReader` and `KafkaReader`. For more information about the specifications read their docstrings, [here](#) and [here](#) respectively.

1.7.3 Online Feature Store Writer

`OnlineFeatureStoreWriter` is currently the only writer that supports streaming dataframes. It will write, in real time, and upserts to Cassandra. It uses `df.writeStream` and `foreachBatch` Spark functionality to do that. You can read more about `foreachBatch` [here](#).



Debug Mode

You can use the `OnlineFeatureStoreWriter` in debug mode (`debug_mode=True`) with streaming dataframes. So instead of trying to write to Cassandra, the data will be written to an in-memory table. So you can query this table to show the output as it is being calculated. Normally this functionality is used for the purpose of testing if the defined features have the expected results in real time.

1.7.4 Pipeline Run

Differently from a batch run, a pipeline running with a streaming dataframe will not “finish to run”. The pipeline will continue to get data from the streaming, process the data and save it to the defined sink sources. So when managing a job using this feature, an operation needs to be designed to support a continuously-up streaming job.

1.8 Setup Configuration

To perform a ETL pipeline, some configurations are needed. For example: credentials to connect to data sources and default paths. This chapter will cover the main setup configurations on Butterfree.

Some of the configurations are possible to set trough environment variables, the API or both. The priority is first trying to use the parameters used in the instantiation, with the environment variables as fallback. Check the following examples:

1.8.1 Kafka Reader

Configurations:

- Kafka Consumer Connection String: connection string to use to connect to the Kafka.

Setup by instantiation:

```
from butterfree.extract.readers import KafkaReader

kafka_reader = KafkaReader(
    id="reader_id",
    topic="topic_name",
    value_schema=schema,
    connection_string="host1:port,host2:port"
)
```

Setup by environment variables:

Setup the following environment variable in the Spark Cluster/Locally: KAFKA_CONSUMER_CONNECTION_STRING

After setting this variables you can instantiate KafkaReader easier:

```
from butterfree.extract.readers import KafkaReader

kafka_reader = KafkaReader(
    id="reader_id",
    topic="topic_name",
    value_schema=schema
)
```


1.8.2 Online Feature Store

Cassandra Configuration

Configurations:

- Cassandra Username: username to connect to CassandraDB.
- Cassandra Password: password to connect to CassandraDB.
- Cassandra Host: host to connect to CassandraDB.
- Cassandra Keyspace: keyspace to connect to CassandraDB.
- Stream Checkpoint Path: path on S3 to save the checkpoint for the streaming query. Only need when performing streaming writes.

Setup by instantiation:

```
from butterfree.configs.db import CassandraConfig
from butterfree.load.writers import OnlineFeatureStoreWriter

db_config = CassandraConfig(
    username="username",
    password="password",
    host="host",
    keyspace="keyspace",
    stream_checkpoint_path="path"
)
writer = OnlineFeatureStoreWriter(db_config=db_config)
```

Setup by environment variables:

Setup the following environment variables in the Spark Cluster/Locally: CASSANDRA_USERNAME, CASSANDRA_PASSWORD, CASSANDRA_HOST, CASSANDRA_KEYSPACE, STREAM_CHECKPOINT_PATH

After setting this variables you can instantiate `CassandraConfig` and `OnlineFeatureStoreWriter` easier:

```
from butterfree.configs.db import CassandraConfig
from butterfree.load.writers import OnlineFeatureStoreWriter

db_config = CassandraConfig()
writer = OnlineFeatureStoreWriter(db_config=db_config)

# or just
writer = OnlineFeatureStoreWriter()
```

Kafka Configuration

Configurations:

- Kafka topic: Kafka topic name.
- Kafka connection string: string with hosts and ports to connect.
- Stream Checkpoint Path: path on S3 to save the checkpoint for the streaming query. Only need when performing streaming writes.

Setup by instantiation:

```
from butterfree.configs.db import KafkaConfig
from butterfree.load.writers import OnlineFeatureStoreWriter

custom_kafka_config = KafkaConfig(
    kafka_topic="custom_topic",
    kafka_connection_string="kafka_connection_string",
    stream_checkpoint_path="path"
)
writer = OnlineFeatureStoreWriter(db_config=custom_kafka_config)
```

or

```
from butterfree.configs.db import KafkaConfig
from butterfree.load.writers import OnlineFeatureStoreWriter

kafka_config = KafkaConfig(
    kafka_connection_string="kafka_connection_string",
    stream_checkpoint_path="path"
)
writer = OnlineFeatureStoreWriter(db_config=kafka_config)
```

Setup by environment variables:

Setup the following environment variables in the Spark Cluster/Locally:
KAFKA_CONSUMER_CONNECTION_STRING, STREAM_CHECKPOINT_PATH

After setting this variables you can instantiate `KafkaConfig` and `OnlineFeatureStoreWriter` easier:

```
from butterfree.configs.db import KafkaConfig
from butterfree.load.writers import OnlineFeatureStoreWriter

kafka_config = KafkaConfig()
writer = OnlineFeatureStoreWriter(db_config=kafka_config)
```

1.8.3 Historical Feature Store (Spark Metastore and S3)

Configurations:

- Feature Store S3 Bucket: Bucket on S3 to use for the Historical Feature Store.
- Feature Store Historical Database: Database on Spark metastore to use to write the tables from Historical Feature Store.

Setup by instantiation:

```
from butterfree.configs.db import S3Config
from butterfree.load.writers import HistoricalFeatureStoreWriter

db_config = S3Config(bucket="bucket")
writer = HistoricalFeatureStoreWriter(
    db_config=db_config,
    database="database"
)
```

Setup by environment variables:

Setup the following environment variables in the Spark Cluster/Locally: FEATURE_STORE_S3_BUCKET, FEATURE_STORE_HISTORICAL_DATABASE

After setting this variables you can instantiate S3Config and HistoricalFeatureStoreWriter easier:

```
from butterfree.configs.db import S3Config
from butterfree.load.writers import HistoricalFeatureStoreWriter

db_config = S3Config()
writer = HistoricalFeatureStoreWriter(db_config=db_config)

# or just
writer = HistoricalFeatureStoreWriter()
```

1.8.4 Important Considerations

API information

For more detailed information on the API arguments is better to check their own docstrings.

Troubleshoot checks

- Check if Spark cluster machines have all the permissions to read/write/modify on the Historical FeatureStore Bucket
- Check if Spark cluster machines can reach the all the configured hosts
- Check if the configured credentials have all the needed permissions on CassandraDB

1.9 API Specification

1.9.1 butterfree package

Subpackages

butterfree.clients package

Submodules

Abstract class for database clients.

class butterfree.clients.abstract_client.**AbstractClient**

Bases: abc.ABC

Abstract base class for database clients.

abstract property conn

Returns a connection object.

abstract get_schema (*table: str, database: str = None*) → Any

Returns desired table schema.

table

desired table.

Returns A list of dictionaries in the format [{"column_name": "example1", type: "Spark_type"}, ...]

abstract sql (*query: str*) → Any

Runs a query.

Parameters query – client query.

Returns Set of records.

CassandraClient entity.

class butterfree.clients.cassandra_client.**CassandraClient** (*host: List[str],
keyspace: str, user: Optional[str] =
None, password: Optional[str] = None*)

Bases: *butterfree.clients.abstract_client.AbstractClient*

Cassandra Client.

user

username to use in connection.

password

password to use in connection.

keyspace

key space used in connection.

host

cassandra endpoint used in connection.

property conn

Establishes a Cassandra connection.

create_table (*columns*: List[butterfree.clients.cassandra_client.CassandraColumn], *table*: str) →

None
Creates a table.

columns

a list dictionaries in the format [{"column_name": "example1", type: "cql_type", primary_key: True}, ...]

get_schema (*table*: str, *database*: str = None) → List[Dict[str, str]]

Returns desired table schema.

table

desired table.

Returns A list dictionaries in the format [{"column_name": "example1", type: "cql_type"}, ...]

sql (*query*: str) → cassandra.cluster.ResponseFuture

Executes desired query.

query

desired query.

class butterfree.clients.cassandra_client.CassandraColumn (**kwargs)

Bases: dict

Type for cassandra columns.

It's just a type abstraction, we can use it or a normal dict

```
>>> def function(column: CassandraColumn) -> CassandraColumn:
...     return column
>>> # The following two lines will pass in the type checking
>>> function({'column_name': 'test', 'type': 'integer', 'primary_key': False})
>>> function(CassandraColumn(column_name='test', type='integer', primary_
↪key=False))
```

column_name: str

primary_key: bool

type: str

SparkClient entity.

class butterfree.clients.spark_client.SparkClient

Bases: *butterfree.clients.abstract_client.AbstractClient*

Handle Spark session connection.

Get query results with SQL, reads and writes data on external systems.

add_table_partitions (*partitions*: List[Dict[str, Any]], *table*: str, *database*: str = None) → None

Add partitions to an existing table.

Parameters

- **partitions** – partitions to add to the table. It's expected a list of partition dicts to add to the table. Example: [{"year": 2020, "month": 8, "day": 14}, ...]
- **table** – table to add the partitions.

- **database** – name of the database where the table is saved.

property conn

Gets or creates an SparkSession.

Returns Spark session

static create_temporary_view (*dataframe: pyspark.sql.dataframe.DataFrame, name: str*) → Any

Create a temporary view from a given dataframe.

Parameters

- **dataframe** – dataframe to be queried by the view.
- **name** – name of the temporary view.

get_schema (*table: str, database: str = None*) → List[Dict[str, str]]

Returns desired table schema.

table

desired table.

Returns A list of dictionaries in the format [{"column_name": "example1", type: "Spark_type"}, ...]

read (*format: str, path: Optional[Union[str, List[str]]] = None, schema: Optional[pyspark.sql.types.StructType] = None, stream: bool = False, **options: Any*) → pyspark.sql.dataframe.DataFrame

Use the SparkSession.read interface to load data into a dataframe.

Check docs for more information: <https://spark.apache.org/docs/latest/sql-data-sources-load-save-functions.html#generic-loadsave-functions>

Parameters

- **format** – string with the format to be used by the DataframeReader.
- **path** – optional string or a list of string for file-system.
- **stream** – flag to indicate if data must be read in stream mode.
- **schema** – an optional pyspark.sql.types.StructType for the input schema.
- **options** – options to setup the DataframeReader.

Returns Dataframe

read_table (*table: str, database: str = None*) → pyspark.sql.dataframe.DataFrame

Use the SparkSession.read interface to read a metastore table.

Parameters

- **database** – name of the metastore database/schema
- **table** – name of the table in metastore

Returns Dataframe

sql (*query: str*) → pyspark.sql.dataframe.DataFrame

Run a query using Spark SQL.

Parameters **query** – Spark SQL query.

Returns Dataframe

static write_dataframe (*dataframe: pyspark.sql.dataframe.DataFrame, format_: str, mode: str, **options: Any*) → None

Receive a spark DataFrame and write it.

Parameters

- **dataframe** – dataframe containing data from a feature set.
- **format** – format used to save the dataframe.
- **mode** – writing modem can be “error”, “append”, “overwrite” or “ignore”. For more information: [here](https://spark.apache.org/docs/2.3.0/sql-programming-guide.html#save-modes).
- ****options** – all other options that can be used in a DataFrameWriter.

write_stream (*dataframe: pyspark.sql.dataframe.DataFrame, processing_time: str, output_mode: str, checkpoint_path: Optional[str], format_: str, mode: str, **options: Any*) → pyspark.sql.streaming.StreamingQuery

Starts streaming data writing job.

Parameters

- **dataframe** – Spark dataframe containing data from a feature set.
- **processing_time** – a processing time interval as a string. E.g. ‘5 seconds’, ‘1 minute’. Set a trigger that runs the mini-batch periodically based on the processing time. If the effect of processing data as soon as the data arrives, without having to wait for the time frame, is desired, the value ‘0 seconds’ can be set.
- **output_mode** – specifies how data of a streaming DataFrame/Dataset is written to a streaming sink destination.
- **checkpoint_path** – path on S3 to save checkpoints for the stream job. These checkpoint can be used on the the job re-start to return from where it stops.
- **format** – format used to save the dataframe.
- **mode** – writing modem can be “error”, “append”, “overwrite” or “ignore”. For more information: [here](https://spark.apache.org/docs/2.3.0/sql-programming-guide.html#save-modes).
- ****options** – all other options that can be used in a DataFrameWriter.

More information about processing_time, output_mode and checkpoint_path can be found in Spark documentation: [here](https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html)

Returns Streaming handler.

static write_table (*dataframe: pyspark.sql.dataframe.DataFrame, database: Optional[str], table_name: str, path: str, format_: str = None, mode: str = None, partition_by: List[str] = None, **options: Any*) → None

Receive a spark DataFrame and write it as a table in metastore.

Parameters

- **dataframe** – spark dataframe containing data from a feature set.
- **database** – specified database name.
- **table_name** – specified table name.
- **path** – string with the local to save the table.
- **format** – string with the format used to save.

- **mode** – writing mode, it can be: “error”, “append”, “overwrite” or “ignore”. More information: [here](https://spark.apache.org/docs/2.3.0/sql-programming-guide.html#save-modes).
- **partition_by** – names of partitioning columns.
- **options** – all other options that can be used in a DataFrameWriter.

Module contents

Holds connection clients.

class `butterfree.clients.AbstractClient`

Bases: `abc.ABC`

Abstract base class for database clients.

abstract property `conn`

Returns a connection object.

abstract `get_schema` (*table: str, database: str = None*) → Any

Returns desired table schema.

table

desired table.

Returns A list of dictionaries in the format [{"column_name": "example1", type: "Spark_type"}, ...]

abstract `sql` (*query: str*) → Any

Runs a query.

Parameters `query` – client query.

Returns Set of records.

class `butterfree.clients.CassandraClient` (*host: List[str], keyspace: str, user: Optional[str] = None, password: Optional[str] = None*)

Bases: `butterfree.clients.abstract_client.AbstractClient`

Cassandra Client.

user

username to use in connection.

password

password to use in connection.

keyspace

key space used in connection.

host

cassandra endpoint used in connection.

property `conn`

Establishes a Cassandra connection.

create_table (*columns: List[butterfree.clients.cassandra_client.CassandraColumn], table: str*) →

None
Creates a table.

columns

a list dictionaries in the format [{"column_name": "example1", type: "cql_type", primary_key: True}, ...]

get_schema (*table: str, database: str = None*) → List[Dict[str, str]]

Returns desired table schema.

table

desired table.

Returns A list dictionaries in the format [{"column_name": "example1", type: "cql_type"}, ...]

sql (*query: str*) → cassandra.cluster.ResponseFuture

Executes desired query.

query

desired query.

class butterfree.clients.**SparkClient**

Bases: *butterfree.clients.abstract_client.AbstractClient*

Handle Spark session connection.

Get query results with SQL, reads and writes data on external systems.

add_table_partitions (*partitions: List[Dict[str, Any]], table: str, database: str = None*) → None

Add partitions to an existing table.

Parameters

- **partitions** – partitions to add to the table. It's expected a list of partition dicts to add to the table. Example: [{"year": 2020, "month": 8, "day": 14}, ...]
- **table** – table to add the partitions.
- **database** – name of the database where the table is saved.

property conn

Gets or creates an SparkSession.

Returns Spark session

static create_temporary_view (*dataframe: pyspark.sql.dataframe.DataFrame, name: str*) → Any

Create a temporary view from a given dataframe.

Parameters

- **dataframe** – dataframe to be queried by the view.
- **name** – name of the temporary view.

get_schema (*table: str, database: str = None*) → List[Dict[str, str]]

Returns desired table schema.

table

desired table.

Returns A list of dictionaries in the format [{"column_name": "example1", type: "Spark_type"}, ...]

read (*format: str, path: Optional[Union[str, List[str]]] = None, schema: Optional[pyspark.sql.types.StructType] = None, stream: bool = False, **options: Any*) → pyspark.sql.dataframe.DataFrame

Use the SparkSession.read interface to load data into a dataframe.

Check docs for more information: <https://spark.apache.org/docs/latest/sql-data-sources-load-save-functions.html#generic-loadsave-functions>

Parameters

- **format** – string with the format to be used by the DataframeReader.
- **path** – optional string or a list of string for file-system.
- **stream** – flag to indicate if data must be read in stream mode.
- **schema** – an optional pyspark.sql.types.StructType for the input schema.
- **options** – options to setup the DataframeReader.

Returns Dataframe

read_table (*table: str, database: str = None*) → pyspark.sql.dataframe.DataFrame

Use the SparkSession.read interface to read a metastore table.

Parameters

- **database** – name of the metastore database/schema
- **table** – name of the table in metastore

Returns Dataframe

sql (*query: str*) → pyspark.sql.dataframe.DataFrame

Run a query using Spark SQL.

Parameters **query** – Spark SQL query.

Returns Dataframe

static write_dataframe (*dataframe: pyspark.sql.dataframe.DataFrame, format_: str, mode: str, **options: Any*) → None

Receive a spark DataFrame and write it.

Parameters

- **dataframe** – dataframe containing data from a feature set.
- **format** – format used to save the dataframe.
- **mode** – writing modem can be “error”, “append”, “overwrite” or “ignore”. For more information: [here](<https://spark.apache.org/docs/2.3.0/sql-programming-guide.html#save-modes>).
- ****options** – all other options that can be used in a DataFrameWriter.

write_stream (*dataframe: pyspark.sql.dataframe.DataFrame, processing_time: str, output_mode: str, checkpoint_path: Optional[str], format_: str, mode: str, **options: Any*) → pyspark.sql.streaming.StreamingQuery

Starts streaming data writing job.

Parameters

- **dataframe** – Spark dataframe containing data from a feature set.

- **processing_time** – a processing time interval as a string. E.g. ‘5 seconds’, ‘1 minute’. Set a trigger that runs the mini-batch periodically based on the processing time. If the effect of processing data as soon as the data arrives, without having to wait for the time frame, is desired, the value ‘0 seconds’ can be set.
- **output_mode** – specifies how data of a streaming DataFrame/Dataset is written to a streaming sink destination.
- **checkpoint_path** – path on S3 to save checkpoints for the stream job. These checkpoints can be used on the the job re-start to return from where it stops.
- **format** – format used to save the dataframe.
- **mode** – writing modem can be “error”, “append”, “overwrite” or “ignore”. For more information: [here](https://spark.apache.org/docs/2.3.0/sql-programming-guide.html#save-modes).
- ****options** – all other options that can be used in a DataFrameWriter.

More information about `processing_time`, `output_mode` and `checkpoint_path` can be found in Spark documentation: [here](https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html)

Returns Streaming handler.

```
static write_table (dataframe: pyspark.sql.dataframe.DataFrame, database: Optional[str], table_name: str, path: str, format_: str = None, mode: str = None, partition_by: List[str] = None, **options: Any) → None
```

Receive a spark DataFrame and write it as a table in metastore.

Parameters

- **dataframe** – spark dataframe containing data from a feature set.
- **database** – specified database name.
- **table_name** – specified table name.
- **path** – string with the local to save the table.
- **format** – string with the format used to save.
- **mode** – writing mode, it can be: “error”, “append”, “overwrite” or “ignore”. More information: [here](https://spark.apache.org/docs/2.3.0/sql-programming-guide.html#save-modes).
- **partition_by** – names of partitioning columns.
- **options** – all other options that can be used in a DataFrameWriter.

butterfree.configs package

Subpackages

butterfree.configs.db package

Submodules

Abstract classes for database configurations with spark.

class `butterfree.configs.db.abstract_config.AbstractWriteConfig`

Bases: `abc.ABC`

Abstract class for database write configurations with spark.

abstract property database

Database name.

abstract property format_

Config option “format” for spark write.

Args:

Returns format.

Return type str

abstract property mode

Config option “mode” for spark write.

Args:

Returns mode.

Return type str

abstract translate (*schema: Any*) → List[Dict[Any, Any]]

Translate feature set spark schema to the corresponding database.

Parameters *schema* – feature set schema

Returns Corresponding database schema.

Holds configurations to read and write with Spark to Cassandra DB.

```
class butterfree.configs.db.cassandra_config.CassandraConfig (username:      str
                                                             = None, password: str = None,
                                                             host: str = None,
                                                             keyspace: str
                                                             = None, mode:
str = None, format_: str = None,
stream_processing_time:
str = None,
stream_output_mode:
str = None,
stream_checkpoint_path:
str = None,
read_consistency_level:
str = None,
write_consistency_level:
str = None, local_dc: str =
None)
```

Bases: `butterfree.configs.db.abstract_config.AbstractWriteConfig`

Configuration for Spark to connect on Cassandra DB.

References can be found [here](<https://docs.databricks.com/data/data-sources/cassandra.html>).

username

username to use in connection.

password

password to use in connection.

host

host to use in connection.

keyspace

Cassandra DB keyspace to write data.

mode

write mode for Spark.

format_

write format for Spark.

stream_processing_time

processing time interval for streaming jobs.

stream_output_mode

specify the mode from writing streaming data.

stream_checkpoint_path

path on S3 to save checkpoints for the stream job.

read_consistency_level

read consistency level used in connection.

write_consistency_level

write consistency level used in connection.

More information about `processing_time`, `output_mode` and `checkpoint_path` can be found in Spark documentation: [here](<https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html>)

property database

Database name.

property format_

Write format for Spark.

get_options (*table: str*) → Dict[Optional[str], Optional[str]]

Get options for connect to Cassandra DB.

Options will be a dictionary with the write and read configuration for spark to cassandra.

Parameters **table** – table name (keyspace) into Cassandra DB.

Returns Configuration to connect to Cassandra DB.

property host

Host used in connection to Cassandra DB.

property keyspace

Cassandra DB keyspace to write data.

property local_dc

Local DC for Cassandra connection.

property mode

Write mode for Spark.

property password

Password used in connection to Cassandra DB.

property read_consistency_level

Read consistency level for Cassandra.

property stream_checkpoint_path

Path on S3 to save checkpoints for the stream job.

property stream_output_mode

Specify the mode from writing streaming data.

property stream_processing_time

Processing time interval for streaming jobs.

translate (*schema: List[Dict[str, Any]]*) → List[Dict[str, Any]]

Get feature set schema to be translated.

The output will be a list of dictionaries regarding cassandra database schema.

Parameters *schema* – feature set schema in spark.

Returns Cassandra schema.

property username

Username used in connection to Cassandra DB.

property write_consistency_level

Write consistency level for Cassandra.

Holds configurations to read and write with Spark to Kafka.

```
class butterfree.configs.db.kafka_config.KafkaConfig (kafka_topic: str = None,  
kafka_connection_string:  
str = None, mode: str =  
None, format_: str = None,  
stream_processing_time: str  
= None, stream_output_mode:  
str = None,  
stream_checkpoint_path:  
str = None)
```

Bases: *butterfree.configs.db.abstract_config.AbstractWriteConfig*

Configuration for Spark to connect to Kafka.

kafka_topic

string with kafka topic name.

kafka_connection_string

string with hosts and ports to connect.

mode

write mode for Spark.

format_

write format for Spark.

stream_processing_time

processing time interval for streaming jobs.

stream_output_mode

specify the mode from writing streaming data.

stream_checkpoint_path

path on S3 to save checkpoints for the stream job.

More information about *processing_time*, *output_mode* and *checkpoint_path* can be found in Spark documentation: [here](<https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html>)

property database

Database name.

property format_

Write format for Spark.

get_options (*topic: str*) → Dict[Optional[str], Optional[str]]

Get options for connecting to Kafka.

Options will be a dictionary with the write and read configuration for spark to kafka.

Parameters **topic** – topic related to Kafka.

Returns Configuration to connect to Kafka.

property kafka_connection_string

Kafka connection string with hosts and ports to connect.

property kafka_topic

Kafka topic name.

property mode

Write mode for Spark.

property stream_checkpoint_path

Path on S3 to save checkpoints for the stream job.

property stream_output_mode

Specify the mode from writing streaming data.

property stream_processing_time

Processing time interval for streaming jobs.

translate (*schema: List[Dict[str, Any]]*) → List[Dict[str, Any]]

Get feature set schema to be translated.

The output will be a list of dictionaries regarding cassandra database schema.

Parameters **schema** – feature set schema in spark.

Returns Kafka schema.

Holds configurations to read and write with Spark to AWS S3.

```
class butterfree.configs.db.metastore_config.MetastoreConfig (path: str = None,  
mode: str = None,  
format_: str =  
None, file_system:  
str = None)
```

Bases: *butterfree.configs.db.abstract_config.AbstractWriteConfig*

Configuration for Spark metastore database stored.

By default the configuration is for AWS S3.

path

database root location.

mode

writing mode used be writers.

format_

expected stored file format.

file_system

file schema uri, like: s3a, file.

property database

Database name.

property file_system

Writing mode used by writers.

property format_

Expected stored file format.

get_options (*key: str*) → Dict[Optional[str], Optional[str]]

Get options for Metastore.

Options will be a dictionary with the write and read configuration for Spark Metastore.

Parameters **key** – path to save data into Metastore.

Returns Options configuration for Metastore.

get_path_with_partitions (*key: str, dataframe: pyspark.sql.dataframe.DataFrame*) → List

Get options for AWS S3 from partitioned parquet file.

Options will be a dictionary with the write and read configuration for Spark to AWS S3.

Parameters

- **key** – path to save data into AWS S3 bucket.
- **dataframe** – spark dataframe containing data from a feature set.

Returns A list of string for file-system backed data sources.

property mode

Writing mode used by writers.

property path

Bucket name.

translate (*schema: List[Dict[str, Any]]*) → List[Dict[str, Any]]

Translate feature set spark schema to the corresponding database.

Module contents

This module holds database configurations to be used by clients.

class butterfree.configs.db.**AbstractWriteConfig**

Bases: abc.ABC

Abstract class for database write configurations with spark.

abstract property database

Database name.

abstract property format_

Config option “format” for spark write.

Args:

Returns format.

Return type str

abstract property mode

Config option “mode” for spark write.

Args:

Returns mode.

Return type str

abstract translate (*schema: Any*) → List[Dict[Any, Any]]

Translate feature set spark schema to the corresponding database.

Parameters **schema** – feature set schema

Returns Corresponding database schema.

```
class butterfree.configs.db.CassandraConfig(username: str = None, password: str = None, host: str = None, keyspace: str = None, mode: str = None, format_: str = None, stream_processing_time: str = None, stream_output_mode: str = None, stream_checkpoint_path: str = None, read_consistency_level: str = None, write_consistency_level: str = None, local_dc: str = None)
```

Bases: *butterfree.configs.db.abstract_config.AbstractWriteConfig*

Configuration for Spark to connect on Cassandra DB.

References can be found [here](<https://docs.databricks.com/data/data-sources/cassandra.html>).

username

username to use in connection.

password

password to use in connection.

host

host to use in connection.

keyspace

Cassandra DB keyspace to write data.

mode

write mode for Spark.

format_

write format for Spark.

stream_processing_time

processing time interval for streaming jobs.

stream_output_mode

specify the mode from writing streaming data.

stream_checkpoint_path

path on S3 to save checkpoints for the stream job.

read_consistency_level

read consistency level used in connection.

write_consistency_level

write consistency level used in connection.

More information about `processing_time`, `output_mode` and `checkpoint_path` can be found in Spark documentation: [here](<https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html>)

property database

Database name.

property format_

Write format for Spark.

get_options (*table: str*) → Dict[Optional[str], Optional[str]]

Get options for connect to Cassandra DB.

Options will be a dictionary with the write and read configuration for spark to cassandra.

Parameters table – table name (keyspace) into Cassandra DB.

Returns Configuration to connect to Cassandra DB.

property host

Host used in connection to Cassandra DB.

property keyspace

Cassandra DB keyspace to write data.

property local_dc

Local DC for Cassandra connection.

property mode

Write mode for Spark.

property password

Password used in connection to Cassandra DB.

property read_consistency_level

Read consistency level for Cassandra.

property stream_checkpoint_path

Path on S3 to save checkpoints for the stream job.

property stream_output_mode

Specify the mode from writing streaming data.

property stream_processing_time

Processing time interval for streaming jobs.

translate (*schema: List[Dict[str, Any]]*) → List[Dict[str, Any]]

Get feature set schema to be translated.

The output will be a list of dictionaries regarding cassandra database schema.

Parameters schema – feature set schema in spark.

Returns Cassandra schema.

property username

Username used in connection to Cassandra DB.

property write_consistency_level

Write consistency level for Cassandra.

class butterfree.configs.db.**KafkaConfig** (*kafka_topic: str = None, kafka_connection_string: str = None, mode: str = None, format_: str = None, stream_processing_time: str = None, stream_output_mode: str = None, stream_checkpoint_path: str = None*)

Bases: *butterfree.configs.db.abstract_config.AbstractWriteConfig*

Configuration for Spark to connect to Kafka.

kafka_topic

string with kafka topic name.

kafka_connection_string
string with hosts and ports to connect.

mode
write mode for Spark.

format_
write format for Spark.

stream_processing_time
processing time interval for streaming jobs.

stream_output_mode
specify the mode from writing streaming data.

stream_checkpoint_path
path on S3 to save checkpoints for the stream job.

More information about `processing_time`, `output_mode` and `checkpoint_path` can be found in Spark documentation: [here](<https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html>)

property database
Database name.

property format_
Write format for Spark.

get_options (*topic: str*) → Dict[Optional[str], Optional[str]]
Get options for connecting to Kafka.

Options will be a dictionary with the write and read configuration for spark to kafka.

Parameters `topic` – topic related to Kafka.

Returns Configuration to connect to Kafka.

property kafka_connection_string
Kafka connection string with hosts and ports to connect.

property kafka_topic
Kafka topic name.

property mode
Write mode for Spark.

property stream_checkpoint_path
Path on S3 to save checkpoints for the stream job.

property stream_output_mode
Specify the mode from writing streaming data.

property stream_processing_time
Processing time interval for streaming jobs.

translate (*schema: List[Dict[str, Any]]*) → List[Dict[str, Any]]
Get feature set schema to be translated.

The output will be a list of dictionaries regarding cassandra database schema.

Parameters `schema` – feature set schema in spark.

Returns Kafka schema.

class `butterfree.configs.db.MetastoreConfig` (*path: str = None, mode: str = None, format_:*
str = None, file_system: str = None)
Bases: `butterfree.configs.db.abstract_config.AbstractWriteConfig`

Configuration for Spark metastore database stored.

By default the configuration is for AWS S3.

path

database root location.

mode

writing mode used be writers.

format_

expected stored file format.

file_system

file schema uri, like: s3a, file.

property database

Database name.

property file_system

Writing mode used be writers.

property format_

Expected stored file format.

get_options (*key: str*) → Dict[Optional[str], Optional[str]]

Get options for Metastore.

Options will be a dictionary with the write and read configuration for Spark Metastore.

Parameters **key** – path to save data into Metastore.

Returns Options configuration for Metastore.

get_path_with_partitions (*key: str, dataframe: pyspark.sql.dataframe.DataFrame*) → List

Get options for AWS S3 from partitioned parquet file.

Options will be a dictionary with the write and read configuration for Spark to AWS S3.

Parameters

- **key** – path to save data into AWS S3 bucket.
- **dataframe** – spark dataframe containing data from a feature set.

Returns A list of string for file-system backed data sources.

property mode

Writing mode used be writers.

property path

Bucket name.

translate (*schema: List[Dict[str, Any]]*) → List[Dict[str, Any]]

Translate feature set spark schema to the corresponding database.

Submodules

Holds functions for managing the running environment.

exception `butterfree.configs.environment.UnspecifiedVariableError` (*variable_name: str*)

Bases: `RuntimeError`

Environment variables not set error.

variable_name

environment variable name.

`butterfree.configs.environment.get_variable` (*variable_name: str, default_value: str = None*) → `Optional[str]`

Gets an environment variable.

The variable comes from it's explicitly declared value in the running environment or from the default value declared in specification or from the `default_value`.

Parameters

- **variable_name** – environment variable name.
- **default_value** – default value to use in case no value is set in the environment nor in the `environment.yaml` specification file.

Returns The variable's string value

Logger funcion.

Module contents

Holds configuration/setup for Butterfree components.

`butterfree.constants` package

Submodules

Holds common column names, constant through all Butterfree.

`DataType` Enum Entity.

class `butterfree.constants.data_type.DataType` (*value*)

Bases: `enum.Enum`

Holds constants for data types within Butterfree.

`ARRAY_BIGINT = (ArrayType(LongType, true), 'frozen<list<bigint>>', 'ARRAY<BIGINT>')`

`ARRAY_FLOAT = (ArrayType(FloatType, true), 'frozen<list<float>>', 'ARRAY<FLOAT>')`

`ARRAY_STRING = (ArrayType(StringType, true), 'frozen<list<text>>', 'ARRAY<STRING>')`

`BIGINT = (LongType, 'bigint', 'BIGINT')`

`BINARY = (BinaryType, 'boolean', 'BINARY')`

`BOOLEAN = (BooleanType, 'boolean', 'BOOLEAN')`

`DATE = (DateType, 'timestamp', 'DATE')`

```
DECIMAL = (DecimalType(10,0), 'decimal', 'DECIMAL')
DOUBLE = (DoubleType, 'double', 'DOUBLE')
FLOAT = (FloatType, 'float', 'FLOAT')
INTEGER = (IntegerType, 'int', 'INT')
STRING = (StringType, 'text', 'STRING')
TIMESTAMP = (TimestampType, 'timestamp', 'TIMESTAMP')
```

Migrations' Constants.

Holds common spark constants, present through all Butterfree.

Allowed windows units and lengths in seconds.

Module contents

Holds constant attributes that are common for Butterfree.

```
class butterfree.constants.DataType(value)
```

Bases: `enum.Enum`

Holds constants for data types within Butterfree.

```
ARRAY_BIGINT = (ArrayType(LongType,true), 'frozen<list<bigint>>', 'ARRAY<BIGINT>')
ARRAY_FLOAT = (ArrayType(FloatType,true), 'frozen<list<float>>', 'ARRAY<FLOAT>')
ARRAY_STRING = (ArrayType(StringType,true), 'frozen<list<text>>', 'ARRAY<STRING>')
BIGINT = (LongType, 'bigint', 'BIGINT')
BINARY = (BinaryType, 'boolean', 'BINARY')
BOOLEAN = (BooleanType, 'boolean', 'BOOLEAN')
DATE = (DateType, 'timestamp', 'DATE')
DECIMAL = (DecimalType(10,0), 'decimal', 'DECIMAL')
DOUBLE = (DoubleType, 'double', 'DOUBLE')
FLOAT = (FloatType, 'float', 'FLOAT')
INTEGER = (IntegerType, 'int', 'INT')
STRING = (StringType, 'text', 'STRING')
TIMESTAMP = (TimestampType, 'timestamp', 'TIMESTAMP')
```

butterfree.dataframe_service package

Submodules

IncrementalStrategy entity.

```
class butterfree.dataframe_service.incremental_strategy.IncrementalStrategy (column:
                                                                    Optional[str]
                                                                    =
                                                                    None)
```

Bases: object

Define an incremental strategy to be used on data sources.

Entity responsible for defining a column expression that will be used to filter the original data source. The purpose is to get only the data related to a specific pipeline execution time interval.

column

column expression on which incremental filter will be applied. The expression need to result on a date or timestamp format, so the filter can properly work with the defined upper and lower bounds.

```
filter_with_incremental_strategy (dataframe:          pyspark.sql.dataframe.DataFrame,
                                   start_date:       Optional[str]      = None,
                                   end_date:         Optional[str]    = None) → pys-
                                   park.sql.dataframe.DataFrame
```

Filters the dataframe according to the date boundaries.

Parameters

- **dataframe** – dataframe that will be filtered.
- **start_date** – date lower bound to use in the filter.
- **end_date** – date upper bound to use in the filter.

Returns Filtered dataframe based on defined time boundaries.

```
from_milliseconds (column_name: str) → butterfree.dataframe_service.incremental_strategy.IncrementalStrategy
Create a column expression from ts column defined as milliseconds.
```

Parameters **column_name** – column name where the filter will be applied.

Returns *IncrementalStrategy* with the defined column expression.

```
from_string (column_name: str, mask: Optional[str] = None) → butter-
                                   free.dataframe_service.incremental_strategy.IncrementalStrategy
Create a column expression from ts column defined as a simple string.
```

Parameters

- **column_name** – column name where the filter will be applied.
- **mask** – mask defining the date/timestamp format on the string.

Returns *IncrementalStrategy* with the defined column expression.

```
from_year_month_day_partitions (year_column: str = 'year', month_column: str
                                   = 'month', day_column: str = 'day') → butter-
                                   free.dataframe_service.incremental_strategy.IncrementalStrategy
Create a column expression from year, month and day partitions.
```

Parameters

- **year_column** – column name from the year partition.
- **month_column** – column name from the month partition.
- **day_column** – column name from the day partition.

Returns *IncrementalStrategy* with the defined column expression.

get_expression (*start_date: Optional[str] = None, end_date: Optional[str] = None*) → str
Get the incremental filter expression using the defined dates.

Both arguments can be set to defined a specific date interval, but it's only necessary to set one of the arguments for this method to work.

Parameters

- **start_date** – date lower bound to use in the filter.
- **end_date** – date upper bound to use in the filter.

Returns Filter expression based on defined column and bounds.

Raises

- **ValueError** – If both arguments, start_date and end_date, are None.
- **ValueError** – If the column expression was not defined.

Module defining partitioning methods.

```
butterfree.dataframe_service.partitioning.extract_partition_values (dataframe: pys-  
park.sql.dataframe.DataFrame,  
partition_columns: List[str])  
→  
List[Dict[str, Any]]
```

Extract distinct partition values from a given dataframe.

Parameters

- **dataframe** – dataframe from where to extract partition values.
- **partition_columns** – name of partition columns presented on the dataframe.

Returns distinct partition values.

Module where there are repartition methods.

```
butterfree.dataframe_service.repartition.repartition_df (dataframe: pys-  
park.sql.dataframe.DataFrame,  
partition_by: List[str],  
num_partitions: int = None,  
num_processors: int = None) → pys-  
park.sql.dataframe.DataFrame
```

Partition the DataFrame.

Parameters

- **dataframe** – Spark DataFrame.
- **partition_by** – list of partitions.
- **num_processors** – number of processors.
- **num_partitions** – number of partitions.

Returns Partitioned dataframe.


```
butterfree.dataframe_service.repartition.repartition_sort_df (dataframe: pyspark.sql.dataframe.DataFrame,
    partition_by: List[str],
    order_by: List[str],
    num_processors: int = None,
    num_partitions: int = None) → pyspark.sql.dataframe.DataFrame
```

Partition and Sort the DataFrame.

Parameters

- **dataframe** – Spark DataFrame.
- **partition_by** – list of columns to partition by.
- **order_by** – list of columns to order by.
- **num_processors** – number of processors.
- **num_partitions** – number of partitions.

Returns Partitioned and sorted dataframe.

Module contents

Dataframe optimization components regarding Butterfree.

```
class butterfree.dataframe_service.IncrementalStrategy (column: Optional[str] = None)
```

Bases: object

Define an incremental strategy to be used on data sources.

Entity responsible for defining a column expression that will be used to filter the original data source. The purpose is to get only the data related to a specific pipeline execution time interval.

column

column expression on which incremental filter will be applied. The expression need to result on a date or timestamp format, so the filter can properly work with the defined upper and lower bounds.

```
filter_with_incremental_strategy (dataframe: pyspark.sql.dataframe.DataFrame,
    start_date: Optional[str] = None,
    end_date: Optional[str] = None) → pyspark.sql.dataframe.DataFrame
```

Filters the dataframe according to the date boundaries.

Parameters

- **dataframe** – dataframe that will be filtered.
- **start_date** – date lower bound to use in the filter.
- **end_date** – date upper bound to use in the filter.

Returns Filtered dataframe based on defined time boundaries.

```
from_milliseconds (column_name: str) → butterfree.dataframe_service.incremental_strategy.IncrementalStrategy
```

Create a column expression from ts column defined as milliseconds.

Parameters **column_name** – column name where the filter will be applied.

Returns *IncrementalStrategy* with the defined column expression.

from_string (*column_name: str, mask: Optional[str] = None*) → *butterfree.dataframe_service.incremental_strategy.IncrementalStrategy*
Create a column expression from ts column defined as a simple string.

Parameters

- **column_name** – column name where the filter will be applied.
- **mask** – mask defining the date/timestamp format on the string.

Returns *IncrementalStrategy* with the defined column expression.

from_year_month_day_partitions (*year_column: str = 'year', month_column: str = 'month', day_column: str = 'day'*) → *butterfree.dataframe_service.incremental_strategy.IncrementalStrategy*
Create a column expression from year, month and day partitions.

Parameters

- **year_column** – column name from the year partition.
- **month_column** – column name from the month partition.
- **day_column** – column name from the day partition.

Returns *IncrementalStrategy* with the defined column expression.

get_expression (*start_date: Optional[str] = None, end_date: Optional[str] = None*) → *str*
Get the incremental filter expression using the defined dates.

Both arguments can be set to defined a specific date interval, but it's only necessary to set one of the arguments for this method to work.

Parameters

- **start_date** – date lower bound to use in the filter.
- **end_date** – date upper bound to use in the filter.

Returns Filter expression based on defined column and bounds.

Raises

- **ValueError** – If both arguments, *start_date* and *end_date*, are *None*.
- **ValueError** – If the column expression was not defined.

butterfree.dataframe_service.extract_partition_values (*dataframe: pyspark.sql.dataframe.DataFrame, partition_columns: List[str]*) → *List[Dict[str, Any]]*

Extract distinct partition values from a given dataframe.

Parameters

- **dataframe** – dataframe from where to extract partition values.
- **partition_columns** – name of partition columns presented on the dataframe.

Returns distinct partition values.

```
butterfree.dataframe_service.repartition_df (dataframe: pys-
park.sql.dataframe.DataFrame, parti-
tion_by: List[str], num_partitions: int =
None, num_processors: int = None) →
pyspark.sql.dataframe.DataFrame
```

Partition the DataFrame.

Parameters

- **dataframe** – Spark DataFrame.
- **partition_by** – list of partitions.
- **num_processors** – number of processors.
- **num_partitions** – number of partitions.

Returns Partitioned dataframe.

```
butterfree.dataframe_service.repartition_sort_df (dataframe: pys-
park.sql.dataframe.DataFrame,
partition_by: List[str], order_by:
List[str], num_processors: int =
None, num_partitions: int = None)
→ pyspark.sql.dataframe.DataFrame
```

Partition and Sort the DataFrame.

Parameters

- **dataframe** – Spark DataFrame.
- **partition_by** – list of columns to partition by.
- **order_by** – list of columns to order by.
- **num_processors** – number of processors.
- **num_partitions** – number of partitions.

Returns Partitioned and sorted dataframe.

butterfree.extract package

Subpackages

butterfree.extract.pre_processing package

Submodules

Explode json column for dataframes.

butterfree.extract.pre_processing.explode_json_column_transform.**explode_json_column** (*df*:

pys-
park.sql.c
col-
umn:
str,
json_sche
pys-
park.sql.t
→
pys-
park.sql.d

Create new columns extracting properties from a JSON column.

Example:

```
>>> from pyspark import SparkContext
>>> from pyspark.sql import session
>>> from butterfree.testing.dataframe import create_df_from_collection
>>> from butterfree.extract.pre_processing import explode_json_column
>>> from pyspark.sql.types import (
...     ArrayType,
...     IntegerType,
...     StringType,
...     StructField,
...     StructType,
... )
>>> spark_context = SparkContext.getOrCreate()
>>> spark_session = session.SparkSession(spark_context)
>>> data = [{"json_column": '{"a": 123, "b": "abc", "c": "123", "d": [1, 2, 3]}'}]
>>> df = create_df_from_collection(data, spark_context, spark_session)
>>> df.collect()
```

```
[Row(json_column='{"a": 123, "b": "abc", "c": "123", "d": [1, 2, 3]}')]
```

```
>>> json_column_schema = StructType(
... [
...     StructField("a", IntegerType()),
...     StructField("b", StringType()),
...     StructField("c", IntegerType()),
...     StructField("d", ArrayType(IntegerType())),
... ]
>>> explode_json_column(
...     df, column='json_column', json_schema=json_column_schema
... ).collect()
```

```
[
  Row( json_column='{"a": 123, "b": "abc", "c": "123", "d": [1, 2, 3]}', a=123, b='abc', c=123, d=[1, 2, 3]
)
]
```

Parameters

- **df** – input dataframe with the target JSON column.

- **column** – column name that is going to be exploded.
- **json_schema** – expected schema from that JSON column. Not all “first layer” fields need to be mapped in the json_schema, just the desired columns. If there is any JSON field that is needed to be cast to a struct, the declared expected schema (a StructType) need to have the exact same schema as the presented record, if don't, the value in the resulting column will be null.

Returns dataframe with the new extracted columns from the JSON column.

Module where filter DataFrames coming from readers.

```
butterfree.extract.pre_processing.filter_transform.filter (dataframe: pys-
park.sql.dataframe.DataFrame,
condition: str) → pys-
park.sql.dataframe.DataFrame
```

Filters DataFrame's rows using the given condition and value.

Parameters

- **dataframe** – Spark DataFrame.
- **condition** – SQL expression with column, operation and value to filter the dataframe.

Returns Filtered dataframe

Forward Fill Transform for dataframes.

```
butterfree.extract.pre_processing.forward_fill_transform.forward_fill (dataframe:
pys-
park.sql.dataframe.DataFrame
partition_by:
Union[str,
List[str]],
order_by:
Union[str,
List[str]],
fill_column:
str,
filled_column:
str =
None)
→
pys-
park.sql.dataframe.DataFrame
```

Applies a forward fill to a single column.

Filling null values with the last known non-null value, leaving leading nulls alone.

```
butterfree.extract.pre_processing.forward_fill_transform.dataframe
dataframe to be transformed.
```

```
butterfree.extract.pre_processing.forward_fill_transform.partition_by
list of columns' names to be used as partition for the operation.
```

```
butterfree.extract.pre_processing.forward_fill_transform.order_by
list of columns' names to be used when sorting column values.
```

```
butterfree.extract.pre_processing.forward_fill_transform.fill_column
column to be forward filled.
```

`butterfree.extract.pre_processing.forward_fill_transform.filled_column`
 new column name. Optional. When none, operation will be inplace.

Example

```
>>> dataframe.orderBy("ts", "sensor_type", "location").show()
+-----+-----+-----+-----+
|sensor_type|          ts|location|temperature|
+-----+-----+-----+-----+
|          1|2017-09-09 12:00:00|  shade|  18.83018|
|          1|2017-09-09 12:00:00|   sun|      null|
|          2|2017-09-09 12:00:00|  shade|  18.61258|
|          2|2017-09-09 12:00:00|   sun|  25.4986|
|          1|2017-09-09 13:00:00|  shade|  18.78458|
|          1|2017-09-09 13:00:00|   sun|  25.68457|
|          2|2017-09-09 13:00:00|  shade|      null|
|          2|2017-09-09 13:00:00|   sun|      null|
|          1|2017-09-09 14:00:00|  shade|  17.98115|
|          1|2017-09-09 14:00:00|   sun|  24.15754|
|          2|2017-09-09 14:00:00|  shade|  18.61258|
|          2|2017-09-09 14:00:00|   sun|      null|
+-----+-----+-----+-----+
```

```
>>> filled_df = forward_fill(
...     dataframe,
...     ["sensor_type", "location"],
...     "ts",
...     "temperature",
...     "temperature_filled"
... )
>>> filled_df.orderBy("ts", "sensor_type", "location").show()
+-----+-----+-----+-----+-----+
|sensor_type|          ts|location|temperature|temperature_filled|
+-----+-----+-----+-----+-----+
|          1|2017-09-09 12:00:00|  shade|  18.83018|  18.83018|
|          1|2017-09-09 12:00:00|   sun|      null|      null|
|          2|2017-09-09 12:00:00|  shade|  18.61258|  18.61258|
|          2|2017-09-09 12:00:00|   sun|  25.4986|  25.4986|
|          1|2017-09-09 13:00:00|  shade|  18.78458|  18.78458|
|          1|2017-09-09 13:00:00|   sun|  25.68457|  25.68457|
|          2|2017-09-09 13:00:00|  shade|      null|  18.61258|
|          2|2017-09-09 13:00:00|   sun|      null|  25.4986|
|          1|2017-09-09 14:00:00|  shade|  17.98115|  17.98115|
|          1|2017-09-09 14:00:00|   sun|  24.15754|  24.15754|
|          2|2017-09-09 14:00:00|  shade|  18.61258|  18.61258|
|          2|2017-09-09 14:00:00|   sun|      null|  25.4986|
+-----+-----+-----+-----+-----+
>>> # inplace forward fill
>>> filled_df = forward_fill(
...     dataframe,
...     ["sensor_type", "location"],
...     "ts",
...     "temperature"
... )
>>> filled_df.orderBy("ts", "sensor_type", "location").show()
+-----+-----+-----+-----+-----+
```

(continues on next page)

(continued from previous page)

sensor_type	ts	location	temperature
1	2017-09-09 12:00:00	shade	18.83018
1	2017-09-09 12:00:00	sun	null
2	2017-09-09 12:00:00	shade	18.61258
2	2017-09-09 12:00:00	sun	25.4986
1	2017-09-09 13:00:00	shade	18.78458
1	2017-09-09 13:00:00	sun	25.68457
2	2017-09-09 13:00:00	shade	18.61258
2	2017-09-09 13:00:00	sun	25.4986
1	2017-09-09 14:00:00	shade	17.98115
1	2017-09-09 14:00:00	sun	24.15754
2	2017-09-09 14:00:00	shade	18.61258
2	2017-09-09 14:00:00	sun	25.4986

Pivot Transform for dataframes.

```
butterfree.extract.pre_processing.pivot_transform.pivot (dataframe: pyspark.sql.dataframe.DataFrame,
group_by_columns: List[str], pivot_column: str, agg_column: str,
aggregation: Callable, mock_value: Union[float, str] = None, mock_type: Union[pyspark.sql.types.DataType, str] = None,
with_forward_fill: bool = False) → pyspark.sql.dataframe.DataFrame
```

Defines a pivot transformation.

`butterfree.extract.pre_processing.pivot_transform.dataframe`
dataframe to be pivoted.

`butterfree.extract.pre_processing.pivot_transform.group_by_columns`
list of columns' names to be grouped.

`butterfree.extract.pre_processing.pivot_transform.pivot_column`
column to be pivoted.

`butterfree.extract.pre_processing.pivot_transform.agg_column`
column to be aggregated by pivoted category.

`butterfree.extract.pre_processing.pivot_transform.aggregation`
desired spark aggregation function to be performed. An example: `spark_agg(col_name)`. See docs for all `spark_agg`: https://spark.apache.org/docs/2.3.1/api/python/_modules/pyspark/sql/functions.html

`butterfree.extract.pre_processing.pivot_transform.mock_value`
value used to make a difference between true nulls resulting from the aggregation and empty values from the pivot transformation.

`butterfree.extract.pre_processing.pivot_transform.mock_type`
mock_value data type (compatible with spark).

`butterfree.extract.pre_processing.pivot_transform.with_forward_fill`
applies a forward fill to null values after the pivot operation.

Example

```
>>> dataframe.orderBy("ts", "id", "amenity").show()
+---+---+-----+---+
| id| ts|amenity| has|
+---+---+-----+---+
| 1| 1| fridge|false|
| 1| 1|  oven| true|
| 1| 1|  pool|false|
| 2| 2|balcony|false|
| 1| 3|balcony| null|
| 1| 4|  oven| null|
| 1| 4|  pool| true|
| 1| 5|balcony| true|
+---+---+-----+---+
```

```
>>> pivoted = pivot(dataframe, ["id", "ts"], "amenity", "has", functions.first)
>>> pivoted.orderBy("ts", "id").show()
+---+---+-----+---+-----+---+-----+
| id| ts|balcony|fridge|oven| pool|
+---+---+-----+---+-----+---+-----+
| 1| 1|  null| false|true|false|
| 2| 2| false|  null|null| null|
| 1| 3|  null|  null|null| null|
| 1| 4|  null|  null|null| true|
| 1| 5| true|  null|null| null|
+---+---+-----+---+-----+---+-----+
```

But, sometimes, you would like to keep the last values that some feature has assumed from previous modifications. In this example, amenity “oven” for the id=1 was set to null and “pool” was set to true at ts=4. All other amenities should then be kept to their actual state at that ts. To do that, we will use a technique called forward fill:

```
>>> pivoted = pivot(
...     dataframe,
...     ["id", "ts"],
...     "amenity",
...     "has",
...     functions.first,
...     with_forward_fill=True
... )
>>> pivoted.orderBy("ts", "id").show()
+---+---+-----+---+-----+---+-----+
| id| ts|balcony|fridge|oven| pool|
+---+---+-----+---+-----+---+-----+
| 1| 1|  null| false|true|false|
| 2| 2| false|  null|null| null|
| 1| 3|  null| false|true|false|
| 1| 4|  null| false|true| true|
| 1| 5| true| false|true| true|
+---+---+-----+---+-----+---+-----+
```

Great! Now every amenity that didn’t have been changed kept its state. BUT, the force change to null for amenity “oven” on id=1 at ts=4 was ignored during forward fill. If the user wants to respect this change, it must provide a mock value and type to be used as a signal for “true nulls”. In other words, we want to forward fill only nulls that were created by the pivot transformation.

In this example, amenities only assume boolean values. So there is no mock values for a boolean. It is only true

or false. So users can give a mock value of another type (for which the column can be cast to). Check this out:

```
>>> pivoted = pivot(
...     dataframe,
...     ["id", "ts"],
...     "amenity",
...     "has",
...     functions.first,
...     with_forward_fill=True,
...     mock_value=-1,
...     mock_type="int"
... )
>>> pivoted.orderBy("ts", "id").show()
+---+---+-----+-----+---+-----+
| id| ts|balcony|fridge|oven| pool|
+---+---+-----+-----+---+-----+
|  1|  1|  null| false|true|false|
|  2|  2| false|  null|null| null|
|  1|  3|  null| false|true|false|
|  1|  4|  null| false|null| true|
|  1|  5|  true| false|null| true|
+---+---+-----+-----+---+-----+
```

During transformation, this method will cast the `agg_column` to `mock_type` data type and fill all “true nulls” with the `mock_value`. After pivot and forward fill are applied, all new pivoted columns will then return to the original type with all mock values replaced by null.

Replace transformer for dataframes.

```
butterfree.extract.pre_processing.replace_transform.replace (dataframe: pyspark.sql.dataframe.DataFrame,
column: str, replace_dict: Dict[str, str]) → pyspark.sql.dataframe.DataFrame
```

Replace values of a string column in the dataframe using a dict.

Example:

```
>>> from butterfree.extract.pre_processing import replace
... from butterfree.testing.dataframe import (
...     assert_dataframe_equality,
...     create_df_from_collection,
... )
>>> from pyspark import SparkContext
>>> from pyspark.sql import session
>>> spark_context = SparkContext.getOrCreate()
>>> spark_session = session.SparkSession(spark_context)
>>> input_data = [
...     {"id":1, "type": "a"}, {"id":2, "type": "b"}, {"id":3, "type": "c"}
... ]
>>> input_df = create_df_from_collection(input_data, spark_context, spark_session)
>>> input_df.collect()
```

```
[Row(id=1, type='a'), Row(id=2, type='b'), Row(id=3, type='c')]
```

```
>>> replace_dict = {"a": "type_a", "b": "type_b"}
>>> replace(input_df, "type", replace_dict).collect()
```

```
[Row(id=1, type='type_a'), Row(id=2, type='type_b'), Row(id=3, type='c')]
```

Parameters

- **dataframe** – data to be transformed.
- **column** – string column on the dataframe where to apply the replace.
- **replace_dict** – dict with values to be replaced. All mapped values must be string.

Returns Dataframe with column values replaced.

Module contents

Pre Processing Components regarding Readers.

```
butterfree.extract.pre_processing.explode_json_column(df: pyspark.sql.dataframe.DataFrame,
column: str, json_schema: pyspark.sql.types.StructType)
→ pyspark.sql.dataframe.DataFrame
```

Create new columns extracting properties from a JSON column.

Example:

```
>>> from pyspark import SparkContext
>>> from pyspark.sql import session
>>> from butterfree.testing.dataframe import create_df_from_collection
>>> from butterfree.extract.pre_processing import explode_json_column
>>> from pyspark.sql.types import (
...     ArrayType,
...     IntegerType,
...     StringType,
...     StructField,
...     StructType,
... )
>>> spark_context = SparkContext.getOrCreate()
>>> spark_session = session.SparkSession(spark_context)
>>> data = [{"json_column": '{"a": 123, "b": "abc", "c": "123", "d": [1, 2, 3]}'}]
>>> df = create_df_from_collection(data, spark_context, spark_session)
>>> df.collect()
```

```
[Row(json_column='{"a": 123, "b": "abc", "c": "123", "d": [1, 2, 3]}')]
```

```
>>> json_column_schema = StructType(
... [
...     StructField("a", IntegerType()),
...     StructField("b", StringType()),
...     StructField("c", IntegerType()),
...     StructField("d", ArrayType(IntegerType())),
... ]
>>> explode_json_column(
...     df, column='json_column', json_schema=json_column_schema
... ).collect()
```

[

```

Row( json_column={'a': 123, "b": "abc", "c": "123", "d": [1, 2, 3]}, a=123, b='abc', c=123, d=[1, 2,
3]
)
]

```

Parameters

- **df** – input dataframe with the target JSON column.
- **column** – column name that is going to be exploded.
- **json_schema** – expected schema from that JSON column. Not all “first layer” fields need to be mapped in the json_schema, just the desired columns. If there is any JSON field that is needed to be cast to a struct, the declared expected schema (a StructType) need to have the exact same schema as the presented record, if don't, the value in the resulting column will be null.

Returns dataframe with the new extracted columns from the JSON column.

```

butterfree.extract.pre_processing.filter (dataframe:    pyspark.sql.dataframe.DataFrame,
condition:      str)      →      pyspark.sql.dataframe.DataFrame

```

Filters DataFrame's rows using the given condition and value.

Parameters

- **dataframe** – Spark DataFrame.
- **condition** – SQL expression with column, operation and value to filter the dataframe.

Returns Filtered dataframe

```

butterfree.extract.pre_processing.forward_fill (dataframe:    pyspark.sql.dataframe.DataFrame,
partition_by: Union[str, List[str]], order_by: Union[str, List[str]],
fill_column: str, filled_column: str = None) → pyspark.sql.dataframe.DataFrame

```

Applies a forward fill to a single column.

Filling null values with the last known non-null value, leaving leading nulls alone.

```

butterfree.extract.pre_processing.dataframe
dataframe to be transformed.

```

```

butterfree.extract.pre_processing.partition_by
list of columns' names to be used as partition for the operation.

```

```

butterfree.extract.pre_processing.order_by
list of columns' names to be used when sorting column values.

```

```

butterfree.extract.pre_processing.fill_column
column to be forward filled.

```

```

butterfree.extract.pre_processing.filled_column
new column name. Optional. When none, operation will be inplace.

```

Example

```
>>> dataframe.orderBy("ts", "sensor_type", "location").show()
+-----+-----+-----+-----+
|sensor_type|          ts|location|temperature|
+-----+-----+-----+-----+
|          1|2017-09-09 12:00:00|  shade|  18.83018|
|          1|2017-09-09 12:00:00|   sun|    null|
|          2|2017-09-09 12:00:00|  shade|  18.61258|
|          2|2017-09-09 12:00:00|   sun|  25.4986|
|          1|2017-09-09 13:00:00|  shade|  18.78458|
|          1|2017-09-09 13:00:00|   sun|  25.68457|
|          2|2017-09-09 13:00:00|  shade|    null|
|          2|2017-09-09 13:00:00|   sun|    null|
|          1|2017-09-09 14:00:00|  shade|  17.98115|
|          1|2017-09-09 14:00:00|   sun|  24.15754|
|          2|2017-09-09 14:00:00|  shade|  18.61258|
|          2|2017-09-09 14:00:00|   sun|    null|
+-----+-----+-----+-----+
```

```
>>> filled_df = forward_fill(
...     dataframe,
...     ["sensor_type", "location"],
...     "ts",
...     "temperature",
...     "temperature_filled"
... )
>>> filled_df.orderBy("ts", "sensor_type", "location").show()
+-----+-----+-----+-----+-----+
|sensor_type|          ts|location|temperature|temperature_filled|
+-----+-----+-----+-----+-----+
|          1|2017-09-09 12:00:00|  shade|  18.83018|  18.83018|
|          1|2017-09-09 12:00:00|   sun|    null|    null|
|          2|2017-09-09 12:00:00|  shade|  18.61258|  18.61258|
|          2|2017-09-09 12:00:00|   sun|  25.4986|  25.4986|
|          1|2017-09-09 13:00:00|  shade|  18.78458|  18.78458|
|          1|2017-09-09 13:00:00|   sun|  25.68457|  25.68457|
|          2|2017-09-09 13:00:00|  shade|    null|  18.61258|
|          2|2017-09-09 13:00:00|   sun|    null|  25.4986|
|          1|2017-09-09 14:00:00|  shade|  17.98115|  17.98115|
|          1|2017-09-09 14:00:00|   sun|  24.15754|  24.15754|
|          2|2017-09-09 14:00:00|  shade|  18.61258|  18.61258|
|          2|2017-09-09 14:00:00|   sun|    null|  25.4986|
+-----+-----+-----+-----+-----+
>>> # inplace forward fill
>>> filled_df = forward_fill(
...     dataframe,
...     ["sensor_type", "location"],
...     "ts",
...     "temperature"
... )
>>> filled_df.orderBy("ts", "sensor_type", "location").show()
+-----+-----+-----+-----+
|sensor_type|          ts|location|temperature|
+-----+-----+-----+-----+
|          1|2017-09-09 12:00:00|  shade|  18.83018|
|          1|2017-09-09 12:00:00|   sun|    null|
```

(continues on next page)

(continued from previous page)

	2	2017-09-09 12:00:00	shade	18.61258
	2	2017-09-09 12:00:00	sun	25.4986
	1	2017-09-09 13:00:00	shade	18.78458
	1	2017-09-09 13:00:00	sun	25.68457
	2	2017-09-09 13:00:00	shade	18.61258
	2	2017-09-09 13:00:00	sun	25.4986
	1	2017-09-09 14:00:00	shade	17.98115
	1	2017-09-09 14:00:00	sun	24.15754
	2	2017-09-09 14:00:00	shade	18.61258
	2	2017-09-09 14:00:00	sun	25.4986
+-----+-----+-----+-----+-----+				

`butterfree.extract.pre_processing.pivot` (*dataframe*: `pyspark.sql.dataframe.DataFrame`,
group_by_columns: `List[str]`, *pivot_column*:
str, *aggregation*: `Callable`,
mock_value: `Union[float, str] = None`, *mock_type*:
`Union[pyspark.sql.types.DataType, str] =`
`None`, *with_forward_fill*: `bool = False`) →
`pyspark.sql.dataframe.DataFrame`

Defines a pivot transformation.

`butterfree.extract.pre_processing.dataframe`
dataframe to be pivoted.

`butterfree.extract.pre_processing.group_by_columns`
list of columns' names to be grouped.

`butterfree.extract.pre_processing.pivot_column`
column to be pivoted.

`butterfree.extract.pre_processing.agg_column`
column to be aggregated by pivoted category.

`butterfree.extract.pre_processing.aggregation`
desired spark aggregation function to be performed. An example: `spark_agg(col_name)`. See docs for all
`spark_agg`: https://spark.apache.org/docs/2.3.1/api/python/_modules/pyspark/sql/functions.html

`butterfree.extract.pre_processing.mock_value`
value used to make a difference between true nulls resulting from the aggregation and empty values from
the pivot transformation.

`butterfree.extract.pre_processing.mock_type`
`mock_value` data type (compatible with spark).

`butterfree.extract.pre_processing.with_forward_fill`
applies a forward fill to null values after the pivot operation.

Example

```
>>> dataframe.orderBy("ts", "id", "amenity").show()
+----+-----+-----+-----+
| id| ts|amenity| has|
+----+-----+-----+
| 1| 1| fridge|false|
| 1| 1| oven| true|
| 1| 1| pool|false|
| 2| 2| balcony|false|
```

(continues on next page)

(continued from previous page)

```
| 1| 3|balcony| null|
| 1| 4|  oven| null|
| 1| 4|  pool| true|
| 1| 5|balcony| true|
+---+---+-----+-----+
```

```
>>> pivoted = pivot(dataframe, ["id", "ts"], "amenity", "has", functions.first)
>>> pivoted.orderBy("ts", "id").show()
+---+---+-----+-----+-----+-----+
| id| ts|balcony|fridge|oven| pool|
+---+---+-----+-----+-----+-----+
| 1| 1|  null| false|true|false|
| 2| 2|  false|  null|null| null|
| 1| 3|  null|  null|null| null|
| 1| 4|  null|  null|null| true|
| 1| 5|  true|  null|null| null|
+---+---+-----+-----+-----+-----+
```

But, sometimes, you would like to keep the last values that some feature has assumed from previous modifications. In this example, amenity “oven” for the id=1 was set to null and “pool” was set to true at ts=4. All other amenities should then be kept to their actual state at that ts. To do that, we will use a technique called forward fill:

```
>>> pivoted = pivot(
...     dataframe,
...     ["id", "ts"],
...     "amenity",
...     "has",
...     functions.first,
...     with_forward_fill=True
... )
>>> pivoted.orderBy("ts", "id").show()
+---+---+-----+-----+-----+-----+
| id| ts|balcony|fridge|oven| pool|
+---+---+-----+-----+-----+-----+
| 1| 1|  null| false|true|false|
| 2| 2|  false|  null|null| null|
| 1| 3|  null| false|true|false|
| 1| 4|  null| false|true| true|
| 1| 5|  true| false|true| true|
+---+---+-----+-----+-----+-----+
```

Great! Now every amenity that didn’t have been changed kept its state. BUT, the force change to null for amenity “oven” on id=1 at ts=4 was ignored during forward fill. If the user wants to respect this change, it must provide a mock value and type to be used as a signal for “true nulls”. In other words, we want to forward fill only nulls that were created by the pivot transformation.

In this example, amenities only assume boolean values. So there is no mock values for a boolean. It is only true or false. So users can give a mock value of another type (for which the column can be cast to). Check this out:

```
>>> pivoted = pivot(
...     dataframe,
...     ["id", "ts"],
...     "amenity",
...     "has",
...     functions.first,
```

(continues on next page)

(continued from previous page)

```

...     with_forward_fill=True,
...     mock_value=-1,
...     mock_type="int"
... )
>>> pivoted.orderBy("ts", "id").show()
+----+-----+-----+-----+-----+
| id| ts|balcony|fridge|oven| pool|
+----+-----+-----+-----+-----+
| 1| 1|  null| false|true|false|
| 2| 2| false|  null|null|null|
| 1| 3|  null| false|true|false|
| 1| 4|  null| false|null| true|
| 1| 5|  true| false|null| true|
+----+-----+-----+-----+-----+

```

During transformation, this method will cast the `agg_column` to `mock_type` data type and fill all “true nulls” with the `mock_value`. After pivot and forward fill are applied, all new pivoted columns will then return to the original type with all mock values replaced by null.

```

butterfree.extract.pre_processing.replace(dataframe: pyspark.sql.dataframe.DataFrame,
                                         column: str, replace_dict: Dict[str, str]) →
                                         pyspark.sql.dataframe.DataFrame

```

Replace values of a string column in the dataframe using a dict.

Example:

```

>>> from butterfree.extract.pre_processing import replace
... from butterfree.testing.dataframe import (
...     assert_dataframe_equality,
...     create_df_from_collection,
... )
>>> from pyspark import SparkContext
>>> from pyspark.sql import session
>>> spark_context = SparkContext.getOrCreate()
>>> spark_session = session.SparkSession(spark_context)
>>> input_data = [
...     {"id":1, "type": "a"}, {"id":2, "type": "b"}, {"id":3, "type": "c"}
... ]
>>> input_df = create_df_from_collection(input_data, spark_context, spark_session)
>>> input_df.collect()

```

```
[Row(id=1, type='a'), Row(id=2, type='b'), Row(id=3, type='c')]
```

```

>>> replace_dict = {"a": "type_a", "b": "type_b"}
>>> replace(input_df, "type", replace_dict).collect()

```

```
[Row(id=1, type='type_a'), Row(id=2, type='type_b'), Row(id=3, type='c')]
```

Parameters

- **dataframe** – data to be transformed.
- **column** – string column on the dataframe where to apply the replace.
- **replace_dict** – dict with values to be replaced. All mapped values must be string.

Returns Dataframe with column values replaced.

butterfree.extract.readers package

Submodules

FileReader entity.

```
class butterfree.extract.readers.file_reader.FileReader (id: str, path: str, format: str, schema: pyspark.sql.types.StructType = None, format_options: Dict[Any, Any] = None, stream: bool = False)
```

Bases: *butterfree.extract.readers.reader.Reader*

Responsible for get data from files.

id

unique string id for register the reader as a view on the metastore.

path

file location.

format

can be one of the keys: json, parquet, orc, or csv.

schema

an optional pyspark.sql.types.StructType for the input schema.

format_options

additional options required by some formats. Check docs: <https://spark.apache.org/docs/latest/sql-data-sources-load-save-functions.html#manually-specifying-options>

Example

Simple example regarding FileReader class instantiation.

```
>>> from butterfree.extract.readers import FileReader
>>> from butterfree.clients import SparkClient
>>> from butterfree.extract.pre_processing import filter
>>> spark_client = SparkClient()
>>> file_reader = FileReader(
...     id="file_reader_id",
...     path="data_path",
...     format="json"
... )
>>> df = file_reader.consume(spark_client)
```

However, we can define the schema and format_options, like header, and provide them to FileReader.

```
>>> spark_client = SparkClient()
>>> schema_csv = StructType([
...     StructField("column_a", LongType()),
...     StructField("column_b", DoubleType()),
...     StructField("column_c", StringType())
... ])
>>> file_reader = FileReader(
```

(continues on next page)

(continued from previous page)

```

...         id="file_reader_id",
...         path="data_path",
...         format="csv",
...         schema=schema_csv,
...         format_options={
...             "header": True
...         }
...     )
>>> df = file_reader.consume(spark_client)

```

This last method will use the Spark Client, as default, to read the desired file, loading data into a dataframe, according to FileReader class arguments.

It's also possible to define simple transformations within the reader's scope:

```
>>> file_reader.with_(filter, condition="year = 2019").build(spark_client)
```

In this case, however, a temp view will be created, containing the transformed data.

consume (*client*: `butterfree.clients.spark_client.SparkClient`) → `pyspark.sql.dataframe.DataFrame`
Extract data from files stored in defined path.

Try to auto-infer schema if in stream mode and not manually defining a schema.

Parameters `client` – client responsible for connecting to Spark session.

Returns Dataframe with all the files data.

KafkaSource entity.

class `butterfree.extract.readers.kafka_reader.KafkaReader` (*id*: *str*, *topic*: *str*, *value_schema*: *pyspark.sql.types.StructType*, *connection_string*: *str* = *None*, *topic_options*: *Dict[Any, Any]* = *None*, *stream*: *bool* = *True*)

Bases: `butterfree.extract.readers.reader.Reader`

Responsible for get data from a Kafka topic.

id

unique string id for register the reader as a view on the metastore

value_schema

expected schema of the default column named "value" from Kafka.

topic

string with the Kafka topic name to subscribe.

connection_string

string with hosts and ports to connect. The string need to be in the format: host1:port,host2:port,...,hostN:portN. The argument is not necessary if is passed as a environment variable named KAFKA_CONSUMER_CONNECTION_STRING.

topic_options

additional options for consuming from topic. See docs: <https://spark.apache.org/docs/latest/structured-streaming-kafka-integration.html>.

stream

flag to indicate the reading mode: stream or batch

The default df schema coming from Kafka reader of Spark is the following: key:string value:string
topic:string partition:integer offset:long timestamp:timestamp timestampType:integer

But using this reader and passing the desired schema under value_schema we would have the following result:

With value_schema declared as:

```
>>> value_schema = StructType(
...     [
...         StructField("ts", LongType(), nullable=True),
...         StructField("id", LongType(), nullable=True),
...         StructField("type", StringType(), nullable=True),
...     ]
... )
```

The output df schema would be: ts:long id:long type:string kafka_metadata:struct

key:string topic:string value:string partition:integer offset:long timestamp:timestamp timestampType:integer

Instantiation example:

```
>>> from butterfree.extract.readers import KafkaReader
>>> from butterfree.clients import SparkClient
>>> from pyspark.sql.types import StructType, StructField, StringType, LongType
>>> spark_client = SparkClient()
>>> value_schema = StructType(
...     [
...         StructField("ts", LongType(), nullable=True),
...         StructField("id", LongType(), nullable=True),
...         StructField("type", StringType(), nullable=True),
...     ]
... )
>>> kafka_reader = KafkaReader(
...     id="kafka_reader_id",
...     topic="topic",
...     value_schema=value_schema
...     connection_string="host1:port,host2:port",
... )
>>> df = kafka_reader.consume(spark_client)
```

This last method will use the Spark Client, as default, to read the desired topic, loading data into a dataframe, according to KafkaReader class arguments.

In this case, however, a temp view will be created, containing the transformed data.

KAFKA_COLUMNS = ['key', 'topic', 'value', 'partition', 'offset', 'timestamp', 'timestampType']

consume (*client*: butterfree.clients.spark_client.SparkClient) → pyspark.sql.dataframe.DataFrame
Extract data from a kafka topic.

When stream mode it will get all the new data arriving at the topic in a streaming dataframe. When not in stream mode it will get all data available in the kafka topic.

Parameters client – client responsible for connecting to Spark session.

Returns Dataframe with data from topic.

Reader entity.

```
class butterfree.extract.readers.reader.Reader (id: str, incremental_strategy: butterfree.dataframe_service.incremental_strategy.IncrementalStrategy = None)
```

Bases: `abc.ABC`, `butterfree.hooks.hookable_component.HookableComponent`

Abstract base class for Readers.

id

unique string id for register the reader as a view on the metastore.

transformations

list os methods that will be applied over the dataframe after the raw data is extracted.

```
build (client: butterfree.clients.spark_client.SparkClient, columns: List[Any] = None, start_date: str = None, end_date: str = None) → None
```

Register the data got from the reader in the Spark metastore.

Create a temporary view in Spark metastore referencing the data extracted from the target origin after the application of all the defined pre-processing transformations.

The arguments `start_date` and `end_date` are going to be use only when there is a defined *IncrementalStrategy* on the *Reader*.

Parameters

- **client** – client responsible for connecting to Spark session.
- **columns** – list of tuples for selecting/renaming columns on the df.
- **start_date** – lower bound to use in the filter expression.
- **end_date** – upper bound to use in the filter expression.

```
abstract consume (client: butterfree.clients.spark_client.SparkClient) → pyspark.sql.dataframe.DataFrame
```

Extract data from target origin.

Parameters **client** – client responsible for connecting to Spark session.

Returns Dataframe with all the data.

Returns Spark dataframe

```
with_ (transformer: Callable[[...], pyspark.sql.dataframe.DataFrame], *args: Any, **kwargs: Any) → Any
```

Define a new transformation for the Reader.

All the transformations are used when the method `consume` is called.

Parameters

- **transformer** – method that receives a dataframe and output a dataframe.
- ***args** – args for the transformer.
- ****kwargs** – kwargs for the transformer.

Returns Reader object with new transformation

```
with_incremental_strategy (incremental_strategy: butterfree.dataframe_service.incremental_strategy.IncrementalStrategy) → butterfree.extract.readers.reader.Reader
```

Define the incremental strategy for the Reader.

Parameters **incremental_strategy** – definition of the incremental strategy.

Returns Reader with defined incremental strategy.

TableSource entity.

```
class butterfree.extract.readers.table_reader.TableReader (id: str, table: str,  
                                                         database: str = None)
```

Bases: *butterfree.extract.readers.reader.Reader*

Responsible for get data from tables registered in the metastore.

id

unique string id for register the reader as a view on the metastore.

database

name of the metastore database/schema.

table

name of the table.

Example

Simple example regarding TableReader class instantiation.

```
>>> from butterfree.extract.readers import TableReader
>>> from butterfree.clients import SparkClient
>>> from butterfree.extract.pre_processing import filter
>>> spark_client = SparkClient()
>>> table_reader = TableReader(
...     id="table_reader_id",
...     database="table_reader_db",
...     table="table_reader_table"
... )
>>> df = table_reader.consume(spark_client)
```

This last method will use the Spark Client, as default, to read the desired table, loading data into a dataframe, according to TableReader class arguments.

It's also possible to define simple transformations within the reader's scope:

```
>>> table_reader.with_(filter, condition="year = 2019").build(spark_client)
```

In this case, however, a temp view will be created, containing the transformed data.

consume (*client: butterfree.clients.spark_client.SparkClient*) → pyspark.sql.dataframe.DataFrame

Extract data from a table in Spark metastore.

Parameters **client** – client responsible for connecting to Spark session.

Returns Dataframe with all the data from the table.

Module contents

The Reader Component of a Source.

```
class butterfree.extract.readers.FileReader (id: str, path: str, format: str, schema:
                                             pyspark.sql.types.StructType = None, for-
                                             mat_options: Dict[Any, Any] = None,
                                             stream: bool = False)
```

Bases: *butterfree.extract.readers.reader.Reader*

Responsible for get data from files.

id

unique string id for register the reader as a view on the metastore.

path

file location.

format

can be one of the keys: json, parquet, orc, or csv.

schema

an optional pyspark.sql.types.StructType for the input schema.

format_options

additional options required by some formats. Check docs: <https://spark.apache.org/docs/latest/sql-data-sources-load-save-functions.html#manually-specifying-options>

Example

Simple example regarding FileReader class instantiation.

```
>>> from butterfree.extract.readers import FileReader
>>> from butterfree.clients import SparkClient
>>> from butterfree.extract.pre_processing import filter
>>> spark_client = SparkClient()
>>> file_reader = FileReader(
...     id="file_reader_id",
...     path="data_path",
...     format="json"
... )
>>> df = file_reader.consume(spark_client)
```

However, we can define the schema and format_options, like header, and provide them to FileReader.

```
>>> spark_client = SparkClient()
>>> schema_csv = StructType([
...     StructField("column_a", LongType()),
...     StructField("column_b", DoubleType()),
...     StructField("column_c", StringType())
... ])
>>> file_reader = FileReader(
...     id="file_reader_id",
...     path="data_path",
...     format="csv",
...     schema=schema_csv,
...     format_options={
```

(continues on next page)

(continued from previous page)

```

...         "header": True
...     }
... )
>>> df = file_reader.consume(spark_client)

```

This last method will use the Spark Client, as default, to read the desired file, loading data into a dataframe, according to FileReader class arguments.

It's also possible to define simple transformations within the reader's scope:

```
>>> file_reader.with_(filter, condition="year = 2019").build(spark_client)
```

In this case, however, a temp view will be created, containing the transformed data.

consume (*client*: `butterfree.clients.spark_client.SparkClient`) → `pyspark.sql.dataframe.DataFrame`
Extract data from files stored in defined path.

Try to auto-infer schema if in stream mode and not manually defining a schema.

Parameters `client` – client responsible for connecting to Spark session.

Returns Dataframe with all the files data.

```

class butterfree.extract.readers.KafkaReader (id: str, topic: str, value_schema:
                                             pyspark.sql.types.StructType, connection_string: str = None, topic_options:
                                             Dict[Any, Any] = None, stream: bool =
                                             True)

```

Bases: `butterfree.extract.readers.reader.Reader`

Responsible for get data from a Kafka topic.

id

unique string id for register the reader as a view on the metastore

value_schema

expected schema of the default column named “value” from Kafka.

topic

string with the Kafka topic name to subscribe.

connection_string

string with hosts and ports to connect. The string need to be in the format: host1:port,host2:port,...,hostN:portN. The argument is not necessary if is passed as a environment variable named KAFKA_CONSUMER_CONNECTION_STRING.

topic_options

additional options for consuming from topic. See docs: <https://spark.apache.org/docs/latest/structured-streaming-kafka-integration.html>.

stream

flag to indicate the reading mode: stream or batch

The default df schema coming from Kafka reader of Spark is the following: key:string value:string
topic:string partition:integer offset:long timestamp:timestamp timestampType:integer

But using this reader and passing the desired schema under `value_schema` we would have the following result:

With `value_schema` declared as:

```
>>> value_schema = StructType(
...     [
...         StructField("ts", LongType(), nullable=True),
...         StructField("id", LongType(), nullable=True),
...         StructField("type", StringType(), nullable=True),
...     ]
... )
```

The output df schema would be: ts:long id:long type:string kafka_metadata:struct

key:string topic:string value:string partition:integer offset:long timestamp:timestamp timestamp-
Type:integer

Instantiation example:

```
>>> from butterfree.extract.readers import KafkaReader
>>> from butterfree.clients import SparkClient
>>> from pyspark.sql.types import StructType, StructField, StringType, LongType
>>> spark_client = SparkClient()
>>> value_schema = StructType(
...     [
...         StructField("ts", LongType(), nullable=True),
...         StructField("id", LongType(), nullable=True),
...         StructField("type", StringType(), nullable=True),
...     ]
... )
>>> kafka_reader = KafkaReader(
...     id="kafka_reader_id",
...     topic="topic",
...     value_schema=value_schema
...     connection_string="host1:port,host2:port",
... )
>>> df = kafka_reader.consume(spark_client)
```

This last method will use the Spark Client, as default, to read the desired topic, loading data into a dataframe, according to KafkaReader class arguments.

In this case, however, a temp view will be created, containing the transformed data.

KAFKA_COLUMNS = ['key', 'topic', 'value', 'partition', 'offset', 'timestamp', 'timestamp-
Type']

consume (*client*: butterfree.clients.spark_client.SparkClient) → pyspark.sql.dataframe.DataFrame
Extract data from a kafka topic.

When stream mode it will get all the new data arriving at the topic in a streaming dataframe. When not in stream mode it will get all data available in the kafka topic.

Parameters client – client responsible for connecting to Spark session.

Returns Dataframe with data from topic.

class butterfree.extract.readers.**TableReader** (*id*: str, *table*: str, *database*: str = None)

Bases: *butterfree.extract.readers.reader.Reader*

Responsible for get data from tables registered in the metastore.

id

unique string id for register the reader as a view on the metastore.

database

name of the metastore database/schema.

table

name of the table.

Example

Simple example regarding TableReader class instantiation.

```
>>> from butterfree.extract.readers import TableReader
>>> from butterfree.clients import SparkClient
>>> from butterfree.extract.pre_processing import filter
>>> spark_client = SparkClient()
>>> table_reader = TableReader(
...     id="table_reader_id",
...     database="table_reader_db",
...     table="table_reader_table"
... )
>>> df = table_reader.consume(spark_client)
```

This last method will use the Spark Client, as default, to read the desired table, loading data into a dataframe, according to TableReader class arguments.

It's also possible to define simple transformations within the reader's scope:

```
>>> table_reader.with_(filter, condition="year = 2019").build(spark_client)
```

In this case, however, a temp view will be created, containing the transformed data.

consume (*client*: `butterfree.clients.spark_client.SparkClient`) → `pyspark.sql.dataframe.DataFrame`
Extract data from a table in Spark metastore.

Parameters **client** – client responsible for connecting to Spark session.

Returns Dataframe with all the data from the table.

Submodules

Holds the SourceSelector class.

class `butterfree.extract.source.Source` (*readers*: `List[butterfree.extract.readers.reader.Reader]`,
query: `str`)

Bases: `butterfree.hooks.hookable_component.HookableComponent`

The definition of the the entry point data for the ETL pipeline.

A FeatureSet (the next step in the pipeline) expects a single dataframe as input. This dataframe is built from a data composition of one or more readers defined in the Source. There is only one Source for pipeline.

TODO refactor query into multiple query components TODO make it harder to do query injection

readers

list of readers from where the source will get data.

query

Spark SQL query to run against the readers.

Example

Simple example regarding Source class instantiation.

```
>>> from butterfree.extract import Source
>>> from butterfree.extract.readers import TableReader, FileReader
>>> from butterfree.clients import SparkClient
>>> spark_client = SparkClient()
>>> source = Source(
...     readers=[
...         TableReader(
...             id="table_reader_id",
...             database="table_reader_db",
...             table="table_reader_table",
...         ),
...         FileReader(id="file_reader_id", path="data_sample_path", format="json
↪"),
...     ],
...     query=f"select a.*, b.feature2 "
...     f"from table_reader_id a "
...     f"inner join file_reader_id b on a.id = b.id ",
... )
>>> df = source.construct(spark_client)
```

This last method will use the Spark Client, as default, to create temporary views regarding each reader and, after, will run the desired query and return a dataframe.

construct (*client*: `butterfree.clients.spark_client.SparkClient`, *start_date*: *str* = `None`, *end_date*: *str* = `None`) → `pyspark.sql.dataframe.DataFrame`
Construct an entry point dataframe for a feature set.

This method will assemble multiple readers, by building each one and querying them using a Spark SQL. It's important to highlight that in order to filter a dataframe regarding date boundaries, it's important to define a `IncrementalStrategy`, otherwise your data will not be filtered. Besides, both start and end dates parameters are optional.

After that, there's the caching of the dataframe, however since `cache()` in Spark is lazy, an action is triggered in order to force persistence.

Parameters

- **client** – client responsible for connecting to Spark session.
- **start_date** – user defined start date for filtering.
- **end_date** – user defined end date for filtering.

Returns `DataFrame` with the query result against all readers.

Module contents

The Source Component of a Feature Set.

class `butterfree.extract.Source` (*readers: List[butterfree.extract.readers.reader.Reader]*, *query: str*)
 Bases: `butterfree.hooks.hookable_component.HookableComponent`

The definition of the the entry point data for the ETL pipeline.

A FeatureSet (the next step in the pipeline) expects a single dataframe as input. This dataframe is built from a data composition of one or more readers defined in the Source. There is only one Source for pipeline.

TODO refactor query into multiple query components TODO make it harder to do query injection

readers

list of readers from where the source will get data.

query

Spark SQL query to run against the readers.

Example

Simple example regarding Source class instantiation.

```
>>> from butterfree.extract import Source
>>> from butterfree.extract.readers import TableReader, FileReader
>>> from butterfree.clients import SparkClient
>>> spark_client = SparkClient()
>>> source = Source(
...     readers=[
...         TableReader(
...             id="table_reader_id",
...             database="table_reader_db",
...             table="table_reader_table",
...         ),
...         FileReader(id="file_reader_id", path="data_sample_path", format="json
↵"),
...     ],
...     query=f"select a.*, b.feature2 "
...     f"from table_reader_id a "
...     f"inner join file_reader_id b on a.id = b.id ",
... )
>>> df = source.construct(spark_client)
```

This last method will use the Spark Client, as default, to create temporary views regarding each reader and, after, will run the desired query and return a dataframe.

construct (*client: butterfree.clients.spark_client.SparkClient*, *start_date: str = None*, *end_date: str = None*) → `pyspark.sql.dataframe.DataFrame`
 Construct an entry point dataframe for a feature set.

This method will assemble multiple readers, by building each one and querying them using a Spark SQL. It's important to highlight that in order to filter a dataframe regarding date boundaries, it's important to define a `IncrementalStrategy`, otherwise your data will not be filtered. Besides, both start and end dates parameters are optional.

After that, there's the caching of the dataframe, however since `cache()` in Spark is lazy, an action is triggered in order to force persistence.

Parameters

- **client** – client responsible for connecting to Spark session.
- **start_date** – user defined start date for filtering.
- **end_date** – user defined end date for filtering.

Returns DataFrame with the query result against all readers.

butterfree.hooks package**Subpackages****butterfree.hooks.schema_compatibility package****Submodules**

Cassandra table schema compatibility Hook definition.

class `butterfree.hooks.schema_compatibility.cassandra_table_schema_compatibility_hook.CassandraTableSchemaCompatibilityHook`

Bases: `butterfree.hooks.hook.Hook`

Hook to verify the schema compatibility with a Cassandra's table.

Verifies if all columns presented on the dataframe exists and are the same type on the target Cassandra's table.

cassandra_client

client to connect to Cassandra DB.

table

table name.

run (*dataframe: pyspark.sql.dataframe.DataFrame*) → `pyspark.sql.dataframe.DataFrame`

Check the schema compatibility from a given Dataframe.

This method does not change anything on the Dataframe.

Parameters **dataframe** – dataframe to verify schema compatibility.

Returns unchanged dataframe.

Raises **ValueError** if the schemas are incompatible. –

Spark table schema compatibility Hook definition.

class butterfree.hooks.schema_compatibility.spark_table_schema_compatibility_hook.**SparkTableSchemaCompatibilityHook**

Bases: *butterfree.hooks.hook.Hook*

Hook to verify the schema compatibility with a Spark's table.

Verifies if all columns presented on the dataframe exists and are the same type on the target Spark's table.

spark_client

client to connect to Spark's metastore.

table

table name.

database

database name.

run (*dataframe: pyspark.sql.dataframe.DataFrame*) → pyspark.sql.dataframe.DataFrame

Check the schema compatibility from a given Dataframe.

This method does not change anything on the Dataframe.

Parameters dataframe – dataframe to verify schema compatibility.

Returns unchanged dataframe.

Raises ValueError if the schemas are incompatible. –

Module contents

Holds Schema Compatibility Hooks definitions.

class butterfree.hooks.schema_compatibility.**CassandraTableSchemaCompatibilityHook** (*cassandra_client*, *table*, *database*)

but-
ter-
free.clients.c
ta-
ble:
str)

Bases: *butterfree.hooks.hook.Hook*

Hook to verify the schema compatibility with a Cassandra's table.

Verifies if all columns presented on the dataframe exists and are the same type on the target Cassandra's table.

cassandra_client

client to connect to Cassandra DB.

table

table name.

run (*dataframe: pyspark.sql.dataframe.DataFrame*) → *pyspark.sql.dataframe.DataFrame*
 Check the schema compatibility from a given Dataframe.

This method does not change anything on the Dataframe.

Parameters *dataframe* – dataframe to verify schema compatibility.

Returns unchanged dataframe.

Raises **ValueError** if the schemas are incompatible. –

```
class butterfree.hooks.schema_compatibility.SparkTableSchemaCompatibilityHook (spark_client:
                                                                    but-
                                                                    ter-
                                                                    free.clients.spark_
                                                                    ta-
                                                                    ble:
                                                                    str,
                                                                    database:
                                                                    str
                                                                    =
                                                                    None)
```

Bases: *butterfree.hooks.hook.Hook*

Hook to verify the schema compatibility with a Spark’s table.

Verifies if all columns presented on the dataframe exists and are the same type on the target Spark’s table.

spark_client

client to connect to Spark’s metastore.

table

table name.

database

database name.

run (*dataframe: pyspark.sql.dataframe.DataFrame*) → *pyspark.sql.dataframe.DataFrame*
 Check the schema compatibility from a given Dataframe.

This method does not change anything on the Dataframe.

Parameters *dataframe* – dataframe to verify schema compatibility.

Returns unchanged dataframe.

Raises **ValueError** if the schemas are incompatible. –

Submodules

Hook abstract class entity.

```
class butterfree.hooks.hook.Hook
```

Bases: *abc.ABC*

Definition of a hook function to call on a Dataframe.

abstract run (*dataframe: pyspark.sql.dataframe.DataFrame*) → *pyspark.sql.dataframe.DataFrame*
 Run interface for Hook.

Parameters *dataframe* – dataframe to use in the Hook.

Returns dataframe result from the Hook.

Definition of hookable component.

class `butterfree.hooks.hookable_component.HookableComponent`

Bases: `object`

Defines a component with the ability to hold pre and post hook functions.

All main module of Butterfree have a common object that enables their integration: dataframes. Spark's dataframe is the glue that enables the transmission of data between the main modules. Hooks have a simple interface, they are functions that accepts a dataframe and outputs a dataframe. These Hooks can be triggered before or after the main execution of a component.

Components from Butterfree that inherit HookableComponent entity, are components that can define a series of steps to occur before or after the execution of their main functionality.

pre_hooks

function steps to trigger before component main functionality.

post_hooks

function steps to trigger after component main functionality.

enable_pre_hooks

property to indicate if the component can define pre_hooks.

enable_post_hooks

property to indicate if the component can define post_hooks.

add_post_hook (**hooks:* `butterfree.hooks.hook.Hook`) → *butter-*
free.hooks.hookable_component.HookableComponent

Add a post-hook steps to the component.

Parameters **hooks** – Hook steps to add to post_hook list.

Returns Component with the Hook inserted in post_hook list.

Raises **ValueError** – if the component does not accept post-hooks.

add_pre_hook (**hooks:* `butterfree.hooks.hook.Hook`) → *butter-*
free.hooks.hookable_component.HookableComponent

Add a pre-hook steps to the component.

Parameters **hooks** – Hook steps to add to pre_hook list.

Returns Component with the Hook inserted in pre_hook list.

Raises **ValueError** – if the component does not accept pre-hooks.

property enable_post_hooks

Property to indicate if the component can define post_hooks.

property enable_pre_hooks

Property to indicate if the component can define pre_hooks.

property post_hooks

Function steps to trigger after component main functionality.

property pre_hooks

Function steps to trigger before component main functionality.

run_post_hooks (*dataframe:* `pyspark.sql.dataframe.DataFrame`) → *pys-*
`park.sql.dataframe.DataFrame`

Run all defined post-hook steps from a given dataframe.

Parameters **dataframe** – data to input in the defined post-hook steps.

Returns dataframe after passing for all defined post-hooks.

run_pre_hooks (*dataframe:* *pyspark.sql.dataframe.DataFrame*) → *pyspark.sql.dataframe.DataFrame*
 Run all defined pre-hook steps from a given dataframe.

Parameters **dataframe** – data to input in the defined pre-hook steps.

Returns dataframe after passing for all defined pre-hooks.

Module contents

Holds Hooks definitions.

class `butterfree.hooks.Hook`

Bases: `abc.ABC`

Definition of a hook function to call on a Dataframe.

abstract **run** (*dataframe:* *pyspark.sql.dataframe.DataFrame*) → *pyspark.sql.dataframe.DataFrame*
 Run interface for Hook.

Parameters **dataframe** – dataframe to use in the Hook.

Returns dataframe result from the Hook.

class `butterfree.hooks.HookableComponent`

Bases: `object`

Defines a component with the ability to hold pre and post hook functions.

All main module of Butterfree have a common object that enables their integration: dataframes. Spark's dataframe is the glue that enables the transmission of data between the main modules. Hooks have a simple interface, they are functions that accepts a dataframe and outputs a dataframe. These Hooks can be triggered before or after the main execution of a component.

Components from Butterfree that inherit HookableComponent entity, are components that can define a series of steps to occur before or after the execution of their main functionality.

pre_hooks

function steps to trigger before component main functionality.

post_hooks

function steps to trigger after component main functionality.

enable_pre_hooks

property to indicate if the component can define pre_hooks.

enable_post_hooks

property to indicate if the component can define post_hooks.

add_post_hook (**hooks:* *butterfree.hooks.hook.Hook*) → *butterfree.hooks.hookable_component.HookableComponent*
 Add a post-hook steps to the component.

Parameters **hooks** – Hook steps to add to post_hook list.

Returns Component with the Hook inserted in post_hook list.

Raises **ValueError** – if the component does not accept post-hooks.

add_pre_hook (**hooks:* *butterfree.hooks.hook.Hook*) → *butterfree.hooks.hookable_component.HookableComponent*
 Add a pre-hook steps to the component.

Parameters **hooks** – Hook steps to add to pre_hook list.

Returns Component with the Hook inserted in pre_hook list.

Raises ValueError – if the component does not accept pre-hooks.

property enable_post_hooks

Property to indicate if the component can define post_hooks.

property enable_pre_hooks

Property to indicate if the component can define pre_hooks.

property post_hooks

Function steps to trigger after component main functionality.

property pre_hooks

Function steps to trigger before component main functionality.

run_post_hooks (*dataframe:* *pyspark.sql.dataframe.DataFrame*) → pys-
park.sql.dataframe.DataFrame

Run all defined post-hook steps from a given dataframe.

Parameters dataframe – data to input in the defined post-hook steps.

Returns dataframe after passing for all defined post-hooks.

run_pre_hooks (*dataframe:* *pyspark.sql.dataframe.DataFrame*) → pys-
park.sql.dataframe.DataFrame

Run all defined pre-hook steps from a given dataframe.

Parameters dataframe – data to input in the defined pre-hook steps.

Returns dataframe after passing for all defined pre-hooks.

butterfree.load package

Subpackages

butterfree.load.processing package

Submodules

Json conversion for writers.

butterfree.load.processing.json_transform.**json_transform** (*dataframe:* *pys-*
park.sql.dataframe.DataFrame)
→ *pys-*
park.sql.dataframe.DataFrame

Filters DataFrame's rows using the given condition and value.

Parameters dataframe – Spark DataFrame.

Returns Converted dataframe.

Module contents

Pre Processing Components regarding Readers.

`butterfree.load.processing.json_transform` (*dataframe: pyspark.sql.dataframe.DataFrame*)
→ `pyspark.sql.dataframe.DataFrame`

Filters DataFrame's rows using the given condition and value.

Parameters `dataframe` – Spark DataFrame.

Returns Converted dataframe.

`butterfree.load.writers` package

Submodules

Holds the Historical Feature Store writer class.

```
class butterfree.load.writers.historical_feature_store_writer.HistoricalFeatureStoreWriter
```

Bases: *butterfree.load.writers.writer.Writer*

Enable writing feature sets into the Historical Feature Store.

db_config

Datalake configuration for Spark, by default on AWS S3. For more information check module ‘butterfree.db.configs’.

database

database name to use in Spark metastore. By default FEATURE_STORE_HISTORICAL_DATABASE environment variable.

num_partitions

value to use when applying repartition on the df before save.

validation_threshold

lower and upper tolerance to using in count validation. The default value is defined in DEFAULT_VALIDATION_THRESHOLD property. For example: with a validation_threshold = 0.01 and a given calculated count on the dataframe equal to 100000 records, if the feature store return a count equal

to 995000 an error will not be thrown. Use `validation_threshold = 0` to not use tolerance in the validation.

debug_mode

“dry run” mode, write the result to a temporary view.

Example

Simple example regarding `HistoricalFeatureStoreWriter` class instantiation. We can instantiate this class without db configurations, so the class get the `S3Config()` where it provides default configurations about AWS S3 service.

```
>>> spark_client = SparkClient()
>>> writer = HistoricalFeatureStoreWriter()
>>> writer.write(feature_set=feature_set,
...             dataframe=dataframe,
...             spark_client=spark_client)
```

However, we can define the db configurations, like write mode, file format and S3 bucket, and provide them to `HistoricalFeatureStoreWriter`.

```
>>> spark_client = SparkClient()
>>> config = MetastoreConfig(path="my_s3_bucket_name",
...                          mode="overwrite",
...                          format_="parquet")
>>> writer = HistoricalFeatureStoreWriter(db_config=config)
>>> writer.write(feature_set=feature_set,
...             dataframe=dataframe,
...             spark_client=spark_client)
```

For what settings you can use on `S3Config` and default settings, to read `S3Config` class.

We can write with interval mode, where `HistoricalFeatureStoreWrite` will need to use Dynamic Partition Inserts, the behaviour of `OVERWRITE` keyword is controlled by `spark.sql.sources.partitionOverwriteMode` configuration property. The dynamic overwrite mode is enabled Spark will only delete the partitions for which it has data to be written to. All the other partitions remain intact.

```
>>> spark_client = SparkClient()
>>> writer = HistoricalFeatureStoreWriter(interval_mode=True)
>>> writer.write(feature_set=feature_set,
...             dataframe=dataframe,
...             spark_client=spark_client)
```

We can instantiate `HistoricalFeatureStoreWriter` class to validate the df to be written.

```
>>> spark_client = SparkClient()
>>> writer = HistoricalFeatureStoreWriter()
>>> writer.validate(feature_set=feature_set,
...                dataframe=dataframe,
...                spark_client=spark_client)
```

Both methods (write and validate) will need the Spark Client, Feature Set and DataFrame, to write or to validate, according to the Writer’s arguments.

P.S.: When writing, the `HistoricalFeatureStoreWrite` partitions the data to improve queries performance. The data is stored in partition folders in AWS S3 based on time (per year, month and day).

`DEFAULT_VALIDATION_THRESHOLD = 0.01`

`PARTITION_BY = ['year', 'month', 'day']`

`check_schema` (*client*: Any, *dataframe*: `pyspark.sql.dataframe.DataFrame`, *table_name*: str, *database*: str = None) → `pyspark.sql.dataframe.DataFrame`

Instantiate the schema check hook to check schema between dataframe and database.

Parameters

- **client** – client for Spark or Cassandra connections with external services.
- **dataframe** – Spark dataframe containing data from a feature set.
- **table_name** – table name where the dataframe will be saved.
- **database** – database name where the dataframe will be saved.

`validate` (*feature_set*: `butterfree.transform.feature_set.FeatureSet`, *dataframe*: `pyspark.sql.dataframe.DataFrame`, *spark_client*: `butterfree.clients.spark_client.SparkClient`) → None

Calculate dataframe rows to validate data into Feature Store.

Parameters

- **feature_set** – object processed with feature_set informations.
- **dataframe** – spark dataframe containing data from a feature set.
- **spark_client** – client for spark connections with external services.

Raises `AssertionError` – if count of written data doesn't match count in current feature set dataframe.

`write` (*feature_set*: `butterfree.transform.feature_set.FeatureSet`, *dataframe*: `pyspark.sql.dataframe.DataFrame`, *spark_client*: `butterfree.clients.spark_client.SparkClient`) → None

Loads the data from a feature set into the Historical Feature Store.

Parameters

- **feature_set** – object processed with feature_set informations.
- **dataframe** – spark dataframe containing data from a feature set.
- **spark_client** – client for spark connections with external services.

If the `debug_mode` is set to `True`, a temporary table with a name in the format: `historical_feature_store__{feature_set.name}` will be created instead of writing to the real historical feature store.

Holds the Online Feature Store writer class.

```

class butterfree.load.writers.online_feature_store_writer.OnlineFeatureStoreWriter (db_config:
    but-
    ter-
    free.config:
    =
    None,
    database:
    str
    =
    None,
    de-
    bug_mode:
    bool
    =
    False,
    write_to_e-
    bool
    =
    False,
    in-
    ter-
    val_mode:
    bool
    =
    False,
    check_sche-
    but-
    ter-
    free.hooks.
    =
    None)

```

Bases: *butterfree.load.writers.writer.Writer*

Enable writing feature sets into the Online Feature Store.

db_config

Spark configuration for connect databases. For more information check the module ‘butterfree.db.configs’.

debug_mode

“dry run” mode, write the result to a temporary view.

write_to_entity

option to write the data to the entity table. With this option set to True, the writer will write the feature set to a table with the name equal to the entity name, defined on the pipeline. So, it WILL NOT write to a table with the name of the feature set, as it normally does.

Example

Simple example regarding OnlineFeatureStoreWriter class instantiation. We can instantiate this class without db configurations, so the class get the CassandraConfig() where it provides default configurations about CassandraDB.

```
>>> spark_client = SparkClient()
>>> writer = OnlineFeatureStoreWriter()
>>> writer.write(feature_set=feature_set,
...             dataframe=dataframe,
...             spark_client=spark_client)
```

However, we can define the db configurations and provide them to OnlineFeatureStoreWriter.

```
>>> spark_client = SparkClient()
>>> config = CassandraConfig(mode="overwrite",
...                          format_="parquet",
...                          keyspace="keyspace_name")
```

```
>>> writer = OnlineFeatureStoreWriter(db_config=config)
>>> writer.write(feature_set=feature_set,
...             dataframe=dataframe,
...             spark_client=spark_client)
```

For what settings you can use on CassandraConfig and default settings, to read CassandraConfig class.

We can instantiate OnlineFeatureStoreWriter class to validate the writers, using the default or custom configs.

```
>>> spark_client = SparkClient()
>>> writer = OnlineFeatureStoreWriter()
>>> writer.validate(feature_set=feature_set,
...                dataframe=dataframe,
...                spark_client=spark_client)
```

Both methods (writer and validate) will need the Spark Client, Feature Set and DataFrame, to write or to validate, according to OnlineFeatureStoreWriter class arguments.

There's an important aspect to be highlighted here: if you're using the incremental mode, we do not check if your data is the newest before writing to the online feature store.

This behavior is known and will be fixed soon.

check_schema (*client: Any, dataframe: pyspark.sql.dataframe.DataFrame, table_name: str, database: str = None*) → *pyspark.sql.dataframe.DataFrame*
 Instantiate the schema check hook to check schema between dataframe and database.

Parameters

- **client** – client for Spark or Cassandra connections with external services.
- **dataframe** – Spark dataframe containing data from a feature set.
- **table_name** – table name where the dataframe will be saved.
- **database** – database name where the dataframe will be saved.

static filter_latest (*dataframe*: *pyspark.sql.dataframe.DataFrame*, *id_columns*: *List[Any]*)
 → *pyspark.sql.dataframe.DataFrame*
 Filters latest data from the dataframe.

Parameters

- **dataframe** – spark dataframe containing data from a feature set.
- **id_columns** – unique identifier column set for this feature set.

Returns

contains only latest data for each unique id in the feature set.

Return type dataframe

get_db_schema (*feature_set*: *butterfree.transform.feature_set.FeatureSet*) → *List[Dict[Any, Any]]*
 Get desired database schema.

Parameters **feature_set** – object processed with feature set metadata.

Returns Desired database schema.

validate (*feature_set*: *butterfree.transform.feature_set.FeatureSet*, *dataframe*: *pyspark.sql.dataframe.DataFrame*, *spark_client*: *butterfree.clients.spark_client.SparkClient*)
 → *None*
 Calculate dataframe rows to validate data into Feature Store.

Parameters

- **feature_set** – object processed with feature set metadata.
- **dataframe** – Spark dataframe containing data from a feature set.
- **spark_client** – client for Spark connections with external services.

Raises **AssertionError** – if validation fails.

write (*feature_set*: *butterfree.transform.feature_set.FeatureSet*, *dataframe*: *pyspark.sql.dataframe.DataFrame*, *spark_client*: *butterfree.clients.spark_client.SparkClient*)
 → *Optional[pyspark.sql.streaming.StreamingQuery]*
 Loads the latest data from a feature set into the Feature Store.

Parameters

- **feature_set** – object processed with feature set metadata.
- **dataframe** – Spark dataframe containing data from a feature set.
- **spark_client** – client for Spark connections with external services.

Returns Streaming handler if writing streaming df, None otherwise.

If the `debug_mode` is set to `True`, a temporary table with a name in the format: `online_feature_store__my_feature_set` will be created instead of writing to the real online feature store. If dataframe is streaming this temporary table will be updated in real time.

Writer entity.

class `butterfree.load.writers.writer.Writer` (*db_config*: *butterfree.configs.db.abstract_config.AbstractWriteConfig*,
debug_mode: *bool = False*, *interval_mode*:
bool = False, *write_to_entity*: *bool = False*)

Bases: `abc.ABC`, `butterfree.hooks.hookable_component.HookableComponent`

Abstract base class for Writers.

Parameters **spark_client** – client for spark connections with external services.

abstract check_schema (*client: Any, dataframe: pyspark.sql.dataframe.DataFrame, table_name: str, database: str = None*) → *pyspark.sql.dataframe.DataFrame*
Instantiate the schema check hook to check schema between dataframe and database.

Parameters

- **client** – client for Spark or Cassandra connections with external services.
- **dataframe** – Spark dataframe containing data from a feature set.
- **table_name** – table name where the dataframe will be saved.
- **database** – database name where the dataframe will be saved.

abstract validate (*feature_set: butterfree.transform.feature_set.FeatureSet, dataframe: pyspark.sql.dataframe.DataFrame, spark_client: butterfree.clients.spark_client.SparkClient*) → *Any*
Calculate dataframe rows to validate data into Feature Store.

Parameters

- **feature_set** – object processed with feature set metadata.
- **dataframe** – Spark dataframe containing data from a feature set.
- **spark_client** – client for Spark connections with external services.

Raises AssertionError – if validation fails.

with_ (*transformer: Callable[[...], pyspark.sql.dataframe.DataFrame], *args: Any, **kwargs: Any*) → *butterfree.load.writers.writer.Writer*
Define a new transformation for the Writer.

All the transformations are used when the method consume is called.

Parameters

- **transformer** – method that receives a dataframe and output a dataframe.
- ***args** – args for the transformer.
- ****kwargs** – kwargs for the transformer.

Returns Reader object with new transformation

abstract write (*feature_set: butterfree.transform.feature_set.FeatureSet, dataframe: pyspark.sql.dataframe.DataFrame, spark_client: butterfree.clients.spark_client.SparkClient*) → *Any*
Loads the data from a feature set into the Feature Store.

Feature Store could be Online or Historical.

Parameters

- **feature_set** – object processed with feature set metadata.
- **dataframe** – Spark dataframe containing data from a feature set.
- **spark_client** – client for Spark connections with external services.

Module contents

Holds data loaders for historical and online feature store.

```
class butterfree.load.writers.HistoricalFeatureStoreWriter (db_config: butter-
free.configs.db.abstract_config.AbstractWriteC
= None, database:
str = None,
num_partitions:
int = None, vali-
dation_threshold:
float = 0.01, de-
bug_mode: bool =
False, interval_mode:
bool = False,
check_schema_hook:
butter-
free.hooks.hook.Hook
= None)
```

Bases: `butterfree.load.writers.writer.Writer`

Enable writing feature sets into the Historical Feature Store.

db_config

Datalake configuration for Spark, by default on AWS S3. For more information check module 'butterfree.db.configs'.

database

database name to use in Spark metastore. By default `FEATURE_STORE_HISTORICAL_DATABASE` environment variable.

num_partitions

value to use when applying repartition on the df before save.

validation_threshold

lower and upper tolerance to using in count validation. The default value is defined in `DEFAULT_VALIDATION_THRESHOLD` property. For example: with a `validation_threshold = 0.01` and a given calculated count on the dataframe equal to 100000 records, if the feature store return a count equal to 995000 an error will not be thrown. Use `validation_threshold = 0` to not use tolerance in the validation.

debug_mode

"dry run" mode, write the result to a temporary view.

Example

Simple example regarding `HistoricalFeatureStoreWriter` class instantiation. We can instantiate this class without db configurations, so the class get the `S3Config()` where it provides default configurations about AWS S3 service.

```
>>> spark_client = SparkClient()
>>> writer = HistoricalFeatureStoreWriter()
>>> writer.write(feature_set=feature_set,
...             dataframe=dataframe,
...             spark_client=spark_client)
```

However, we can define the db configurations, like write mode, file format and S3 bucket, and provide them to `HistoricalFeatureStoreWriter`.

```

>>> spark_client = SparkClient()
>>> config = MetastoreConfig(path="my_s3_bucket_name",
...                          mode="overwrite",
...                          format_"parquet")
>>> writer = HistoricalFeatureStoreWriter(db_config=config)
>>> writer.write(feature_set=feature_set,
...              dataframe=dataframe,
...              spark_client=spark_client)

```

For what settings you can use on S3Config and default settings, to read S3Config class.

We can write with interval mode, where HistoricalFeatureStoreWrite will need to use Dynamic Partition Inserts, the behaviour of OVERWRITE keyword is controlled by spark.sql.sources.partitionOverwriteMode configuration property. The dynamic overwrite mode is enabled Spark will only delete the partitions for which it has data to be written to. All the other partitions remain intact.

```

>>> spark_client = SparkClient()
>>> writer = HistoricalFeatureStoreWriter(interval_mode=True)
>>> writer.write(feature_set=feature_set,
...              dataframe=dataframe,
...              spark_client=spark_client)

```

We can instantiate HistoricalFeatureStoreWriter class to validate the df to be written.

```

>>> spark_client = SparkClient()
>>> writer = HistoricalFeatureStoreWriter()
>>> writer.validate(feature_set=feature_set,
...                 dataframe=dataframe,
...                 spark_client=spark_client)

```

Both methods (write and validate) will need the Spark Client, Feature Set and DataFrame, to write or to validate, according to the Writer's arguments.

P.S.: When writing, the HistoricalFeatureStoreWrite partitions the data to improve queries performance. The data is stored in partition folders in AWS S3 based on time (per year, month and day).

DEFAULT_VALIDATION_THRESHOLD = 0.01

PARTITION_BY = ['year', 'month', 'day']

check_schema (*client: Any, dataframe: pyspark.sql.dataframe.DataFrame, table_name: str, database: str = None*) → pyspark.sql.dataframe.DataFrame

Instantiate the schema check hook to check schema between dataframe and database.

Parameters

- **client** – client for Spark or Cassandra connections with external services.
- **dataframe** – Spark dataframe containing data from a feature set.
- **table_name** – table name where the dataframe will be saved.
- **database** – database name where the dataframe will be saved.

validate (*feature_set*: `butterfree.transform.feature_set.FeatureSet`, *dataframe*: `pyspark.sql.dataframe.DataFrame`, *spark_client*: `butterfree.clients.spark_client.SparkClient`)
 → None

Calculate dataframe rows to validate data into Feature Store.

Parameters

- **feature_set** – object processed with feature_set informations.
- **dataframe** – spark dataframe containing data from a feature set.
- **spark_client** – client for spark connections with external services.

Raises AssertionError – if count of written data doesn't match count in current feature set dataframe.

write (*feature_set*: `butterfree.transform.feature_set.FeatureSet`, *dataframe*: `pyspark.sql.dataframe.DataFrame`, *spark_client*: `butterfree.clients.spark_client.SparkClient`)
 → None

Loads the data from a feature set into the Historical Feature Store.

Parameters

- **feature_set** – object processed with feature_set informations.
- **dataframe** – spark dataframe containing data from a feature set.
- **spark_client** – client for spark connections with external services.

If the `debug_mode` is set to `True`, a temporary table with a name in the format: `historical_feature_store__{feature_set.name}` will be created instead of writing to the real historical feature store.

class `butterfree.load.writers.OnlineFeatureStoreWriter` (*db_config*: `butterfree.configs.db.abstract_config.AbstractWriteConfig`
 = None, *database*: `str` = None, *debug_mode*: `bool` = False, *write_to_entity*: `bool` = False, *interval_mode*: `bool` = False, *check_schema_hook*: `butterfree.hooks.hook.Hook` = None)

Bases: `butterfree.load.writers.writer.Writer`

Enable writing feature sets into the Online Feature Store.

db_config

Spark configuration for connect databases. For more information check the module 'butterfree.db.configs'.

debug_mode

“dry run” mode, write the result to a temporary view.

write_to_entity

option to write the data to the entity table. With this option set to `True`, the writer will write the feature set to a table with the name equal to the entity name, defined on the pipeline. So, it WILL NOT write to a table with the name of the feature set, as it normally does.

Example

Simple example regarding OnlineFeatureStoreWriter class instantiation. We can instantiate this class without db configurations, so the class get the CassandraConfig() where it provides default configurations about CassandraDB.

```
>>> spark_client = SparkClient()
>>> writer = OnlineFeatureStoreWriter()
>>> writer.write(feature_set=feature_set,
...             dataframe=dataframe,
...             spark_client=spark_client)
```

However, we can define the db configurations and provide them to OnlineFeatureStoreWriter.

```
>>> spark_client = SparkClient()
>>> config = CassandraConfig(mode="overwrite",
...                          format_="parquet",
...                          keyspace="keyspace_name")
```

```
>>> writer = OnlineFeatureStoreWriter(db_config=config)
>>> writer.write(feature_set=feature_set,
...             dataframe=dataframe,
...             spark_client=spark_client)
```

For what settings you can use on CassandraConfig and default settings, to read CassandraConfig class.

We can instantiate OnlineFeatureStoreWriter class to validate the writers, using the default or custom configs.

```
>>> spark_client = SparkClient()
>>> writer = OnlineFeatureStoreWriter()
>>> writer.validate(feature_set=feature_set,
...                dataframe=dataframe,
...                spark_client=spark_client)
```

Both methods (writer and validate) will need the Spark Client, Feature Set and DataFrame, to write or to validate, according to OnlineFeatureStoreWriter class arguments.

There's an important aspect to be highlighted here: if you're using the incremental mode, we do not check if your data is the newest before writing to the online feature store.

This behavior is known and will be fixed soon.

check_schema (*client: Any, dataframe: pyspark.sql.dataframe.DataFrame, table_name: str, database: str = None*) → pyspark.sql.dataframe.DataFrame
Instantiate the schema check hook to check schema between dataframe and database.

Parameters

- **client** – client for Spark or Cassandra connections with external services.
- **dataframe** – Spark dataframe containing data from a feature set.
- **table_name** – table name where the dataframe will be saved.
- **database** – database name where the dataframe will be saved.

static filter_latest (*dataframe*: *pyspark.sql.dataframe.DataFrame*, *id_columns*: *List[Any]*)
 → *pyspark.sql.dataframe.DataFrame*

Filters latest data from the dataframe.

Parameters

- **dataframe** – spark dataframe containing data from a feature set.
- **id_columns** – unique identifier column set for this feature set.

Returns

contains only latest data for each unique id in the feature set.

Return type dataframe

get_db_schema (*feature_set*: *butterfree.transform.feature_set.FeatureSet*) → *List[Dict[Any, Any]]*

Get desired database schema.

Parameters **feature_set** – object processed with feature set metadata.

Returns Desired database schema.

validate (*feature_set*: *butterfree.transform.feature_set.FeatureSet*, *dataframe*: *pyspark.sql.dataframe.DataFrame*, *spark_client*: *butterfree.clients.spark_client.SparkClient*)
 → *None*

Calculate dataframe rows to validate data into Feature Store.

Parameters

- **feature_set** – object processed with feature set metadata.
- **dataframe** – Spark dataframe containing data from a feature set.
- **spark_client** – client for Spark connections with external services.

Raises **AssertionError** – if validation fails.

write (*feature_set*: *butterfree.transform.feature_set.FeatureSet*, *dataframe*: *pyspark.sql.dataframe.DataFrame*, *spark_client*: *butterfree.clients.spark_client.SparkClient*)
 → *Optional[pyspark.sql.streaming.StreamingQuery]*

Loads the latest data from a feature set into the Feature Store.

Parameters

- **feature_set** – object processed with feature set metadata.
- **dataframe** – Spark dataframe containing data from a feature set.
- **spark_client** – client for Spark connections with external services.

Returns Streaming handler if writing streaming df, None otherwise.

If the `debug_mode` is set to `True`, a temporary table with a name in the format: `online_feature_store__my_feature_set` will be created instead of writing to the real online feature store. If dataframe is streaming this temporary table will be updated in real time.

Submodules

Holds the Sink class.

```
class butterfree.load.sink.Sink (writers: List[butterfree.load.writers.writer.Writer], validation: Optional[butterfree.validations.validation.Validation] = None)
```

Bases: *butterfree.hooks.hookable_component.HookableComponent*

Define the destinations for the feature set pipeline.

A Sink is created from a set of writers. The main goal of the Sink is to trigger the load in each defined writers. After the load the entity can be used to make sure that all data was written properly using the validate method.

writers

list of Writers to use to load the data.

validation

validation to check the data before starting to write.

```
flush (feature_set: butterfree.transform.feature_set.FeatureSet, dataframe: pyspark.sql.dataframe.DataFrame, spark_client: butterfree.clients.spark_client.SparkClient)
→ List[pyspark.sql.streaming.StreamingQuery]
```

Trigger a write job in all the defined Writers.

Parameters

- **dataframe** – spark dataframe containing data from a feature set.
- **feature_set** – object processed with feature set metadata.
- **spark_client** – client used to run a query.

Returns Streaming handlers for each defined writer, if writing streaming dfs.

```
validate (feature_set: butterfree.transform.feature_set.FeatureSet, dataframe: pyspark.sql.dataframe.DataFrame, spark_client: butterfree.clients.spark_client.SparkClient)
→ None
```

Trigger a validation job in all the defined Writers.

Parameters

- **dataframe** – spark dataframe containing data from a feature set.
- **feature_set** – object processed with feature set metadata.
- **spark_client** – client used to run a query.

Raises **RuntimeError** – if any on the Writers returns a failed validation.

property validation

Validation to check the data before starting to write.

property writers

List of Writers to use to load the data.

Module contents

Holds the Sink component of a feature set pipeline.

class `butterfree.load.Sink` (*writers*: `List[butterfree.load.writers.writer.Writer]`, *validation*: `Optional[butterfree.validations.validation.Validation] = None`)

Bases: `butterfree.hooks.hookable_component.HookableComponent`

Define the destinations for the feature set pipeline.

A Sink is created from a set of writers. The main goal of the Sink is to trigger the load in each defined writers. After the load the entity can be used to make sure that all data was written properly using the validate method.

writers

list of Writers to use to load the data.

validation

validation to check the data before starting to write.

flush (*feature_set*: `butterfree.transform.feature_set.FeatureSet`, *dataframe*: `pyspark.sql.dataframe.DataFrame`, *spark_client*: `butterfree.clients.spark_client.SparkClient`)

→ `List[pyspark.sql.streaming.StreamingQuery]`

Trigger a write job in all the defined Writers.

Parameters

- **dataframe** – spark dataframe containing data from a feature set.
- **feature_set** – object processed with feature set metadata.
- **spark_client** – client used to run a query.

Returns Streaming handlers for each defined writer, if writing streaming dfs.

validate (*feature_set*: `butterfree.transform.feature_set.FeatureSet`, *dataframe*: `pyspark.sql.dataframe.DataFrame`, *spark_client*: `butterfree.clients.spark_client.SparkClient`)

→ `None`

Trigger a validation job in all the defined Writers.

Parameters

- **dataframe** – spark dataframe containing data from a feature set.
- **feature_set** – object processed with feature set metadata.
- **spark_client** – client used to run a query.

Raises `RuntimeError` – if any on the Writers returns a failed validation.

property validation

Validation to check the data before starting to write.

property writers

List of Writers to use to load the data.

butterfree.migrations package

Subpackages

butterfree.migrations.database_migration package

Submodules

Cassandra Migration entity.

class butterfree.migrations.database_migration.cassandra_migration.**CassandraMigration**

Bases: `butterfree.migrations.database_migration.database_migration.DatabaseMigration`

Cassandra class for performing migrations.

This class implements some methods of the parent DatabaseMigration class and has specific methods for query building.

The CassandraMigration class will be used, as the name suggests, for applying changes to a given Cassandra table. There are, however, some remarks that need to be highlighted:

- If an existing feature has its type changed, then it's extremely important to

make sure that this conversion would not result in data loss; - If new features are added to your feature set, then they're going to be added to the corresponding Cassandra table; - Since feature sets can be written both to a feature set and an entity table, we're not going to automatically drop features when using entity tables, since it means that some features belong to a different feature set. In summary, if data is being loaded into an entity table, then users can drop columns manually.

Migration entity.

class butterfree.migrations.database_migration.database_migration.**DatabaseMigration** (*client:*

but-
ter-
free.clien

Bases: `abc.ABC`

Abstract base class for Migrations.

apply_migration (*feature_set:* `butterfree.transform.feature_set.FeatureSet`, *writer:* `butterfree.load.writers.writer.Writer`, *debug_mode:* `bool`) → None

Apply the migration in the respective database.

Parameters

- **feature_set** – the feature set.
- **writer** – the writer being used to load the feature set.
- **debug_mode** – if active, it brings up the queries generated.

create_query (*fs_schema:* `List[Dict[str, Any]]`, *table_name:* `str`, *db_schema:* `List[Dict[str, Any]]`) = `None`, *write_on_entity:* `bool = None`) → Any

Create a query regarding a data source.

Returns The desired queries for the given database.


```

class butterfree.migrations.database_migration.database_migration.Diff (column:
                                                                    str,
                                                                    kind:
                                                                    but-
                                                                    ter-
                                                                    free.migrations.database_mig
                                                                    value:
                                                                    Any)

```

Bases: object

DataClass to help identifying different types of diff between schemas.

```

class Kind (value)

```

Bases: enum.Enum

Mapping actions to take given a difference between columns of a schema.

```

ADD = 1

```

```

ALTER_KEY = 2

```

```

ALTER_TYPE = 3

```

```

DROP = 4

```

```

column: str

```

```

kind: Kind

```

```

value: Any

```

Metastore Migration entity.

```

class butterfree.migrations.database_migration.metastore_migration.MetastoreMigration (database_migration:
                                                                    str
                                                                    =
                                                                    None)

```

Bases: `butterfree.migrations.database_migration.database_migration.DatabaseMigration`

MetastoreMigration class for performing migrations.

This class implements some methods of the parent DatabaseMigration class and has specific methods for query building. The MetastoreMigration class will be used, as the name suggests, for applying changes to a given Metastore table. There are, however, some remarks that need to be highlighted:

- If an existing feature has its type changed, then it's extremely important to

make sure that this conversion would not result in data loss; - If new features are added to your feature set, then they're going to be added to the corresponding Metastore table; - Since feature sets can be written both to a feature set and an entity table, we're not going to automatically drop features when using entity tables, since it means that some features belong to a different feature set. In summary, if data is being loaded into an entity table, then users can drop columns manually.

Module contents

Holds available database migrations.

class `butterfree.migrations.database_migration.CassandraMigration`

Bases: `butterfree.migrations.database_migration.database_migration.DatabaseMigration`

Cassandra class for performing migrations.

This class implements some methods of the parent `DatabaseMigration` class and has specific methods for query building.

The `CassandraMigration` class will be used, as the name suggests, for applying changes to a given Cassandra table. There are, however, some remarks that need to be highlighted:

- If an existing feature has its type changed, then it's extremely important to

make sure that this conversion would not result in data loss; - If new features are added to your feature set, then they're going to be added to the corresponding Cassandra table; - Since feature sets can be written both to a feature set and an entity table, we're not going to automatically drop features when using entity tables, since it means that some features belong to a different feature set. In summary, if data is being loaded into an entity table, then users can drop columns manually.

class `butterfree.migrations.database_migration.Diff` (*column: str, kind: butterfree.migrations.database_migration.database_migration.D*
value: Any)

Bases: `object`

`DataClass` to help identifying different types of diff between schemas.

class `Kind` (*value*)

Bases: `enum.Enum`

Mapping actions to take given a difference between columns of a schema.

ADD = 1

ALTER_KEY = 2

ALTER_TYPE = 3

DROP = 4

column: str

kind: Kind

value: Any

class `butterfree.migrations.database_migration.MetastoreMigration` (*database: str = None*)

Bases: `butterfree.migrations.database_migration.database_migration.DatabaseMigration`

`MetastoreMigration` class for performing migrations.

This class implements some methods of the parent `DatabaseMigration` class and has specific methods for query building. The `MetastoreMigration` class will be used, as the name suggests, for applying changes to a given `Metastore` table. There are, however, some remarks that need to be highlighted:

- If an existing feature has its type changed, then it's extremely important to

make sure that this conversion would not result in data loss; - If new features are added to your feature set, then they're going to be added to the corresponding `Metastore` table; - Since feature sets can be

written both to a feature set and an entity table, we're not going to automatically drop features when using entity tables, since it means that some features belong to a different feature set. In summary, if data is being loaded into an entity table, then users can drop columns manually.

Module contents

Holds available migrations.

butterfree.pipelines package

Submodules

FeatureSetPipeline entity.

```
class butterfree.pipelines.feature_set_pipeline.FeatureSetPipeline (source:
    butter-
    free.extract.source.Source,
    fea-
    ture_set:
    butter-
    free.transform.feature_set.FeatureS
    sink:
    butter-
    free.load.sink.Sink,
    spark_client:
    butter-
    free.clients.spark_client.SparkClient
    = None)
```

Bases: object

Defines a ETL pipeline for the construction of a feature set.

source

source of the data, the entry point of the pipeline.

feature_set

feature set composed by features and context metadata.

sink

sink used to write the output dataframe in the desired locations.

spark_client

client used to access Spark connection.

Example

This an example regarding the feature set pipeline definition. All sources, feature set (and its features) and writers are defined.

```
>>> import os
```

```
>>> from butterfree.pipelines import FeatureSetPipeline
>>> from butterfree.constants.columns import TIMESTAMP_COLUMN
>>> from butterfree.configs.db import MetastoreConfig
```

(continues on next page)

(continued from previous page)

```

>>> from butterfree.extract import Source
>>> from butterfree.extract.readers import TableReader
>>> from butterfree.transform import FeatureSet
>>> from butterfree.transform.features import (
...     Feature,
...     KeyFeature,
...     TimestampFeature,
... )
>>> from butterfree.transform.transformations import (
...     SparkFunctionTransform,
...     CustomTransform,
... )
>>> from butterfree.load import Sink
>>> from butterfree.load.writers import HistoricalFeatureStoreWriter
>>> from pyspark.sql import functions

```

```

>>> def divide(df, fs, column1, column2):
...     name = fs.get_output_columns()[0]
...     df = df.withColumn(name,
...         functions.col(column1) / functions.col(column2))
...     return df

```

```

>>> pipeline = FeatureSetPipeline(
...     source=Source(
...         readers=[
...             TableReader(
...                 id="table_reader_id",
...                 database="table_reader_db",
...                 table="table_reader_table",
...             ),
...         ],
...         query=f"select * from table_reader_id ",
...     ),
...     feature_set=FeatureSet(
...         name="feature_set",
...         entity="entity",
...         description="description",
...         features=[
...             Feature(
...                 name="feature1",
...                 description="test",
...                 transformation=SparkFunctionTransform(
...                     functions=[Function(functions.avg, DataType.DOUBLE),
...                         Function(functions.stddev_pop, DataType.DOUBLE)],
...                 ).with_window(
...                     partition_by="id",
...                     order_by=TIMESTAMP_COLUMN,
...                     mode="fixed_windows",
...                     window_definition=["2 minutes", "15 minutes"],
...                 ),
...             ),
...             Feature(
...                 name="divided_feature",
...                 description="unit test",
...                 transformation=CustomTransform(
...                     transformer=divide,

```

(continues on next page)

(continued from previous page)

```

...         column1="feature1",
...         column2="feature2",
...     ),
... ],
... keys=[
...     KeyFeature(
...         name="id",
...         description="The user's Main ID or device ID"
...     )
... ],
...     timestamp=TimestampFeature(),
... ),
... sink=Sink(
...     writers=[
...         HistoricalFeatureStoreWriter(
...             db_config=MetastoreConfig(
...                 format_="parquet",
...                 path=os.path.join(
...                     os.path.dirname(os.path.abspath(__file__))
...                 ),
...             ),
...         )
...     ],
... ),
... )
... )
... )
... )

```

```
>>> pipeline.run()
```

This last method (run) will execute the pipeline flow, it'll read from the defined sources, compute all the transformations and save the data to the specified locations.

We can run the pipeline over a range of dates by passing an end-date and a start-date, where it will only bring data within this date range.

```
>>> pipeline.run(end_date="2020-08-04", start_date="2020-07-04")
```

Or run up to a date, where it will only bring data up to the specific date.

```
>>> pipeline.run(end_date="2020-08-04")
```

Or just a specific date, where you will only bring data for that day.

```
>>> pipeline.run_for_date(execution_date="2020-08-04")
```

property feature_set

Feature set composed by features and context metadata.

run (*end_date: str = None, partition_by: List[str] = None, order_by: List[str] = None, num_processors: int = None, start_date: str = None*) → None
Runs the defined feature set pipeline.

The pipeline consists in the following steps: - Constructs the input dataframe from the data source. - Construct the feature set dataframe using the defined Features. - Load the data to the configured sink locations.

It's important to notice, however, that both parameters `partition_by` and `num_processors` are WIP, we intend to enhance their functionality soon. Use only if strictly necessary.

run_for_date (*execution_date: str = None, partition_by: List[str] = None, order_by: List[str] = None, num_processors: int = None*) → None
Runs the defined feature set pipeline for a specific date.

The pipeline consists in the following steps:

- Constructs the input dataframe from the data source.
- Construct the feature set dataframe using the defined Features.
- Load the data to the configured sink locations.

It's important to notice, however, that both parameters `partition_by` and `num_processors` are WIP, we intend to enhance their functionality soon. Use only if strictly necessary.

property sink

Sink used to write the output dataframe in the desired locations.

property source

Source of the data, the entry point of the pipeline.

property spark_client

Client used to access Spark connection.

Module contents

ETL Pipelines.

```
class butterfree.pipelines.FeatureSetPipeline (source: butterfree.extract.source.Source,  
feature_set: butterfree.transform.feature_set.FeatureSet,  
sink: butterfree.load.sink.Sink,  
spark_client: butterfree.clients.spark_client.SparkClient  
= None)
```

Bases: object

Defines a ETL pipeline for the construction of a feature set.

source

source of the data, the entry point of the pipeline.

feature_set

feature set composed by features and context metadata.

sink

sink used to write the output dataframe in the desired locations.

spark_client

client used to access Spark connection.

Example

This an example regarding the feature set pipeline definition. All sources, feature set (and its features) and writers are defined.

```
>>> import os
```

```
>>> from butterfree.pipelines import FeatureSetPipeline
>>> from butterfree.constants.columns import TIMESTAMP_COLUMNN
>>> from butterfree.configs.db import MetastoreConfig
>>> from butterfree.extract import Source
>>> from butterfree.extract.readers import TableReader
>>> from butterfree.transform import FeatureSet
>>> from butterfree.transform.features import (
...     Feature,
...     KeyFeature,
...     TimestampFeature,
... )
>>> from butterfree.transform.transformations import (
...     SparkFunctionTransform,
...     CustomTransform,
... )
>>> from butterfree.load import Sink
>>> from butterfree.load.writers import HistoricalFeatureStoreWriter
>>> from pyspark.sql import functions
```

```
>>> def divide(df, fs, column1, column2):
...     name = fs.get_output_columns()[0]
...     df = df.withColumn(name,
...         functions.col(column1) / functions.col(column2))
...     return df
```

```
>>> pipeline = FeatureSetPipeline(
...     source=Source(
...         readers=[
...             TableReader(
...                 id="table_reader_id",
...                 database="table_reader_db",
...                 table="table_reader_table",
...             ),
...         ],
...         query=f"select * from table_reader_id ",
...     ),
...     feature_set=FeatureSet(
...         name="feature_set",
...         entity="entity",
...         description="description",
...         features=[
...             Feature(
...                 name="feature1",
...                 description="test",
...                 transformation=SparkFunctionTransform(
...                     functions=[Function(functions.avg, DataType.DOUBLE),
...                         Function(functions.stddev_pop, DataType.DOUBLE)],
...                 ).with_window(
...                     partition_by="id",
...                     order_by=TIMESTAMP_COLUMNN,
...                 )
...             )
...         ]
...     )
... )
```

(continues on next page)

(continued from previous page)

```

...         mode="fixed_windows",
...         window_definition=["2 minutes", "15 minutes"],
...     ),
... ),
...     Feature(
...         name="divided_feature",
...         description="unit test",
...         transformation=CustomTransform(
...             transformer=divide,
...             column1="feature1",
...             column2="feature2",
...         ),
...     ),
... ],
...     keys=[
...         KeyFeature(
...             name="id",
...             description="The user's Main ID or device ID"
...         )
...     ],
...     timestamp=TimestampFeature(),
... ),
...     sink=Sink(
...         writers=[
...             HistoricalFeatureStoreWriter(
...                 db_config=MetastoreConfig(
...                     format_="parquet",
...                     path=os.path.join(
...                         os.path.dirname(os.path.abspath(__file__))
...                     ),
...                 ),
...             ),
...         ],
...     ),
... ],
... ),
... )

```

```
>>> pipeline.run()
```

This last method (run) will execute the pipeline flow, it'll read from the defined sources, compute all the transformations and save the data to the specified locations.

We can run the pipeline over a range of dates by passing an end-date and a start-date, where it will only bring data within this date range.

```
>>> pipeline.run(end_date="2020-08-04", start_date="2020-07-04")
```

Or run up to a date, where it will only bring data up to the specific date.

```
>>> pipeline.run(end_date="2020-08-04")
```

Or just a specific date, where you will only bring data for that day.

```
>>> pipeline.run_for_date(execution_date="2020-08-04")
```

property feature_set

Feature set composed by features and context metadata.

run (*end_date: str = None, partition_by: List[str] = None, order_by: List[str] = None, num_processors: int = None, start_date: str = None*) → None
Runs the defined feature set pipeline.

The pipeline consists in the following steps: - Constructs the input dataframe from the data source. - Construct the feature set dataframe using the defined Features. - Load the data to the configured sink locations.

It's important to notice, however, that both parameters `partition_by` and `num_processors` are WIP, we intend to enhance their functionality soon. Use only if strictly necessary.

run_for_date (*execution_date: str = None, partition_by: List[str] = None, order_by: List[str] = None, num_processors: int = None*) → None
Runs the defined feature set pipeline for a specific date.

The pipeline consists in the following steps:

- Constructs the input dataframe from the data source.
- Construct the feature set dataframe using the defined Features.
- Load the data to the configured sink locations.

It's important to notice, however, that both parameters `partition_by` and `num_processors` are WIP, we intend to enhance their functionality soon. Use only if strictly necessary.

property sink

Sink used to write the output dataframe in the desired locations.

property source

Source of the data, the entry point of the pipeline.

property spark_client

Client used to access Spark connection.

butterfree.reports package

Submodules

Write feature set metadata.

class `butterfree.reports.metadata.Metadata` (*pipeline: butterfree.pipelines.feature_set_pipeline.FeatureSetPipeline, save: bool = False*)

Bases: `object`

Generate metadata for feature set pipeline.

pipeline

object processed with feature set pipeline.

save

bool value with default is False. When this value is True, it will generate a file.

Example

```
>>> pipeline = FeatureSetPipeline()
>>> metadata = Metadata(pipeline)
>>> metadata.to_json()
```

```
“feature_set”: “feature_set”, “description”: “description”, “source”: [
  { “reader”: “Table Reader”, “location”: “db.table”
  }, {
    “reader”: “File Reader”, “location”: “path”
  }
], “sink”: [
  { “writer”: “Historical Feature Store Writer”
  }, {
    “writer”: “Online Feature Store Writer”
  }
], “features”: [
  {
    “column”: { “name”: “user_id”, “data_type”: “IntegerType”
    }, “description”: “The user’s Main ID or device ID”
  }, {
    “column”: { “name”: “timestamp”, “data_type”: “TimestampType”
    }, “description”: “Time tag for the state of all features.”
  }, {
    “column”: {
      “name”: “listing_page_viewed__rent_per_month__avg_over_7_days_fixed_windows”,
      “data_type”: “FloatType”
    }, “description”: “Average of something.”
  }, {
    “column”: {
      “name”: “listing_page_viewed__rent_per_month__avg_over_2_weeks_fixed_windows”,
      “data_type”: “FloatType”
    }, “description”: “Average of something.”
  }
]
]
```

`to_json()` → Any
Generate json file.

`to_markdown()` → Any
Generate markdown file.

Module contents

Reports module.

class `butterfree.reports.Metadata` (*pipeline: butterfree.pipelines.feature_set_pipeline.FeatureSetPipeline, save: bool = False*)

Bases: object

Generate metadata for feature set pipeline.

pipeline

object processed with feature set pipeline.

save

bool value with default is False. When this value is True, it will generate a file.

Example

```
>>> pipeline = FeatureSetPipeline()
>>> metadata = Metadata(pipeline)
>>> metadata.to_json()
```

```
“feature_set”: “feature_set”, “description”: “description”, “source”: [
  { “reader”: “Table Reader”, “location”: “db.table”
  }, {
    “reader”: “File Reader”, “location”: “path”
  }
], “sink”: [
  { “writer”: “Historical Feature Store Writer”
  }, {
    “writer”: “Online Feature Store Writer”
  }
], “features”: [
  {
    “column”: { “name”: “user_id”, “data_type”: “IntegerType”
    }, “description”: “The user’s Main ID or device ID”
  }, {
    “column”: { “name”: “timestamp”, “data_type”: “TimestampType”
    }, “description”: “Time tag for the state of all features.”
  }, {
    “column”: {
```

```
        "name": "listing_page_viewed__rent_per_month__avg_over_7_days_fixed_windows",
        "data_type": "FloatType"
    }, {"description": "Average of something."
    }, {
        "column": {
            "name": "listing_page_viewed__rent_per_month__avg_over_2_weeks_fixed_windows",
            "data_type": "FloatType"
        }, {"description": "Average of something."
        }
    }
]
```

`to_json()` → Any
Generate json file.

`to_markdown()` → Any
Generate markdown file.

butterfree.testing package

Subpackages

butterfree.testing.dataframe package

Module contents

Methods to assert properties regarding Apache Spark Dataframes.

`butterfree.testing.dataframe.assert_column_equality` (*output_df*: *pyspark.sql.dataframe.DataFrame*,
target_df: *pyspark.sql.dataframe.DataFrame*,
output_column: *pyspark.sql.column.Column*,
target_column: *pyspark.sql.column.Column*)
→ None

Columns comparison method.

`butterfree.testing.dataframe.assert_dataframe_equality` (*output_df*: *pyspark.sql.dataframe.DataFrame*,
target_df: *pyspark.sql.dataframe.DataFrame*)
→ None

Dataframe comparison method.

```
butterfree.testing.dataframe.create_df_from_collection (data: List[Dict[Any, Any]], spark_context: pyspark.context.SparkContext, spark_session: pyspark.sql.session.SparkSession, schema: pyspark.sql.types.StructType = None) → pyspark.sql.dataframe.DataFrame
```

Creates a dataframe from a list of dicts.

Module contents

Utilities to make testing of Butterfree tools easier.

butterfree.transform package

Subpackages

butterfree.transform.features package

Submodules

Feature entity.

```
class butterfree.transform.features.feature.Feature (name: str, description: str, dtype: butterfree.constants.data_type.DataType = None, from_column: str = None, transformation: butterfree.transform.transformations.transform_component.TransformComponent = None)
```

Bases: object

Defines a Feature.

A Feature is the result of a transformation over one (or more) data columns over an input dataframe. Transformations can be as simple as renaming, casting types, mathematical expressions or complex functions/models.

name

feature name. Can be use by the transformation to derive multiple output columns.

description

brief explanation regarding the feature.

dtype

data type for the output columns of this feature.

from_column

original column to build feature. Used when there is transformation or the transformation has no reference about the column to use for.

transformation

transformation that will be applied to create this feature.

property dtype

Attribute dtype getter.

property from_column

Attribute from_column getter.

get_output_columns () → List[str]

Get output columns that will be generated by this feature engineering.

Returns Output columns names.

transform (dataframe: pyspark.sql.dataframe.DataFrame) → pyspark.sql.dataframe.DataFrame

Performs a transformation to the feature pipeline.

Parameters dataframe – input dataframe for the transformation.

Returns Transformed dataframe.

property transformation

Attribute transformation getter.

KeyFeature entity.

```
class butterfree.transform.features.key_feature.KeyFeature (name: str, de-
description: str,
dtype: butter-
free.constants.data_type.DataType,
from_column: str =
None, transfor-
mation: butter-
free.transform.transformations.transform_comp
= None)
```

Bases: *butterfree.transform.features.feature.Feature*

Defines a KeyFeature.

A FeatureSet must contain one or more KeyFeatures, which will be used as keys when storing the feature set dataframe as tables. The Feature Set may validate keys are unique for the latest state of a feature set.

name

key name. Can be use by the transformation to derive multiple key columns.

description

brief explanation regarding the key.

dtype

data type for the output column of this key.

from_column

original column to build a key. Used when there is transformation or the transformation has no reference about the column to use for.

transformation

transformation that will be applied to create this key. Keys can be derived by transformations over any data column. Like a location hash based on latitude and longitude.

TimestampFeature entity.

```

class butterfree.transform.features.timestamp_feature.TimestampFeature (from_column:
                                                                    str
                                                                    =
                                                                    None,
                                                                    trans-
                                                                    for-
                                                                    ma-
                                                                    tion:
                                                                    but-
                                                                    ter-
                                                                    free.transform.transformation
                                                                    =
                                                                    None,
                                                                    from_ms:
                                                                    bool
                                                                    =
                                                                    False,
                                                                    mask:
                                                                    str
                                                                    =
                                                                    None)

```

Bases: *butterfree.transform.features.feature.Feature*

Defines a TimestampFeature.

A FeatureSet must contain one TimestampFeature, which will be used as a time tag for the state of all features. By containing a timestamp feature, users may time travel over their features. The Feature Set may validate that the set of keys and timestamp are unique for a feature set.

By defining a TimestampColumn, the feature set will always contain a data column called “timestamp” of TimestampType (spark dtype).

from_column

original column to build a “timestamp” feature column. Used when there is transformation or the transformation has no reference about the column to use for. If from_column is None, the FeatureSet will assume the input dataframe already has a data column called “timestamp”.

transformation

transformation that will be applied to create the “timestamp”. Type casting will already happen when no transformation is given. But a timestamp can be derived from multiple columns, like year, month and day, for example. The transformation must always handle naming and typing.

from_ms

true if timestamp column presents milliseconds time unit. A

conversion is then performed.

mask

specified timestamp format by the user.

transform (*dataframe: pyspark.sql.dataframe.DataFrame*) → pyspark.sql.dataframe.DataFrame

Performs a transformation to the feature pipeline.

Parameters dataframe – input dataframe for the transformation.

Returns Transformed dataframe.

Module contents

Holds all feature types to be part of a FeatureSet.

```
class butterfree.transform.features.Feature (name: str, description: str, dtype: butterfree.constants.data_type.DataType  
                                             = None, from_column: str =  
                                             None, transformation: butterfree.transform.transformations.transform_component.TransformComponent = None)
```

Bases: object

Defines a Feature.

A Feature is the result of a transformation over one (or more) data columns over an input dataframe. Transformations can be as simple as renaming, casting types, mathematical expressions or complex functions/models.

name

feature name. Can be use by the transformation to derive multiple output columns.

description

brief explanation regarding the feature.

dtype

data type for the output columns of this feature.

from_column

original column to build feature. Used when there is transformation or the transformation has no reference about the column to use for.

transformation

transformation that will be applied to create this feature.

property dtype

Attribute dtype getter.

property from_column

Attribute from_column getter.

get_output_columns () → List[str]

Get output columns that will be generated by this feature engineering.

Returns Output columns names.

transform (*dataframe: pyspark.sql.dataframe.DataFrame*) → pyspark.sql.dataframe.DataFrame

Performs a transformation to the feature pipeline.

Parameters **dataframe** – input dataframe for the transformation.

Returns Transformed dataframe.

property transformation

Attribute transformation getter.

```
class butterfree.transform.features.KeyFeature (name: str, description: str, dtype: butterfree.constants.data_type.DataType,  
                                             from_column: str = None,  
                                             transformation: butterfree.transform.transformations.transform_component.TransformComponent = None)
```

Bases: *butterfree.transform.features.feature.Feature*

Defines a KeyFeature.

A FeatureSet must contain one or more KeyFeatures, which will be used as keys when storing the feature set dataframe as tables. The Feature Set may validate keys are unique for the latest state of a feature set.

name

key name. Can be use by the transformation to derive multiple key columns.

description

brief explanation regarding the key.

dtype

data type for the output column of this key.

from_column

original column to build a key. Used when there is transformation or the transformation has no reference about the column to use for.

transformation

transformation that will be applied to create this key. Keys can be derived by transformations over any data column. Like a location hash based on latitude and longitude.

```
class butterfree.transform.features.TimestampFeature (from_column: str = None,
                                                    transformation: butter-
free.transform.transformations.transform_component.Tr
                                                    = None, from_ms: bool =
False, mask: str = None)
```

Bases: *butterfree.transform.features.feature.Feature*

Defines a TimestampFeature.

A FeatureSet must contain one TimestampFeature, which will be used as a time tag for the state of all features. By containing a timestamp feature, users may time travel over their features. The Feature Set may validate that the set of keys and timestamp are unique for a feature set.

By defining a TimestampColumn, the feature set will always contain a data column called “timestamp” of TimestampType (spark dtype).

from_column

original column to build a “timestamp” feature column. Used when there is transformation or the transformation has no reference about the column to use for. If from_column is None, the FeatureSet will assume the input dataframe already has a data column called “timestamp”.

transformation

transformation that will be applied to create the “timestamp”. Type casting will already happen when no transformation is given. But a timestamp can be derived from multiple columns, like year, month and day, for example. The transformation must always handle naming and typing.

from_ms

true if timestamp column presents milliseconds time unit. A

conversion is then performed.

mask

specified timestamp format by the user.

transform (*dataframe: pyspark.sql.dataframe.DataFrame*) → *pyspark.sql.dataframe.DataFrame*

Performs a transformation to the feature pipeline.

Parameters *dataframe* – input dataframe for the transformation.

Returns Transformed dataframe.

butterfree.transform.transformations package

Subpackages

butterfree.transform.transformations.user_defined_functions package

Submodules

Module contents

Submodules

Aggregated Transform entity.

```
class butterfree.transform.transformations.aggregated_transform.AggregatedTransform (functions.  
List[butte  
fil-  
ter_expre  
str  
=  
None)
```

Bases: `butterfree.transform.transformations.transform_component.TransformComponent`

Specifies an aggregation.

This transformation needs to be used within an `AggregatedFeatureSet`. Unlike the other transformations, this class won't have a transform method implemented.

The idea behind aggregating is that, in spark, we should execute all aggregation functions after a single `groupby`. So an `AggregateFeatureSet` will have many `Features` with `AggregatedTransform`. If each one of them needs to apply a `groupby.agg()`, then we must join all the results in the end, making this computation extremely slow.

Now, the `AggregateFeatureSet` will collect all `Features'` `AggregatedTransform` definitions and run, at once, a `groupby.agg(*aggregations)`.

This class helps defining on a feature, which aggregation function will be applied to build a new aggregated column. Allowed aggregations are registered under the

`allowed_aggregations` property.

functions

namedtuple with aggregation function and data type.

filter_expression

sql boolean expression to be used inside `agg` function. The filter expression can be used to aggregate some column only with records that obey certain condition. Has the same behaviour of the following SQL expression: `agg(case when filter_expression then col end)`

Example

```

>>> from butterfree.transform.transformations import AggregatedTransform
>>> from butterfree.transform.features import Feature
>>> from butterfree.constants import DataType
>>> from butterfree.transform.utils import Function
>>> import pyspark.sql.functions as F
>>> feature = Feature(
...     name="feature",
...     description="aggregated transform",
...     transformation=AggregatedTransform(
...         functions=[
...             Function(F.avg, DataType.DOUBLE),
...             Function(F.stddev_pop, DataType.DOUBLE)],
...     ),
...     from_column="somenumber",
...)
>>> feature.get_output_columns()
['feature__avg', 'feature__stddev_pop']
>>> feature.transform(anydf)
NotImplementedError: ...

```

property aggregations

Aggregated spark columns.

property output_columns

Columns names generated by the transformation.

transform (*dataframe: pyspark.sql.dataframe.DataFrame*) → pyspark.sql.dataframe.DataFrame
(NotImplemented) Performs a transformation to the feature pipeline.

For the AggregatedTransform, the transformation won't be applied without using an AggregatedFeature-Set.

Parameters dataframe – input dataframe.

Raises NotImplementedError. –

CustomTransform entity.

class butterfree.transform.transformations.custom_transform.**CustomTransform** (*transformer: Callable[[...], Any], **kwargs: Any*)

Bases: *butterfree.transform.transformations.transform_component.TransformComponent*

Defines a Custom Transform.

transformer

function to use for transforming the dataframe

**kwargs

kwargs for the transformer

Example

It's necessary to instantiate the CustomTransform class using a custom method that must always receive a dataframe and the parent feature as arguments and the custom arguments must be passed to the builder through `*kwargs`.

```
>>> from butterfree.transform.transformations import CustomTransform
>>> from butterfree.transform.features import Feature
>>> from butterfree.constants import DataType
>>> from pyspark import SparkContext
>>> from pyspark.sql import session
>>> import pyspark.sql.functions as F
>>> sc = SparkContext.getOrCreate()
>>> spark = session.SparkSession(sc)
>>> df = spark.createDataFrame([(1, "2016-04-11 11:31:11", 200, 200),
...                             (1, "2016-04-11 11:44:12", 300, 300),
...                             (1, "2016-04-11 11:46:24", 400, 400),
...                             (1, "2016-04-11 12:03:21", 500, 500)]
...                             ).toDF("id", "timestamp", "feature1", "feature2")
>>> def divide(df, parent_feature, column1, column2):
...     name = parent_feature.get_output_columns()[0]
...     df = df.withColumn(name, F.col(column1) / F.col(column2))
...     return df
>>> feature = Feature(
...     name="feature",
...     description="custom transform usage example",
...     dtype=DataType.DOUBLE,
...     transformation=CustomTransform(
...         transformer=divide, column1="feature1", column2="feature2",
...     )
... )
>>> feature.transform(df).orderBy("timestamp").show()
+-----+-----+-----+-----+-----+
|feature1|feature2| id|          timestamp|feature|
+-----+-----+-----+-----+-----+
|      200|      200| 1|2016-04-11 11:31:11|      1.0|
|      300|      300| 1|2016-04-11 11:44:12|      1.0|
|      400|      400| 1|2016-04-11 11:46:24|      1.0|
|      500|      500| 1|2016-04-11 12:03:21|      1.0|
+-----+-----+-----+-----+-----+
```

property output_columns

Columns generated by transformation.

transform (*dataframe: pyspark.sql.dataframe.DataFrame*) → *pyspark.sql.dataframe.DataFrame*

Performs a transformation to the feature pipeline.

Parameters dataframe – input dataframe to be transformed.

Returns Transformed dataframe.

property transformer

Function to use for transforming the dataframe.

H3 Transform entity.

```
class butterfree.transform.transformations.h3_transform.H3HashTransform(h3_resolutions:
                                                                    List[int],
                                                                    lat_column:
                                                                    str,
                                                                    lng_column:
                                                                    str)
```

Bases: `butterfree.transform.transformations.transform_component.TransformComponent`

Defines a H3 hash transformation.

h3_resolutions
h3 resolutions from 6 to 12.

lat_column
latitude column.

lng_column
longitude column.

Example

It's necessary to declare the desired h3 resolutions and latitude and longitude columns.

```
>>> from butterfree.transform.features import Feature
>>> from butterfree.transform.transformations.h3_transform import (
...     H3HashTransform
... )
>>> from butterfree.constants import DataType
>>> from pyspark import SparkContext
>>> from pyspark.sql import session
>>> sc = SparkContext.getOrCreate()
>>> spark = session.SparkSession(sc)
>>> df = spark.createDataFrame([(1, 200, -23.554190, -46.670723),
...                             (1, 300, -23.554190, -46.670723),
...                             (1, 400, -23.554190, -46.670723),
...                             (1, 500, -23.554190, -46.670723)]
...                             ).toDF("id", "feature", "lat", "lng")
>>> feature = Feature(
...     name="feature",
...     description="h3 hash transform usage example",
...     dtype=DataType.STRING,
...     transformation=H3HashTransform(
...         h3_resolutions=[6, 7, 8, 9, 10, 11, 12],
...         lat_column="lat",
...         lng_column="lng",
...     )
... )
>>> feature.transform(df).show()
```

feature	id	lat	lng	lat_lng_h3_hash_6	lat_lng_h3_hash_7
200	1	-23.55419	-46.670723	86a8100e	87a8100ea
300	1	-23.55419	-46.670723	86a8100e	87a8100ea
400	1	-23.55419	-46.670723	86a8100e	87a8100ea
500	1	-23.55419	-46.670723	86a8100e	87a8100ea

property output_columns

Columns generated by the transformation.

transform (*dataframe: pyspark.sql.dataframe.DataFrame*) → *pyspark.sql.dataframe.DataFrame*

Performs a transformation to the feature pipeline.

Parameters *dataframe* – input dataframe.

Returns Transformed dataframe.

with_stack () → *butterfree.transform.transformations.h3_transform.H3HashTransform*

Add a final Stack step to the transformation.

A new column will be created stacking all the the resolution columns generated by H3. The name of this column will be the parent name of H3Transform, for example, the name of the KeyFeature that has the H3 as a transformation.

butterfree.transform.transformations.h3_transform.define_h3 (*args) → Any
UDF for h3 hash retrieval.

butterfree.transform.transformations.h3_transform.lat
latitude column.

butterfree.transform.transformations.h3_transform.lng
longitude column.

butterfree.transform.transformations.h3_transform.resolution
desired h3 resolution.

Spark Function Transform entity.

class *butterfree.transform.transformations.spark_function_transform.SparkFunctionTransform*

Bases: *butterfree.transform.transformations.transform_component.TransformComponent*

Defines an Spark Function.

function

namedtuple with spark function and data type.

Example

It's necessary to declare the function method, Any spark and user defined functions are supported.

```
>>> from butterfree.transform.transformations import SparkFunctionTransform
>>> from butterfree.constants.columns import TIMESTAMP_COLUMN
>>> from butterfree.transform.features import Feature
>>> from butterfree.transform.utils import Function
>>> from butterfree.constants import DataType
>>> from pyspark import SparkContext
>>> from pyspark.sql import session
>>> from pyspark.sql.types import TimestampType
>>> from pyspark.sql import functions
>>> sc = SparkContext.getOrCreate()
>>> spark = session.SparkSession(sc)
>>> df = spark.createDataFrame([(1, "2016-04-11 11:31:11", 200),
...                             (1, "2016-04-11 11:44:12", 300),
...                             (1, "2016-04-11 11:46:24", 400),
...                             (1, "2016-04-11 12:03:21", 500)]
...                             ).toDF("id", "timestamp", "feature")
```

(continues on next page)

(continued from previous page)

```
>>> df = df.withColumn("timestamp", df.timestamp.cast(TimestampType()))
>>> feature = Feature(
...     name="feature",
...     description="spark function transform",
...     transformation=SparkFunctionTransform(
...         functions=[Function(functions.cos, DataType.DOUBLE)],)
... )
>>> feature.transform(df).orderBy("timestamp").show()
+-----+-----+-----+-----+
| id|      timestamp|feature|   feature__cos|
+-----+-----+-----+-----+
|  1|2016-04-11 11:31:11|    200|  0.4871876750070059|
|  1|2016-04-11 11:44:12|    300|-0.02209661927868...|
|  1|2016-04-11 11:46:24|    400| -0.525296338642536|
|  1|2016-04-11 12:03:21|    500| -0.883849273431478|
+-----+-----+-----+-----+
```

We can use this transformation with windows.

```
>>> feature_row_windows = Feature(
...     name="feature",
...     description="spark function transform with windows",
...     transformation=SparkFunctionTransform(
...         functions=[Function(functions.avg, DataType.DOUBLE)],)
...         .with_window(partition_by="id",
...                       mode="row_windows",
...                       window_definition=["2 events"],
...     )
... )
>>> feature_row_windows.transform(df).orderBy("timestamp").show()
+-----+-----+-----+-----+
|feature | id|      timestamp|feature_avg_over_2_events_row_windows|
+-----+-----+-----+-----+
|    200|  1|2016-04-11 11:31:11|                200.0|
|    300|  1|2016-04-11 11:44:12|                250.0|
|    400|  1|2016-04-11 11:46:24|                350.0|
|    500|  1|2016-04-11 12:03:21|                450.0|
+-----+-----+-----+-----+
```

It's important to notice that transformation doesn't affect the dataframe granularity.

property `output_columns`

Columns generated by the transformation.

transform (*dataframe*: `pyspark.sql.dataframe.DataFrame`) → `pyspark.sql.dataframe.DataFrame`
 Performs a transformation to the feature pipeline.

Parameters *dataframe* – input dataframe.

Returns Transformed dataframe.

with_window (*partition_by*: `str`, *window_definition*: `List[str]`, *order_by*: `str = None`, *mode*: `str = None`) → `butterfree.transform.transformations.spark_function_transform.SparkFunctionTransform`
 Create a list with windows defined.

SQL Expression Transform entity.

class `butterfree.transform.transformations.sql_expression_transform.SQLExpressionTransform`

Bases: `butterfree.transform.transformations.transform_component.TransformComponent`

Defines an SQL Expression Transformation.

expression

SQL expression defined by the user.

Example

It's necessary to declare the custom SQL query, such that it corresponds to operations between existing columns in the dataframe. Besides, the usage of SparkSQL functions is also allowed. Finally, a “complete select statement”, such as “select col_a * col_b from my_table”, is not necessary, just the simple operations are required, for instance “col_a / col_b” or “col_a * col_b”.

```
>>> from butterfree.transform.features import Feature
>>> from butterfree.transform.transformations import SQLExpressionTransform
>>> from butterfree.constants import DataType
>>> from pyspark import SparkContext
>>> from pyspark.sql import session
>>> import pyspark.sql.functions as F
>>> sc = SparkContext.getOrCreate()
>>> spark = session.SparkSession(sc)
>>> df = spark.createDataFrame([(1, "2016-04-11 11:31:11", 200, 200),
...                             (1, "2016-04-11 11:44:12", 300, 300),
...                             (1, "2016-04-11 11:46:24", 400, 400),
...                             (1, "2016-04-11 12:03:21", 500, 500)]
...                             ).toDF("id", "timestamp", "feature1", "feature2")
>>> feature = Feature(
...     name="feature",
...     description="SQL expression transform usage example",
...     dtype=DataType.DOUBLE,
...     transformation=SQLExpressionTransform(expression="feature1/feature2"),
... )
>>> feature.transform(df).orderBy("timestamp").show()
+-----+-----+-----+-----+-----+
|feature1|feature2| id|          timestamp|feature1_over_feature2|
+-----+-----+-----+-----+-----+
|    200|    200|  1|2016-04-11 11:31:11|                1.0|
|    300|    300|  1|2016-04-11 11:44:12|                1.0|
|    400|    400|  1|2016-04-11 11:46:24|                1.0|
|    500|    500|  1|2016-04-11 12:03:21|                1.0|
+-----+-----+-----+-----+-----+
```

property output_columns

Columns generated by the transformation.

transform (*dataframe*: `pyspark.sql.dataframe.DataFrame`) → `pyspark.sql.dataframe.DataFrame`

Performs a transformation to the feature pipeline.

Parameters *dataframe* – input dataframe.

Returns Transformed dataframe.

Stack Transform entity.


```
class butterfree.transform.transformations.stack_transform.StackTransform(*columns_names:
                                                                    str,
                                                                    is_regex:
                                                                    bool
                                                                    =
                                                                    False)
```

Bases: `butterfree.transform.transformations.transform_component.TransformComponent`

Defines a Stack transformation.

For instantiation it is needed the name of the columns or a pattern to use to find the columns that need to be stacked. This transform generates just one column as output.

columns_names

full names or patterns to search for target columns on the dataframe. By default a single * character is considered a wildcard and can be anywhere in the string, multiple wildcards are not supported. Strings can also start with an ! (exclamation mark), it indicates a negation, be it a regular string or simple pattern. When parameter :param is_regex: is *True*, simple patterns wildcards and negation are disabled and all strings are interpreted as regular expressions.

is_regex

boolean flag to indicate if columns_names passed are a Python regex string patterns.

Example

```
>>> from pyspark import SparkContext
>>> from pyspark.sql import session
>>> from butterfree.testing.dataframe import create_df_from_collection
>>> from butterfree.constants import DataType
>>> from butterfree.transform.transformations import StackTransform
>>> from butterfree.transform.features import Feature
>>> spark_context = SparkContext.getOrCreate()
>>> spark_session = session.SparkSession(spark_context)
>>> data = [
...     {"feature": 100, "id_a": 1, "id_b": 2},
...     {"feature": 120, "id_a": 3, "id_b": 4},
... ]
>>> df = create_df_from_collection(data, spark_context, spark_session)
>>> df.collect()
[Row(feature=100, id_a=1, id_b=2), Row(feature=120, id_a=3, id_b=4)]
>>> feature = Feature(
...     name="stack_ids",
...     description="id_a and id_b stacked in a single column.",
...     dtype=DataType.BIGINT,
...     transformation=StackTransform("id_a", "id_b"),
... )
>>> feature.transform(df).collect()
[
  Row(feature=100, id_a=1, id_b=2, stack_ids=1),
  Row(feature=100, id_a=1, id_b=2, stack_ids=2),
  Row(feature=120, id_a=3, id_b=4, stack_ids=3),
  Row(feature=120, id_a=3, id_b=4, stack_ids=4)
]
```

The StackTransform can be instantiated using a column pattern instead of the columns full names. Like this way:

```
>>> feature = Feature(
...     name="stack_ids",
...     description="id_a and id_b stacked in a single column.",
...     dtype=DataType.BIGINT,
...     transformation=StackTransform(columns_prefix="id_*"),
... )
```

property output_columns

Columns generated by the transformation.

transform (*dataframe*: *pyspark.sql.dataframe.DataFrame*) → *pyspark.sql.dataframe.DataFrame*

Performs a transformation to the feature pipeline.

Parameters *dataframe* – input dataframe.

Returns Transformed dataframe.

Transform Abstract Class.

class *butterfree.transform.transformations.transform_component.TransformComponent*

Bases: *abc.ABC*

Defines an abstract class for Transform entities.

parent

parent transform component.

abstract property output_columns

Columns generated by the transformation.

property parent

Parent transform component.

abstract transform (*dataframe*: *pyspark.sql.dataframe.DataFrame*) → *pyspark.sql.dataframe.DataFrame*

Performs a transformation to the feature pipeline.

Parameters *dataframe* – input dataframe.

Returns Transformed dataframe.

Module contents

Holds all transformations to be used by Features.

A transformation must inherit from a *TransformComponent* and handle data modification, renaming and cast types using parent's (a *Feature*) information.

class *butterfree.transform.transformations.AggregatedTransform* (*functions*:

List[butterfree.transform.utils.function.F

fil-

ter_expression:

str = None)

Bases: *butterfree.transform.transformations.transform_component.TransformComponent*

Specifies an aggregation.

This transformation needs to be used within an *AggregatedFeatureSet*. Unlike the other transformations, this class won't have a *transform* method implemented.

The idea behind aggregating is that, in spark, we should execute all aggregation functions after a single groupby. So an AggregateFeatureSet will have many Features with AggregatedTransform. If each one of them needs to apply a groupby.agg(), then we must join all the results in the end, making this computation extremely slow.

Now, the AggregateFeatureSet will collect all Features' AggregatedTransform definitions and run, at once, a groupby.agg(*aggregations).

This class helps defining on a feature, which aggregation function will be applied to build a new aggregated column. Allowed aggregations are registered under the

allowed_aggregations property.

functions

namedtuple with aggregation function and data type.

filter_expression

sql boolean expression to be used inside agg function. The filter expression can be used to aggregate some column only with records that obey certain condition. Has the same behaviour of the following SQL expression: *agg(case when filter_expression then col end)*

Example

```
>>> from butterfree.transform.transformations import AggregatedTransform
>>> from butterfree.transform.features import Feature
>>> from butterfree.constants import DataType
>>> from butterfree.transform.utils import Function
>>> import pyspark.sql.functions as F
>>> feature = Feature(
...     name="feature",
...     description="aggregated transform",
...     transformation=AggregatedTransform(
...         functions=[
...             Function(F.avg, DataType.DOUBLE),
...             Function(F.stddev_pop, DataType.DOUBLE)],
...     ),
...     from_column="somenumber",
... )
>>> feature.get_output_columns()
['feature__avg', 'feature__stddev_pop']
>>> feature.transform(anydf)
NotImplementedError: ...
```

property aggregations

Aggregated spark columns.

property output_columns

Columns names generated by the transformation.

transform (*dataframe: pyspark.sql.dataframe.DataFrame*) → pyspark.sql.dataframe.DataFrame
(NotImplemented) Performs a transformation to the feature pipeline.

For the AggregatedTransform, the transformation won't be applied without using an AggregatedFeatureSet.

Parameters dataframe – input dataframe.

Raises NotImplementedError. –

```
class butterfree.transform.transformations.CustomTransform(transformer:
                                                    Callable[[...], Any],
                                                    **kwargs: Any)

Bases:      butterfree.transform.transformations.t.transform_component.
            TransformComponent
```

Defines a Custom Transform.

t transformer

function to use for transforming the dataframe

****kwargs**

kwargs for the transformer

Example

It's necessary to instantiate the CustomTransform class using a custom method that must always receive a dataframe and the parent feature as arguments and the custom arguments must be passed to the builder through **kwargs*.

```
>>> from butterfree.transform.transformations import CustomTransform
>>> from butterfree.transform.features import Feature
>>> from butterfree.constants import DataType
>>> from pyspark import SparkContext
>>> from pyspark.sql import session
>>> import pyspark.sql.functions as F
>>> sc = SparkContext.getOrCreate()
>>> spark = session.SparkSession(sc)
>>> df = spark.createDataFrame([(1, "2016-04-11 11:31:11", 200, 200),
...                             (1, "2016-04-11 11:44:12", 300, 300),
...                             (1, "2016-04-11 11:46:24", 400, 400),
...                             (1, "2016-04-11 12:03:21", 500, 500)]
...                             ).toDF("id", "timestamp", "feature1", "feature2")
>>> def divide(df, parent_feature, column1, column2):
...     name = parent_feature.get_output_columns()[0]
...     df = df.withColumn(name, F.col(column1) / F.col(column2))
...     return df
>>> feature = Feature(
...     name="feature",
...     description="custom transform usage example",
...     dtype=DataType.DOUBLE,
...     transformation=CustomTransform(
...         transformer=divide, column1="feature1", column2="feature2",
...     )
... )
>>> feature.transform(df).orderBy("timestamp").show()
+-----+-----+-----+-----+-----+
|feature1|feature2| id|          timestamp|feature|
+-----+-----+-----+-----+-----+
|      200|      200|  1|2016-04-11 11:31:11|      1.0|
|      300|      300|  1|2016-04-11 11:44:12|      1.0|
|      400|      400|  1|2016-04-11 11:46:24|      1.0|
|      500|      500|  1|2016-04-11 12:03:21|      1.0|
+-----+-----+-----+-----+-----+
```

property output_columns

Columns generated by transformation.

transform (*dataframe*: *pyspark.sql.dataframe.DataFrame*) → *pyspark.sql.dataframe.DataFrame*
 Performs a transformation to the feature pipeline.

Parameters *dataframe* – input dataframe to be transformed.

Returns Transformed dataframe.

property transformer

Function to use for transforming the dataframe.

class `butterfree.transform.transformations.SQLExpressionTransform` (*expression*:
str)

Bases: `butterfree.transform.transformations.transform_component.TransformComponent`

Defines an SQL Expression Transformation.

expression

SQL expression defined by the user.

Example

It's necessary to declare the custom SQL query, such that it corresponds to operations between existing columns in the dataframe. Besides, the usage of SparkSQL functions is also allowed. Finally, a “complete select statement”, such as “select col_a * col_b from my_table”, is not necessary, just the simple operations are required, for instance “col_a / col_b” or “col_a * col_b”.

```
>>> from butterfree.transform.features import Feature
>>> from butterfree.transform.transformations import SQLExpressionTransform
>>> from butterfree.constants import DataType
>>> from pyspark import SparkContext
>>> from pyspark.sql import session
>>> import pyspark.sql.functions as F
>>> sc = SparkContext.getOrCreate()
>>> spark = session.SparkSession(sc)
>>> df = spark.createDataFrame([(1, "2016-04-11 11:31:11", 200, 200),
...                             (1, "2016-04-11 11:44:12", 300, 300),
...                             (1, "2016-04-11 11:46:24", 400, 400),
...                             (1, "2016-04-11 12:03:21", 500, 500)]
...                             ).toDF("id", "timestamp", "feature1", "feature2")
>>> feature = Feature(
...     name="feature",
...     description="SQL expression transform usage example",
...     dtype=DataType.DOUBLE,
...     transformation=SQLExpressionTransform(expression="feature1/feature2"),
... )
>>> feature.transform(df).orderBy("timestamp").show()
+-----+-----+-----+-----+-----+
|feature1|feature2| id|          timestamp|feature1_over_feature2|
+-----+-----+-----+-----+-----+
|      200|      200|  1|2016-04-11 11:31:11|              1.0|
|      300|      300|  1|2016-04-11 11:44:12|              1.0|
|      400|      400|  1|2016-04-11 11:46:24|              1.0|
|      500|      500|  1|2016-04-11 12:03:21|              1.0|
+-----+-----+-----+-----+-----+
```

property output_columns

Columns generated by the transformation.

transform (*dataframe*: *pyspark.sql.dataframe.DataFrame*) → *pyspark.sql.dataframe.DataFrame*
 Performs a transformation to the feature pipeline.

Parameters *dataframe* – input dataframe.

Returns Transformed dataframe.

class `butterfree.transform.transformations.SparkFunctionTransform` (*functions*:
List[*butterfree.transform.utils.function*
TransformComponent])

Bases: *butterfree.transform.transformations.transform_component.TransformComponent*

Defines an Spark Function.

function
 namedtuple with spark function and data type.

Example

It's necessary to declare the function method, Any spark and user defined functions are supported.

```
>>> from butterfree.transform.transformations import SparkFunctionTransform
>>> from butterfree.constants.columns import TIMESTAMP_COLUMN
>>> from butterfree.transform.features import Feature
>>> from butterfree.transform.utils import Function
>>> from butterfree.constants import DataType
>>> from pyspark import SparkContext
>>> from pyspark.sql import session
>>> from pyspark.sql.types import TimestampType
>>> from pyspark.sql import functions
>>> sc = SparkContext.getOrCreate()
>>> spark = session.SparkSession(sc)
>>> df = spark.createDataFrame([(1, "2016-04-11 11:31:11", 200),
...                             (1, "2016-04-11 11:44:12", 300),
...                             (1, "2016-04-11 11:46:24", 400),
...                             (1, "2016-04-11 12:03:21", 500)]
...                             ).toDF("id", "timestamp", "feature")
>>> df = df.withColumn("timestamp", df.timestamp.cast(TimestampType()))
>>> feature = Feature(
...     name="feature",
...     description="spark function transform",
...     transformation=SparkFunctionTransform(
...         functions=[Function(functions.cos, DataType.DOUBLE)],)
... )
>>> feature.transform(df).orderBy("timestamp").show()
+----+-----+-----+-----+
| id|      timestamp|feature|  feature__cos|
+----+-----+-----+-----+
|  1|2016-04-11 11:31:11|    200| 0.4871876750070059|
|  1|2016-04-11 11:44:12|    300|-0.02209661927868...|
|  1|2016-04-11 11:46:24|    400| -0.525296338642536|
|  1|2016-04-11 12:03:21|    500| -0.883849273431478|
+----+-----+-----+-----+
```

We can use this transformation with windows.

```
>>> feature_row_windows = Feature(
...     name="feature",
```

(continues on next page)

(continued from previous page)

```

...     description="spark function transform with windows",
...     transformation=SparkFunctionTransform(
...         functions=[Function(functions.avg, DataType.DOUBLE)],)
...         .with_window(partition_by="id",
...                       mode="row_windows",
...                       window_definition=["2 events"],
...     )
... )
... )
>>> feature_row_windows.transform(df).orderBy("timestamp").show()
+-----+-----+-----+-----+
|feature | id|          timestamp| feature_avg_over_2_events_row_windows|
+-----+-----+-----+-----+
|      200|  1|2016-04-11 11:31:11|                200.0|
|      300|  1|2016-04-11 11:44:12|                250.0|
|      400|  1|2016-04-11 11:46:24|                350.0|
|      500|  1|2016-04-11 12:03:21|                450.0|
+-----+-----+-----+-----+

```

It's important to notice that transformation doesn't affect the dataframe granularity.

property `output_columns`

Columns generated by the transformation.

transform (*dataframe*: *pyspark.sql.dataframe.DataFrame*) → *pyspark.sql.dataframe.DataFrame*

Performs a transformation to the feature pipeline.

Parameters *dataframe* – input dataframe.

Returns Transformed dataframe.

with_window (*partition_by*: *str*, *window_definition*: *List[str]*, *order_by*: *str* = *None*, *mode*: *str* = *None*) → *butterfree.transform.transformations.spark_function_transform.SparkFunctionTransform*
 Create a list with windows defined.

class `butterfree.transform.transformations.StackTransform` (**columns_names*: *str*, *is_regex*: *bool* = *False*)
 Bases: `butterfree.transform.transformations.transform_component.TransformComponent`

Defines a Stack transformation.

For instantiation it is needed the name of the columns or a pattern to use to find the columns that need to be stacked. This transform generates just one column as output.

property `columns_names`

full names or patterns to search for target columns on the dataframe. By default a single * character is considered a wildcard and can be anywhere in the string, multiple wildcards are not supported. Strings can also start with an ! (exclamation mark), it indicates a negation, be it a regular string or simple pattern. When parameter `:param is_regex` is *True*, simple patterns wildcards and negation are disabled and all strings are interpreted as regular expressions.

property `is_regex`

boolean flag to indicate if `columns_names` passed are a Python regex string patterns.

Example

```

>>> from pyspark import SparkContext
>>> from pyspark.sql import session
>>> from butterfree.testing.dataframe import create_df_from_collection
>>> from butterfree.constants import DataType
>>> from butterfree.transform.transformations import StackTransform
>>> from butterfree.transform.features import Feature
>>> spark_context = SparkContext.getOrCreate()
>>> spark_session = session.SparkSession(spark_context)
>>> data = [
...     {"feature": 100, "id_a": 1, "id_b": 2},
...     {"feature": 120, "id_a": 3, "id_b": 4},
... ]
>>> df = create_df_from_collection(data, spark_context, spark_session)
>>> df.collect()
[Row(feature=100, id_a=1, id_b=2), Row(feature=120, id_a=3, id_b=4)]
>>> feature = Feature(
...     name="stack_ids",
...     description="id_a and id_b stacked in a single column.",
...     dtype=DataType.BIGINT,
...     transformation=StackTransform("id_a", "id_b"),
... )
>>> feature.transform(df).collect()
[
  Row(feature=100, id_a=1, id_b=2, stack_ids=1),
  Row(feature=100, id_a=1, id_b=2, stack_ids=2),
  Row(feature=120, id_a=3, id_b=4, stack_ids=3),
  Row(feature=120, id_a=3, id_b=4, stack_ids=4)
]

```

The StackTransform can be instantiated using a column pattern instead of the columns full names. Like this way:

```

>>> feature = Feature(
...     name="stack_ids",
...     description="id_a and id_b stacked in a single column.",
...     dtype=DataType.BIGINT,
...     transformation=StackTransform(columns_prefix="id_*"),
... )

```

property output_columns

Columns generated by the transformation.

transform (*dataframe*: *pyspark.sql.dataframe.DataFrame*) → *pyspark.sql.dataframe.DataFrame*
 Performs a transformation to the feature pipeline.

Parameters dataframe – input dataframe.

Returns Transformed dataframe.

class `butterfree.transform.transformations.TransformComponent`

Bases: `abc.ABC`

Defines an abstract class for Transform entities.

parent

parent transform component.

abstract property output_columns

Columns generated by the transformation.

property parent

Parent transform component.

abstract transform (*dataframe*: *pyspark.sql.dataframe.DataFrame*) → *pyspark.sql.dataframe.DataFrame*

Performs a transformation to the feature pipeline.

Parameters *dataframe* – input dataframe.

Returns Transformed dataframe.

butterfree.transform.utils package**Submodules**

Utils for date range generation.

`butterfree.transform.utils.date_range.get_date_range` (*client*: *butterfree.clients.spark_client.SparkClient*, *start_date*: *Union[str, datetime.datetime]*, *end_date*: *Union[str, datetime.datetime]*, *step*: *int = None*) → *pyspark.sql.dataframe.DataFrame*

Create a date range dataframe.

The dataframe returning from this method will containing a single column `TIMESTAMP_COLUMN`, of timestamp type, with dates between start and end.

Parameters

- **client** – a spark client.
- **start_date** – range beginning value (inclusive).
- **end_date** – range last value (exclusive)
- **step** – optional step, in seconds.

Returns A single column date range spark dataframe.

Utils for custom or spark function to generation namedtuple.

class `butterfree.transform.utils.function.Function` (*func*: *Callable*, *data_type*: *butterfree.constants.data_type.DataType*)

Bases: `object`

Define a class Function.

Like a namedtuple: `Function = namedtuple("Function", ["function", "data_type"])`.

func

custom or spark functions, such as avg, std, count. For more information check spark functions:

`'https://spark.apache.org/docs/2.3.1/api/python/_modules/pyspark/sql/functions.html'`

For custom functions, look the path: `'butterfree/transform/transformations/user_defined_functions'`.

data_type

data type for the output columns.

property data_type

Function to be used in the transformation.

property func

Function to be used in the transformation.

Holds function for defining window in DataFrames.

```
class butterfree.transform.utils.window_spec.FrameBoundaries (mode: Optional[str], window_definition: str)
```

Bases: object

Utility functions for defining the frame boundaries.

Parameters

- **mode** – available modes to be used in time aggregations.
- **window_definition** – time ranges to be used in the windows,
- **can be second** (*it*) –

get (*window*: *pyspark.sql.window.WindowSpec*) → Any
Returns window with or without the frame boundaries.

property window_size

Returns window size.

property window_unit

Returns window unit.

```
class butterfree.transform.utils.window_spec.Window (window_definition: str,  
           partition_by: Optional[Union[pyspark.sql.column.Column,  
           str, List[str]]] =  
           None, order_by: Optional[Union[pyspark.sql.column.Column,  
           str]] = None, mode: str = None,  
           slide: str = None)
```

Bases: object

Utility functions for defining a window specification.

Parameters

- **partition_by** – the partitioning defined.
- **order_by** – the ordering defined.
- **mode** – available modes to be used in time aggregations.
- **window_definition** – time ranges to be used in the windows, it can be second(s), minute(s), hour(s), day(s), week(s) and year(s),

Use the static methods in *Window* to create a *WindowSpec*.

```
DEFAULT_SLIDE_DURATION: str = '1 day'
```

get () → Any
Defines a common window to be used both in time and rows windows.

get_name () → str
Return window suffix name based on passed criteria.

Module contents

This module holds utils to be used by transformations.

class `butterfree.transform.utils.Function` (*func*: Callable, *data_type*: `butterfree.constants.data_type.DataType`)

Bases: object

Define a class Function.

Like a namedtuple: `Function = namedtuple("Function", ["function", "data_type"])`.

func

custom or spark functions, such as avg, std, count. For more information check spark functions:

`'https://spark.apache.org/docs/2.3.1/api/python/_modules/pyspark/sql/functions.html'`

For custom functions, look the path: `'butterfree/transform/transformations/user_defined_functions'`.

data_type

data type for the output columns.

property data_type

Function to be used in the transformation.

property func

Function to be used in the transformation.

class `butterfree.transform.utils.Window` (*window_definition*: str, *partition_by*: *Optional*[*Union*[`pyspark.sql.column.Column`, str, *List*[str]]] = None, *order_by*: *Optional*[*Union*[`pyspark.sql.column.Column`, str]] = None, *mode*: str = None, *slide*: str = None)

Bases: object

Utility functions for defining a window specification.

Parameters

- **partition_by** – he partitioning defined.
- **order_by** – the ordering defined.
- **mode** – available modes to be used in time aggregations.
- **window_definition** – time ranges to be used in the windows, it can be second(s), minute(s), hour(s), day(s), week(s) and year(s),

Use the static methods in `Window` to create a `WindowSpec`.

DEFAULT_SLIDE_DURATION: str = '1 day'

get () → Any
Defines a common window to be used both in time and rows windows.

get_name () → str
Return window suffix name based on passed criteria.

Submodules

AggregatedFeatureSet entity.

```
class butterfree.transform.aggregated_feature_set.AggregatedFeatureSet (name:
                                                                    str,
                                                                    en-
                                                                    tity:
                                                                    str,
                                                                    de-
                                                                    scrip-
                                                                    tion:
                                                                    str,
                                                                    keys:
                                                                    List[butterfree.transform.featur-
                                                                    times-
                                                                    tamp:
                                                                    but-
                                                                    ter-
                                                                    free.transform.features.times-
                                                                    fea-
                                                                    tures:
                                                                    List[butterfree.transform.featur-
```

Bases: *butterfree.transform.feature_set.FeatureSet*

Holds metadata about the aggregated feature set.

This class overrides some methods of the parent FeatureSet class and has specific methods for aggregations.

The AggregatedTransform can only be used on AggregatedFeatureSets. The construct method will be responsible by collecting every feature's transformation definitions so it can run a groupby over the input dataframe, taking into account whether the user want's to run an rolling window aggregation, pivoting or just apply aggregation functions.

Example

This an example regarding the aggregated feature set definition. All features and its transformations are defined.

```
>>> from butterfree.transform.aggregated_feature_set import (
...     AggregatedFeatureSet
... )
>>> from butterfree.transform.features import (
...     Feature,
...     KeyFeature,
...     TimestampFeature,
... )
>>> from butterfree.transform.transformations import (
...     AggregatedTransform,
... )
>>> from butterfree.constants import DataType
>>> from butterfree.clients import SparkClient
>>> from butterfree.transform.utils import Function
>>> import pyspark.sql.functions as F
>>> client = SparkClient()
>>> client.conn.conf.set("spark.sql.session.timeZone", "UTC")
>>> dataframe = client.conn.createDataFrame(
```

(continues on next page)

(continued from previous page)

```

...     [
...         (1, "2020-01-01 13:01:00+000", 1000, "publicado"),
...         (2, "2020-01-01 14:01:00+000", 2000, "publicado"),
...         (1, "2020-01-02 13:01:00+000", 2000, "alugado"),
...         (1, "2020-01-03 13:01:00+000", 1000, "despublicado"),
...         (2, "2020-01-09 14:01:00+000", 1000, "despublicado"),
...     ],
...     ("id", "ts", "rent", "status"),
... )
>>> dataframe = dataframe.withColumn("ts", dataframe["ts"].cast("timestamp"))
>>> feature_set = AggregatedFeatureSet(
...     name="aggregated_feature_set",
...     entity="entity",
...     description="description",
...     features=[
...         Feature(
...             name="feature1",
...             description="test",
...             transformation=AggregatedTransform(
...                 functions=[
...                     Function(F.avg, DataType.DOUBLE),
...                     Function(F.stddev_pop, DataType.DOUBLE)],
...                 ),
...             from_column="rent",
...         ),
...     ],
...     keys=[KeyFeature(name="id", description="lul")],
...     timestamp=TimestampFeature(from_column="ts"),
... )
>>> result = feature_set.construct(
...     dataframe=dataframe,
...     client=client,
...     end_date="2020-01-15"
... )
>>> result.show()
+---+-----+-----+-----+
| id|      timestamp|feature1__avg|feature1__stddev|
+---+-----+-----+-----+
| 1|2020-01-01 13:01:00|      1000.0|           null|
| 1|2020-01-03 13:01:00|      1000.0|           null|
| 1|2020-01-02 13:01:00|      2000.0|           null|
| 2|2020-01-09 14:01:00|      1000.0|           null|
| 2|2020-01-01 14:01:00|      2000.0|           null|
+---+-----+-----+-----+

```

Since you didn't define a window, the `AggregateFeatureSet` will always group by keys and timestamp feature columns. So in this example, there will be no changes to the dataframe, since it doesn't duplicate on id and timestamp :)

Let's run one example with windows:

```

>>> feature_set.with_windows(definitions=["3 days"])
>>> result = feature_set.construct(
...     dataframe=dataframe,
...     client=client,
...     end_date="2020-01-15"
... )

```

(continues on next page)

(continued from previous page)

```
>>> result.orderBy("timestamp", "id").show()
+----+-----+-----+-----+
| id|      timestamp|feature1__avg_over_3_days_rolling_windows|
+----+-----+-----+-----+
| 1|2020-01-01 00:00:00|                                     null|
| 2|2020-01-01 00:00:00|                                     null|
| 1|2020-01-02 00:00:00|                                     1000.0|
| 2|2020-01-02 00:00:00|                                     2000.0|
| 1|2020-01-03 00:00:00|                                     1500.0|
| 1|2020-01-04 00:00:00|                    1333.3333333333333|
| 1|2020-01-05 00:00:00|                                     1500.0|
| 2|2020-01-05 00:00:00|                                     null|
| 1|2020-01-06 00:00:00|                                     1000.0|
| 1|2020-01-07 00:00:00|                                     null|
| 2|2020-01-10 00:00:00|                                     1000.0|
| 2|2020-01-13 00:00:00|                                     null|
+----+-----+-----+-----+
```

id	timestamp	feature1__stddev_over_3_days_rolling_windows
1 2 1 2 1 1 2 1 1 2 2	2020-01-01 00:00:00 2020-01-01 00:00:00 2020-01-02 00:00:00 2020-01-02 00:00:00 2020-01-03 00:00:00 2020-01-04 00:00:00 2020-01-05 00:00:00 2020-01-05 00:00:00 2020-01-06 00:00:00 2020-01-07 00:00:00 2020-01-10 00:00:00 2020-01-13 00:00:00	null null null null 707.1067811865476 577.3502691896258 707.1067811865476 null null null null null

(Had to break down the table result.)

And with pivot:

```
>>> feature_set.with_pivot(column="status", values=["publicado", "despublicado"])
+----+-----+-----+-----+
| id|      timestamp|publicado_feature1__avg|publicado_feature1__stddev|
+----+-----+-----+-----+
| 1|2020-01-01 13:01:00|          1000.0|          null|
| 2|2020-01-01 14:01:00|          2000.0|          null|
| 1|2020-01-02 13:01:00|          null|          null|
| 1|2020-01-03 13:01:00|          null|          null|
| 2|2020-01-09 14:01:00|          null|          null|
+----+-----+-----+-----+
```

id	timestamp	despublicado_feature1__avg	despublicado_feature1__stddev
1 2 1 1 2	2020-01-01 13:01:00 2020-01-01 14:01:00 2020-01-02 13:01:00 2020-01-03 13:01:00 2020-01-09 14:01:00	null null null 1000.0 1000.0	null null null null null

As you can see, we need to pass the values you want to pivot. It optimizes this processing in spark and allows ignoring values when pivoting. If you wanted to get the pivot aggregation for "alugado" too, just use:

```
>>> feature_set.with_pivot(
...     column="status", values=["publicado", "despublicado", "alugado"]
... )
```

You can also run it with pivot AND windows:

```
>>> feature_set.with_pivot(
...     column="status",
...     values=["publicado", "despublicado", "alugado"]
... ).with_windows(definitions=["1 day", "2 weeks"])
```

The construct method will execute the feature set, computing all the defined aggregated transformations at once. Remember: when using an AggregatedFeatureSet without window, the group will use the timestamp column.

construct (*dataframe*: *pyspark.sql.dataframe.DataFrame*, *client*: *butterfree.clients.spark_client.SparkClient*, *end_date*: *str = None*, *num_processors*: *int = None*, *start_date*: *str = None*) → *pyspark.sql.dataframe.DataFrame*

Use all the features to build the feature set dataframe.

After that, there's the caching of the dataframe, however since `cache()` in Spark is lazy, an action is triggered in order to force persistence, but we only cache if it is not a streaming spark dataframe.

Parameters

- **dataframe** – input dataframe to be transformed by the features.
- **client** – client responsible for connecting to Spark session.
- **end_date** – user defined max date for having aggregated data (exclusive).
- **num_processors** – cluster total number of processors for repartitioning.
- **start_date** – user defined min date for having aggregated data.

Returns Spark dataframe with all the feature columns.

define_start_date (*start_date*: *str = None*) → *Optional[str]*

Get aggregated feature set start date.

Parameters **start_date** – start date regarding source dataframe.

Returns start date.

property features

Features to compose the feature set.

property features_columns

Name of the columns of all features in feature set.

get_schema () → *List[Dict[str, Any]]*

Get feature set schema.

Returns List of dicts with the feature set schema.

with_distinct (*subset*: *List*, *keep*: *str = 'last'*) → *butterfree.transform.aggregated_feature_set.AggregatedFeatureSet*

Add a distinct configuration for your aggregated feature set.

Parameters

- **subset** – the columns where it will identify duplicates.
- **keep** – determines which duplicates to keep. Default 'last'. - first : Ascending sorting by timestamp. - last : Descending sorting by timestamp.

Returns An AggregatedFeatureSet configured with distinct.

with_pivot (*column*: str, *values*: Optional[List[Union[bool, float, int, str]]]) → *butterfree.transform.aggregated_feature_set.AggregatedFeatureSet*

Add a pivot configuration for your aggregated feature set.

This means we will group the input data, pivot over the column parameter and run each aggregation function over the columns within each value group in the values parameter. In spark it will be something like: `dataframe.groupBy(*group).pivot(column, values).agg(*aggregations)`

Parameters

- **column** – the column, containing categorical values, to pivot on.
- **values** – the distinct values you want to be pivoted.

Returns An AggregatedFeatureSet configured with pivot.

with_windows (*definitions*: List[str], *slide*: str = None) → *butterfree.transform.aggregated_feature_set.AggregatedFeatureSet*

Create a list with windows defined.

FeatureSet entity.

```
class butterfree.transform.feature_set.FeatureSet (name: str, entity: str,
                                                description: str, keys:
                                                List[butterfree.transform.features.key_feature.KeyFeature],
                                                timestamp: butter-
                                                free.transform.features.timestamp_feature.TimestampFeature,
                                                features:
                                                List[butterfree.transform.features.feature.Feature])
```

Bases: *butterfree.hooks.hookable_component.HookableComponent*

Holds metadata about the feature set and constructs the final dataframe.

name

name of the feature set.

entity

business context tag for the feature set, an entity for which we are creating all these features.

description

details about the feature set purpose.

keys

key features to define this feature set. Values for keys (may be a composition) should be unique on each moment in time (controlled by the TimestampFeature).

timestamp

A single feature that define a timestamp for each observation in this feature set.

features

features to compose the feature set.

Example

This an example regarding the feature set definition. All features and its transformations are defined.

```
>>> from butterfree.transform import FeatureSet
>>> from butterfree.transform.features import (
...     Feature,
...     KeyFeature,
...     TimestampFeature,
... )
>>> from butterfree.transform.transformations import (
...     SparkFunctionTransform,
...     CustomTransform,
... )
>>> from butterfree.constants import DataType
>>> from butterfree.transform.utils import Function
>>> import pyspark.sql.functions as F

>>> def divide(df, fs, column1, column2):
...     name = fs.get_output_columns()[0]
...     df = df.withColumn(name, F.col(column1) / F.col(column2))
...     return df

>>> feature_set = FeatureSet(
...     name="feature_set",
...     entity="entity",
...     description="description",
...     features=[
...         Feature(
...             name="feature1",
...             description="test",
...             transformation=SparkFunctionTransform(
...                 functions=[
...                     Function(F.avg, DataType.DOUBLE),
...                     Function(F.stddev_pop, DataType.DOUBLE)]
...             ).with_window(
...                 partition_by="id",
...                 order_by=TIMESTAMP_COLUMN,
...                 mode="fixed_windows",
...                 window_definition=["2 minutes", "15 minutes"],
...             ),
...         ),
...         Feature(
...             name="divided_feature",
...             description="unit test",
...             transformation=CustomTransform(
...                 transformer=divide, column1="feature1", column2="feature2",
...             ),
...         ),
...     ],
...     keys=[KeyFeature(name="id", description="The user's Main ID or device ID
↪")],
...     timestamp=TimestampFeature(),
... )

>>> feature_set.construct(dataframe=dataframe)
```

This last method (construct) will execute the feature set, computing all the defined transformations.

There's also a functionality regarding the construct method within the scope of FeatureSet called `filter_duplicated_rows`. We drop rows that have repeated values over key columns and timestamp column, we do this in order to reduce our dataframe (regarding the number of rows). A detailed explanation of this method can be found at `filter_duplicated_rows` docstring.

property columns

All data columns within this feature set.

This references all data columns that will be created by the construct method, given keys, timestamp and features of this feature set.

Returns List of column names built in this feature set.

construct (*dataframe*: `pyspark.sql.dataframe.DataFrame`, *client*: `butterfree.clients.spark_client.SparkClient`, *end_date*: `str = None`, *num_processors*: `int = None`, *start_date*: `str = None`) → `pyspark.sql.dataframe.DataFrame`

Use all the features to build the feature set dataframe.

After that, there's the caching of the dataframe, however since `cache()` in Spark is lazy, an action is triggered in order to force persistence.

Parameters

- **dataframe** – input dataframe to be transformed by the features.
- **client** – client responsible for connecting to Spark session.
- **start_date** – user defined start date.
- **end_date** – user defined end date.
- **num_processors** – cluster total number of processors for repartitioning.

Returns Spark dataframe with all the feature columns.

define_start_date (*start_date*: `str = None`) → `Optional[str]`

Get feature set start date.

Parameters **start_date** – start date regarding source dataframe.

Returns start date.

property description

Details about the feature set purpose.

property entity

Business context tag for the feature set.

property features

Features to compose the feature set.

property features_columns

Name of the columns of all features in feature set.

get_schema () → `List[Dict[str, Any]]`

Get feature set schema.

Returns List of dicts regarding cassandra feature set schema.

property keys

Key features to define this feature set.

property keys_columns

Name of the columns of all keys in feature set.

property name

Name of the feature set.

property timestamp

Defines a timestamp for each observation in this feature set.

property timestamp_column

Name of the timestamp column in feature set.

Module contents

The Transform Component of a Feature Set.

```
class butterfree.transform.FeatureSet (name: str, entity: str, description: str, keys:
    List[butterfree.transform.features.key_feature.KeyFeature],
    timestamp: butterfree.transform.features.timestamp_feature.TimestampFeature,
    features: List[butterfree.transform.features.feature.Feature])
Bases: butterfree.hooks.hookable_component.HookableComponent
```

Holds metadata about the feature set and constructs the final dataframe.

name

name of the feature set.

entity

business context tag for the feature set, an entity for which we are creating all these features.

description

details about the feature set purpose.

keys

key features to define this feature set. Values for keys (may be a composition) should be unique on each moment in time (controlled by the TimestampFeature).

timestamp

A single feature that define a timestamp for each observation in this feature set.

features

features to compose the feature set.

Example

This an example regarding the feature set definition. All features and its transformations are defined.

```
>>> from butterfree.transform import FeatureSet
>>> from butterfree.transform.features import (
...     Feature,
...     KeyFeature,
...     TimestampFeature,
... )
>>> from butterfree.transform.transformations import (
...     SparkFunctionTransform,
...     CustomTransform,
... )
>>> from butterfree.constants import DataType
>>> from butterfree.transform.utils import Function
>>> import pyspark.sql.functions as F
```

```
>>> def divide(df, fs, column1, column2):
...     name = fs.get_output_columns()[0]
...     df = df.withColumn(name, F.col(column1) / F.col(column2))
...     return df
```

```
>>> feature_set = FeatureSet(
...     name="feature_set",
...     entity="entity",
...     description="description",
...     features=[
...         Feature(
...             name="feature1",
...             description="test",
...             transformation=SparkFunctionTransform(
...                 functions=[
...                     Function(F.avg, DataType.DOUBLE),
...                     Function(F.stddev_pop, DataType.DOUBLE)]
...             ).with_window(
...                 partition_by="id",
...                 order_by=TIMESTAMP_COLUMN,
...                 mode="fixed_windows",
...                 window_definition=["2 minutes", "15 minutes"],
...             ),
...         ),
...         Feature(
...             name="divided_feature",
...             description="unit test",
...             transformation=CustomTransform(
...                 transformer=divide, column1="feature1", column2="feature2",
...             ),
...         ),
...     ],
...     keys=[KeyFeature(name="id", description="The user's Main ID or device ID
... ↪")],
...     timestamp=TimestampFeature(),
... )
```

```
>>> feature_set.construct(dataframe=dataframe)
```

This last method (construct) will execute the feature set, computing all the defined transformations.

There's also a functionality regarding the construct method within the scope of FeatureSet called `filter_duplicated_rows`. We drop rows that have repeated values over key columns and timestamp column, we do this in order to reduce our dataframe (regarding the number of rows). A detailed explanation of this method can be found at `filter_duplicated_rows` docstring.

property columns

All data columns within this feature set.

This references all data columns that will be created by the construct method, given keys, timestamp and features of this feature set.

Returns List of column names built in this feature set.

construct (*dataframe*: `pyspark.sql.dataframe.DataFrame`, *client*: `butterfree.clients.spark_client.SparkClient`, *end_date*: `str = None`, *num_processors*: `int = None`, *start_date*: `str = None`) → `pyspark.sql.dataframe.DataFrame`

Use all the features to build the feature set dataframe.

After that, there's the caching of the dataframe, however since `cache()` in Spark is lazy, an action is triggered in order to force persistence.

Parameters

- **dataframe** – input dataframe to be transformed by the features.
- **client** – client responsible for connecting to Spark session.
- **start_date** – user defined start date.
- **end_date** – user defined end date.
- **num_processors** – cluster total number of processors for repartitioning.

Returns Spark dataframe with all the feature columns.

define_start_date (*start_date: str = None*) → Optional[str]

Get feature set start date.

Parameters **start_date** – start date regarding source dataframe.

Returns start date.

property description

Details about the feature set purpose.

property entity

Business context tag for the feature set.

property features

Features to compose the feature set.

property features_columns

Name of the columns of all features in feature set.

get_schema () → List[Dict[str, Any]]

Get feature set schema.

Returns List of dicts regarding cassandra feature set schema.

property keys

Key features to define this feature set.

property keys_columns

Name of the columns of all keys in feature set.

property name

Name of the feature set.

property timestamp

Defines a timestamp for each observation in this feature set.

property timestamp_column

Name of the timestamp column in feature set.

butterfree.validations package

Submodules

Validation implementing basic checks over the dataframe.

```
class butterfree.validations.basic_validation.BasicValidation (dataframe: pyspark.sql.dataframe.DataFrame = None)
```

Bases: *butterfree.validations.validation.Validation*

Basic validation suite for Feature Set's dataframe.

dataframe
object to be verified

check () → None
Check basic validation properties about the dataframe.

Raises ValueError – if any of the verifications fail

validate_column_ts () → None
Check dataframe's ts column.

Raises ValueError – if dataframe don't have a column named ts.

validate_df_is_empty () → None
Check dataframe emptiness.

Raises ValueError – if dataframe is empty and is not streaming.

validate_df_is_spark_df () → None
Check type of dataframe object.

Raises ValueError – if dataframe is not instance of pyspark.sql.DataFrame.

Abstract Validation class.

```
class butterfree.validations.validation.Validation (dataframe: pyspark.sql.dataframe.DataFrame = None)
```

Bases: abc.ABC

Validate dataframe properties.

dataframe
data to be verified.

abstract check () → None
Check validation properties about the dataframe.

Raises ValueError – if any of the verifications fail.

input (dataframe: pyspark.sql.dataframe.DataFrame) → *butterfree.validations.validation.Validation*
Input a dataframe to check.

Parameters dataframe – data to check.

Module contents

Holds dataframe validate for multiple destinations.

```
class butterfree.validations.BasicValidation (dataframe:                                pys-
                                             park.sql.dataframe.DataFrame = None)
    Bases: butterfree.validations.validation.Validation

    Basic validation suite for Feature Set's dataframe.

    dataframe
        object to be verified

    check () → None
        Check basic validation properties about the dataframe.

        Raises ValueError – if any of the verifications fail

    validate_column_ts () → None
        Check dataframe's ts column.

        Raises ValueError – if dataframe don't have a column named ts.

    validate_df_is_empty () → None
        Check dataframe emptiness.

        Raises ValueError – if dataframe is empty and is not streaming.

    validate_df_is_spark_df () → None
        Check type of dataframe object.

        Raises ValueError – if dataframe is not instance of pyspark.sql.DataFrame.
```

Module contents

Module docstring example, following Google's docstring style.

1.10 Command-line Interface (CLI)

Butterfree has now a command-line interface, introduced with the new automatic migration ability.

As soon as you install butterfree, you can check what's available through butterfree's cli with:

```
$~ butterfree --help
```

1.10.1 Automated Database Schema Migration

When developing your feature sets, you need also to prepare your database for the changes to come into your Feature Store. Normally, when creating a new feature set, you needed to manually create a new table in cassandra. Or, when creating a new feature in an existing feature set, you needed to create new column in cassandra too.

Now, you can just use `butterfree migrate apply ...`, butterfree will scan your python files, looking for classes that inherit from `butterfree.pipelines.FeatureSetPipeline`, then compare its schema with the database schema where the feature set would be written. Then it will prepare migration queries and run against the databases.

For more information, please, check `butterfree migrate apply --help`:

1.10.2 Supported databases

This functionality currently supports only the **Cassandra** database, which is the default storage for an Online Feature Store built with Butterfree. Nonetheless, it was made with the intent to be easily extended for other databases.

Also, each database has its own rules for schema migration commands. Some changes may still require manual interference.

PYTHON MODULE INDEX

b

butterfree, 131

butterfree.clients, 20

butterfree.clients.abstract_client, 16

butterfree.clients.cassandra_client, 16

butterfree.clients.spark_client, 17

butterfree.configs, 33

butterfree.configs.db, 28

butterfree.configs.db.abstract_config, 23

butterfree.configs.db.cassandra_config, 24

butterfree.configs.db.kafka_config, 26

butterfree.configs.db.metastore_config, 27

butterfree.configs.environment, 33

butterfree.configs.logger, 33

butterfree.constants, 34

butterfree.constants.columns, 33

butterfree.constants.data_type, 33

butterfree.constants.migrations, 34

butterfree.constants.spark_constants, 34

butterfree.constants.window_definitions, 34

butterfree.dataframe_service, 37

butterfree.dataframe_service.incremental_strategy, 34

butterfree.dataframe_service.partitioning, 36

butterfree.dataframe_service.repartition, 36

butterfree.extract, 62

butterfree.extract.pre_processing, 46

butterfree.extract.pre_processing.explode_json_column_transform, 39

butterfree.extract.pre_processing.filter_transform, 41

butterfree.extract.pre_processing.forward_fill_transform, 41

butterfree.extract.pre_processing.pivot_transform, 43

butterfree.extract.pre_processing.replace_transform, 45

butterfree.extract.readers, 57

butterfree.extract.readers.file_reader, 52

butterfree.extract.readers.kafka_reader, 53

butterfree.extract.readers.reader, 55

butterfree.extract.readers.table_reader, 56

butterfree.extract.source, 60

butterfree.hooks, 67

butterfree.hooks.hook, 65

butterfree.hooks.hookable_component, 66

butterfree.hooks.schema_compatibility, 64

butterfree.hooks.schema_compatibility.cassandra_table_reader, 63

butterfree.hooks.schema_compatibility.spark_table_reader, 63

butterfree.load, 83

butterfree.load.processing, 69

butterfree.load.processing.json_transform, 68

butterfree.load.sink, 82

butterfree.load.writers, 77

butterfree.load.writers.historical_feature_store_writer, 69

butterfree.load.writers.online_feature_store_writer, 72

butterfree.load.writers.writer, 75

butterfree.migrations, 87

butterfree.migrations.database_migration, 86

butterfree.migrations.database_migration.cassandra_client, 84

butterfree.migrations.database_migration.database_reader, 84

butterfree.migrations.database_migration.metastore_client, 85

butterfree.pipelines, 90

butterfree.pipelines.feature_set_pipeline,

- 87
- butterfree.reports, 95
- butterfree.reports.metadata, 93
- butterfree.testing, 97
- butterfree.testing.dataframe, 96
- butterfree.transform, 127
- butterfree.transform.aggregated_feature_set,
120
- butterfree.transform.feature_set, 124
- butterfree.transform.features, 100
- butterfree.transform.features.feature,
97
- butterfree.transform.features.key_feature,
98
- butterfree.transform.features.timestamp_feature,
98
- butterfree.transform.transformations,
110
- butterfree.transform.transformations.aggregated_transform,
102
- butterfree.transform.transformations.custom_transform,
103
- butterfree.transform.transformations.h3_transform,
104
- butterfree.transform.transformations.spark_function_transform,
106
- butterfree.transform.transformations.sql_expression_transform,
107
- butterfree.transform.transformations.stack_transform,
108
- butterfree.transform.transformations.transform_component,
110
- butterfree.transform.utils, 119
- butterfree.transform.utils.date_range,
117
- butterfree.transform.utils.function, 117
- butterfree.transform.utils.window_spec,
118
- butterfree.validations, 131
- butterfree.validations.basic_validator,
130
- butterfree.validations.validation, 130

A

- AbstractClient (class in *butterfree.clients*), 20
- AbstractClient (class in *butterfree.clients.abstract_client*), 16
- AbstractWriteConfig (class in *butterfree.configs.db*), 28
- AbstractWriteConfig (class in *butterfree.configs.db.abstract_config*), 23
- ADD (*butterfree.migrations.database_migration.database_migration.Diff.Kind* attribute), 85
- ADD (*butterfree.migrations.database_migration.Diff.Kind* attribute), 86
- add_post_hook () (butterfree.hooks.hookable_component.HookableComponent method), 66
- add_post_hook () (butterfree.hooks.HookableComponent method), 67
- add_pre_hook () (butterfree.hooks.hookable_component.HookableComponent method), 66
- add_pre_hook () (butterfree.hooks.HookableComponent method), 67
- add_table_partitions () (butterfree.clients.spark_client.SparkClient method), 17
- add_table_partitions () (butterfree.clients.SparkClient method), 21
- agg_column (in module *butterfree.extract.pre_processing*), 49
- agg_column (in module *butterfree.extract.pre_processing.pivot_transform*), 43
- AggregatedFeatureSet (class in *butterfree.transform.aggregated_feature_set*), 120
- AggregatedTransform (class in *butterfree.transform.transformations*), 110
- AggregatedTransform (class in *butterfree.transform.transformations.aggregated_transform*), 102
- aggregation (in module *butterfree.extract.pre_processing*), 49
- aggregation (in module *butterfree.extract.pre_processing.pivot_transform*), 43
- aggregations () (butterfree.transform.transformations.aggregated_transform.AggregatedTransform property), 103
- aggregations () (butterfree.transform.transformations.AggregatedTransform property), 111
- ALTER_KEY (*butterfree.migrations.database_migration.database_migration.Diff.Kind* attribute), 85
- ALTER_KEY (*butterfree.migrations.database_migration.Diff.Kind* attribute), 86
- ALTER_TYPE (*butterfree.migrations.database_migration.database_migration.Diff.Kind* attribute), 85
- ALTER_TYPE (*butterfree.migrations.database_migration.Diff.Kind* attribute), 86
- apply_migration () (butterfree.migrations.database_migration.database_migration.DatabaseMigration method), 84
- ARRAY_BIGINT (butterfree.constants.data_type.DataType attribute), 33
- ARRAY_BIGINT (butterfree.constants.DataType attribute), 34
- ARRAY_FLOAT (butterfree.constants.data_type.DataType attribute), 33
- ARRAY_FLOAT (butterfree.constants.DataType attribute), 34
- ARRAY_STRING (butterfree.constants.data_type.DataType attribute), 33
- ARRAY_STRING (butterfree.constants.DataType attribute), 34
- assert_column_equality () (in module *butterfree.testing.dataframe*), 96
- assert_dataframe_equality () (in module *butterfree.testing.dataframe*), 96
- aggregation (in module *butterfree.extract.pre_processing*), 49

B

- BasicValidation (*class in butterfree.validations*), 131
- BasicValidation (*class in butterfree.validations.basic_validation*), 130
- BIGINT (*butterfree.constants.data_type.DataType attribute*), 33
- BIGINT (*butterfree.constants.DataType attribute*), 34
- BINARY (*butterfree.constants.data_type.DataType attribute*), 33
- BINARY (*butterfree.constants.DataType attribute*), 34
- BOOLEAN (*butterfree.constants.data_type.DataType attribute*), 33
- BOOLEAN (*butterfree.constants.DataType attribute*), 34
- build() (*butterfree.extract.readers.reader.Reader method*), 55
- butterfree
 - module, 131
- butterfree.clients
 - module, 20
- butterfree.clients.abstract_client
 - module, 16
- butterfree.clients.cassandra_client
 - module, 16
- butterfree.clients.spark_client
 - module, 17
- butterfree.configs
 - module, 33
- butterfree.configs.db
 - module, 28
- butterfree.configs.db.abstract_config
 - module, 23
- butterfree.configs.db.cassandra_config
 - module, 24
- butterfree.configs.db.kafka_config
 - module, 26
- butterfree.configs.db.metastore_config
 - module, 27
- butterfree.configs.environment
 - module, 33
- butterfree.configs.logger
 - module, 33
- butterfree.constants
 - module, 34
- butterfree.constants.columns
 - module, 33
- butterfree.constants.data_type
 - module, 33
- butterfree.constants.migrations
 - module, 34
- butterfree.constants.spark_constants
 - module, 34
- butterfree.constants.window_definitions
 - module, 34
- butterfree.dataframe_service
 - module, 37
- butterfree.dataframe_service.incremental_strategy
 - module, 34
- butterfree.dataframe_service.partitioning
 - module, 36
- butterfree.dataframe_service.repartition
 - module, 36
- butterfree.extract
 - module, 62
- butterfree.extract.pre_processing
 - module, 46
- butterfree.extract.pre_processing.explode_json_colu
 - module, 39
- butterfree.extract.pre_processing.filter_transform
 - module, 41
- butterfree.extract.pre_processing.forward_fill_tra
 - module, 41
- butterfree.extract.pre_processing.pivot_transform
 - module, 43
- butterfree.extract.pre_processing.replace_transform
 - module, 45
- butterfree.extract.readers
 - module, 57
- butterfree.extract.readers.file_reader
 - module, 52
- butterfree.extract.readers.kafka_reader
 - module, 53
- butterfree.extract.readers.reader
 - module, 55
- butterfree.extract.readers.table_reader
 - module, 56
- butterfree.extract.source
 - module, 60
- butterfree.hooks
 - module, 67
- butterfree.hooks.hook
 - module, 65
- butterfree.hooks.hookable_component
 - module, 66
- butterfree.hooks.schema_compatibility
 - module, 64
- butterfree.hooks.schema_compatibility.cassandra_ta
 - module, 63
- butterfree.hooks.schema_compatibility.spark_table_s
 - module, 63
- butterfree.load
 - module, 83
- butterfree.load.processing
 - module, 69
- butterfree.load.processing.json_transform
 - module, 68
- butterfree.load.sink
 - module, 82

butterfree.load.writers module, 77
 butterfree.load.writers.historical_feature_set module, 69
 butterfree.load.writers.online_feature_set module, 72
 butterfree.load.writers.writer module, 75
 butterfree.migrations module, 87
 butterfree.migrations.database_migration module, 86
 butterfree.migrations.database_migration_base module, 84
 butterfree.migrations.database_migration_base_migration module, 84
 butterfree.migrations.database_migration_base_migration_metadata module, 85
 butterfree.pipelines module, 90
 butterfree.pipelines.feature_set_pipeline module, 87
 butterfree.reports module, 95
 butterfree.reports.metadata module, 93
 butterfree.testing module, 97
 butterfree.testing.dataframe module, 96
 butterfree.transform module, 127
 butterfree.transform.aggregated_feature_set module, 120
 butterfree.transform.feature_set module, 124
 butterfree.transform.features module, 100
 butterfree.transform.features.feature module, 97
 butterfree.transform.features.key_feature module, 98
 butterfree.transform.features.timestamp_feature module, 98
 butterfree.transform.transformations module, 110
 butterfree.transform.transformations.aggregated_transform module, 102
 butterfree.transform.transformations.custom_transform module, 103
 butterfree.transform.transformations.h3_transform module, 104
 butterfree.transform.transformations.spark_function module, 106
 butterfree.transform.transformations.sql_expression module, 107
 butterfree.transform.transformations.stack_transform module, 108
 butterfree.transform.transformations.transform_compatibility module, 110
 butterfree.transform.utils module, 119
 butterfree.transform.utils.date_range module, 117
 butterfree.transform.utils.function module, 117
 butterfree.transform.utils.window_spec module, 118
 butterfree.transform.utils.window_spec_base module, 131
 butterfree.transform.utils.window_spec_base_migration module, 131
 butterfree.transformations.basic_validation module, 130
 butterfree.validations.validation module, 130

C

cassandra_client (*butterfree.hooks.schema_compatibility.cassandra_table_schema_compatibility.attribute*), 63
 cassandra_client (*butterfree.hooks.schema_compatibility.CassandraTableSchemaCompatibility.attribute*), 64
 CassandraClient (*class in butterfree.clients*), 20
 CassandraClient (*class in butterfree.clients.cassandra_client*), 16
 CassandraColumn (*class in butterfree.clients.cassandra_client*), 17
 CassandraConfig (*class in butterfree.configs.db*), 29
 CassandraConfig (*class in butterfree.configs.db.cassandra_config*), 24
 CassandraMigration (*class in butterfree.migrations.database_migration*), 86
 CassandraMigration (*class in butterfree.migrations.database_migration.cassandra_migration*), 84
 CassandraTableSchemaCompatibilityHook (*class in butterfree.hooks.schema_compatibility*), 64
 CassandraTableSchemaCompatibilityHook (*class in butterfree.hooks.schema_compatibility.cassandra_table_schema_compatibility*), 63
 check_transform (*butterfree.validations.basic_validation.BasicValidation.method*), 130
 check_transform (*butterfree.validations.basic_validation.BasicValidation.method*), 131
 check_transform (*butterfree.validations.validation.Validation.method*), 130

check_schema () (butterfree.load.writers.historical_feature_store_writer.HistoricalFeatureStoreWriter method), 72
 check_schema () (butterfree.load.writers.HistoricalFeatureStoreWriter method), 78
 check_schema () (butterfree.load.writers.online_feature_store_writer.OnlineFeatureStoreWriter method), 74
 check_schema () (butterfree.load.writers.OnlineFeatureStoreWriter method), 80
 check_schema () (butterfree.load.writers.writer.Writer method), 75
 column (butterfree.dataframe_service.incremental_strategy.IncrementalStrategy attribute), 35
 column (butterfree.dataframe_service.IncrementalStrategy consume () (butterfree.extract.readers.kafka_reader.KafkaReader method), 54
 column (butterfree.migrations.database_migration.database_migration.Diff (butterfree.extract.readers.KafkaReader method), 59
 column (butterfree.migrations.database_migration.Diff consume () (butterfree.extract.readers.reader.Reader method), 55
 column_name (butterfree.clients.cassandra_client.CassandraColumn attribute), 17
 columns (butterfree.clients.cassandra_client.CassandraClient attribute), 17
 columns (butterfree.clients.CassandraClient attribute), 20
 columns () (butterfree.transform.feature_set.FeatureSet property), 126
 columns () (butterfree.transform.FeatureSet property), 128
 columns_names (butterfree.transform.transformations.stack_transform.StackTransform attribute), 109
 columns_names (butterfree.transform.transformations.StackTransform attribute), 115
 conn () (butterfree.clients.abstract_client.AbstractClient property), 16
 conn () (butterfree.clients.AbstractClient property), 20
 conn () (butterfree.clients.cassandra_client.CassandraClient property), 16
 conn () (butterfree.clients.CassandraClient property), 20
 conn () (butterfree.clients.spark_client.SparkClient property), 18
 conn () (butterfree.clients.SparkClient property), 21
 connection_string (butterfree.extract.readers.kafka_reader.KafkaReader attribute), 53
 connection_string (butterfree.extract.readers.KafkaReader attribute), 53
 connection_string (butterfree.extract.readers.KafkaReader attribute), 53
 connection_string (butterfree.transform.aggregated_feature_set.AggregatedFeatureSet attribute), 123
 connection_string (butterfree.transform.feature_set.FeatureSet method), 126
 connection_string (butterfree.transform.FeatureSet method), 128
 consume () (butterfree.extract.readers.file_reader.FileReader method), 53
 consume () (butterfree.extract.readers.kafka_reader.KafkaReader method), 54
 consume () (butterfree.extract.readers.KafkaReader method), 59
 consume () (butterfree.extract.readers.reader.Reader method), 55
 consume () (butterfree.extract.readers.table_reader.TableReader method), 56
 consume () (butterfree.extract.readers.TableReader method), 60
 create_df_from_collection () (in module butterfree.testing.dataframe), 96
 create_query () (butterfree.migrations.database_migration.database_migration.DatabaseMigration method), 84
 create_table () (butterfree.clients.cassandra_client.CassandraClient method), 17
 create_table () (butterfree.clients.CassandraClient method), 20
 create_temporary_view () (butterfree.clients.spark_client.SparkClient static method), 18
 create_temporary_view () (butterfree.clients.SparkClient static method), 21
 CustomTransform (class in butterfree.transform.transformations), 111
 CustomTransform (class in butterfree.transform.transformations.custom_transform), 103

D

data_type (butterfree.transform.utils.Function attribute), 119
 data_type (butterfree.transform.utils.function.Function attribute), 117

free.transform.features.KeyFeature attribute), 101
description (*butterfree.transform.FeatureSet* attribute), 127
description () (*butterfree.transform.feature_set.FeatureSet* property), 126
description () (*butterfree.transform.FeatureSet* property), 129
Diff (class in *butterfree.migrations.database_migration*), 86
Diff (class in *butterfree.migrations.database_migration.database_migration*), 84
Diff.Kind (class in *butterfree.migrations.database_migration*), 86
Diff.Kind (class in *butterfree.migrations.database_migration.database_migration*), 85
DOUBLE (*butterfree.constants.data_type.DataType* attribute), 34
DOUBLE (*butterfree.constants.DataType* attribute), 34
DROP (*butterfree.migrations.database_migration.database_migration* attribute), 85
DROP (*butterfree.migrations.database_migration.Diff.Kind* attribute), 86
dtype (*butterfree.transform.features.Feature* attribute), 100
dtype (*butterfree.transform.features.feature.Feature* attribute), 97
dtype (*butterfree.transform.features.key_feature.KeyFeature* attribute), 98
dtype (*butterfree.transform.features.KeyFeature* attribute), 101
dtype () (*butterfree.transform.features.Feature* property), 100
dtype () (*butterfree.transform.features.feature.Feature* property), 97
E
enable_post_hooks (*butterfree.hooks.hookable_component.HookableComponent* attribute), 66
enable_post_hooks (*butterfree.hooks.HookableComponent* attribute), 67
enable_post_hooks () (*butterfree.hooks.hookable_component.HookableComponent* property), 66
enable_post_hooks () (*butterfree.hooks.HookableComponent* property), 68
enable_pre_hooks (*butterfree.hooks.hookable_component.HookableComponent* attribute), 66
enable_pre_hooks (*butterfree.hooks.HookableComponent* attribute), 67
enable_pre_hooks () (*butterfree.hooks.hookable_component.HookableComponent* property), 66
enable_pre_hooks () (*butterfree.hooks.HookableComponent* property), 68
enable_pre_hooks () (*butterfree.transform.FeatureSet* attribute), 127
enable_pre_hooks () (*butterfree.transform.FeatureSet* property), 129
enable_pre_hooks () (*butterfree.transform.feature_set.FeatureSet* property), 124
entity (*butterfree.transform.feature_set.FeatureSet* attribute), 124
entity () (*butterfree.transform.FeatureSet* attribute), 127
entity () (*butterfree.transform.feature_set.FeatureSet* property), 126
entity () (*butterfree.transform.FeatureSet* property), 129
explode_json_column () (in module *butterfree.extract.pre_processing*), 46
explode_json_column () (in module *butterfree.extract.pre_processing.explode_json_column_transform*), 39
expression (*butterfree.transform.transformations.sql_expression_transform.Diff.Kind* attribute), 108
expression (*butterfree.transform.transformations.SQLExpressionTransform* attribute), 113
extract_partition_values () (in module *butterfree.dataframe_service*), 38
extract_partition_values () (in module *butterfree.dataframe_service.partitioning*), 36
F
Feature (class in *butterfree.transform.features*), 100
Feature (class in *butterfree.transform.features.feature*), 97
feature_set (*butterfree.pipelines.feature_set_pipeline.FeatureSetPipeline* attribute), 87
feature_set (*butterfree.pipelines.FeatureSetPipeline* attribute), 90
feature_set () (*butterfree.pipelines.feature_set_pipeline.FeatureSetPipeline* property), 89
feature_set () (*butterfree.pipelines.FeatureSetPipeline* property), 92
features (*butterfree.transform.feature_set.FeatureSet* attribute), 124
features (*butterfree.transform.FeatureSet* attribute), 127
features () (*butterfree.transform.aggregated_feature_set.AggregatedFeatureSet* property), 123
features () (*butterfree.transform.feature_set.FeatureSet* property), 126

- features() (*butterfree.transform.FeatureSet* property), 129
- features_columns() (*butterfree.transform.aggregated_feature_set.AggregatedFeatureSet* property), 123
- features_columns() (*butterfree.transform.feature_set.FeatureSet* property), 126
- features_columns() (*butterfree.transform.FeatureSet* property), 129
- FeatureSet (class in *butterfree.transform*), 127
- FeatureSet (class in *butterfree.transform.feature_set*), 124
- FeatureSetPipeline (class in *butterfree.pipelines*), 90
- FeatureSetPipeline (class in *butterfree.pipelines.feature_set_pipeline*), 87
- file_system (*butterfree.configs.db.metastore_config.MetastoreConfig* attribute), 27
- file_system (*butterfree.configs.db.MetastoreConfig* attribute), 32
- file_system() (*butterfree.configs.db.metastore_config.MetastoreConfig* property), 28
- file_system() (*butterfree.configs.db.MetastoreConfig* property), 32
- FileReader (class in *butterfree.extract.readers*), 57
- FileReader (class in *butterfree.extract.readers.file_reader*), 52
- fill_column (in module *butterfree.extract.pre_processing*), 47
- fill_column (in module *butterfree.extract.pre_processing.forward_fill_transform*), 41
- filled_column (in module *butterfree.extract.pre_processing*), 47
- filled_column (in module *butterfree.extract.pre_processing.forward_fill_transform*), 41
- filter() (in module *butterfree.extract.pre_processing*), 47
- filter() (in module *butterfree.extract.pre_processing.filter_transform*), 41
- filter_expression (*butterfree.transform.transformations.aggregated_transform.AggregatedTransform* attribute), 102
- filter_expression (*butterfree.transform.transformations.AggregatedTransform* attribute), 111
- filter_latest() (*butterfree.load.writers.online_feature_store_writer.OnlineFeatureStoreWriter* static method), 74
- filter_latest() (*butterfree.load.writers.OnlineFeatureStoreWriter* static method), 80
- filter_with_incremental_strategy() (*butterfree.dataframe_service.incremental_strategy.IncrementalStrategy* method), 35
- filter_with_incremental_strategy() (*butterfree.dataframe_service.IncrementalStrategy* method), 37
- FLOAT (*butterfree.constants.data_type.DataType* attribute), 34
- FLOAT (*butterfree.constants.DataType* attribute), 34
- flush() (*butterfree.load.Sink* method), 83
- flush() (*butterfree.load.sink.Sink* method), 82
- format (*butterfree.extract.readers.file_reader.FileReader* attribute), 52
- format (*butterfree.extract.readers.FileReader* attribute), 57
- format_ (*butterfree.configs.db.cassandra_config.CassandraConfig* attribute), 25
- format_ (*butterfree.configs.db.CassandraConfig* attribute), 29
- format_ (*butterfree.configs.db.kafka_config.KafkaConfig* attribute), 26
- format_ (*butterfree.configs.db.KafkaConfig* attribute), 31
- format_ (*butterfree.configs.db.metastore_config.MetastoreConfig* attribute), 27
- format_ (*butterfree.configs.db.MetastoreConfig* attribute), 32
- format_ () (*butterfree.configs.db.abstract_config.AbstractWriteConfig* property), 24
- format_ () (*butterfree.configs.db.AbstractWriteConfig* property), 28
- format_ () (*butterfree.configs.db.cassandra_config.CassandraConfig* property), 25
- format_ () (*butterfree.configs.db.CassandraConfig* property), 29
- format_ () (*butterfree.configs.db.kafka_config.KafkaConfig* property), 27
- format_ () (*butterfree.configs.db.KafkaConfig* property), 31
- format_ () (*butterfree.configs.db.metastore_config.MetastoreConfig* property), 28
- format_ () (*butterfree.configs.db.MetastoreConfig* property), 32
- format_options (*butterfree.extract.readers.file_reader.FileReader* attribute), 52
- format_options (*butterfree.extract.readers.FileReader* attribute), 57
- in module *butterfree*

`free.extract.pre_processing`), 47

`forward_fill()` (in module `butterfree.extract.pre_processing.forward_fill_transform`), 41

`FrameBoundaries` (class in `butterfree.transform.utils.window_spec`), 118

`from_column` (`butterfree.transform.features.Feature` attribute), 100

`from_column` (`butterfree.transform.features.feature.Feature` attribute), 97

`from_column` (`butterfree.transform.features.key_feature.KeyFeature` attribute), 98

`from_column` (`butterfree.transform.features.KeyFeature` attribute), 101

`from_column` (`butterfree.transform.features.timestamp_feature.TimestampFeature` attribute), 99

`from_column` (`butterfree.transform.features.TimestampFeature` attribute), 101

`from_column()` (`butterfree.transform.features.Feature` property), 100

`from_column()` (`butterfree.transform.features.feature.Feature` property), 98

`from_milliseconds()` (`butterfree.dataframe_service.incremental_strategy.IncrementalStrategy` method), 35

`from_milliseconds()` (`butterfree.dataframe_service.IncrementalStrategy` method), 37

`from_ms` (`butterfree.transform.features.timestamp_feature.TimestampFeature` attribute), 99

`from_ms` (`butterfree.transform.features.TimestampFeature` attribute), 101

`from_string()` (`butterfree.dataframe_service.incremental_strategy.IncrementalStrategy` method), 35

`from_string()` (`butterfree.dataframe_service.IncrementalStrategy` method), 38

`from_year_month_day_partitions()` (`butterfree.dataframe_service.incremental_strategy.IncrementalStrategy` method), 35

`from_year_month_day_partitions()` (`butterfree.dataframe_service.IncrementalStrategy` method), 38

`func` (`butterfree.transform.utils.Function` attribute), 119

`func` (`butterfree.transform.utils.function.Function` attribute), 117

`func()` (`butterfree.transform.utils.Function` property), 119

`func()` (`butterfree.transform.utils.function.Function` property), 118

`function` (`butterfree.transform.transformations.spark_function_transform` attribute), 106

`function` (`butterfree.transform.transformations.SparkFunctionTransform` attribute), 114

`Function` (class in `butterfree.transform.utils`), 119

`Function` (class in `butterfree.transform.utils.function`), 117

`functions` (`butterfree.transform.transformations.aggregated_transform` attribute), 102

`functions` (`butterfree.transform.transformations.AggregatedTransform` attribute), 111

G

`get()` (`butterfree.transform.utils.Window` method), 119

`get()` (`butterfree.transform.utils.window_spec.FrameBoundaries` method), 118

`get()` (`butterfree.transform.utils.window_spec.Window` method), 118

`get_date_range()` (in module `butterfree.transform.utils.date_range`), 117

`get_db_schema()` (`butterfree.load.writers.online_feature_store_writer.OnlineFeatureStoreWriter` method), 75

`get_db_schema()` (`butterfree.load.writers.OnlineFeatureStoreWriter` method), 81

`get_expression()` (`butterfree.dataframe_service.incremental_strategy.IncrementalStrategy` method), 35

`get_expression()` (`butterfree.dataframe_service.IncrementalStrategy` method), 38

`get_name()` (`butterfree.transform.utils.Window` method), 119

`get_name()` (`butterfree.transform.utils.window_spec.Window` method), 118

`get_options()` (`butterfree.configs.db.cassandra_config.CassandraConfig` method), 25

`get_options()` (`butterfree.configs.db.CassandraConfig` method), 30

`get_options()` (`butterfree.configs.db.kafka_config.KafkaConfig` method), 27

`get_options()` (`butterfree.configs.db.KafkaConfig` method), 31

`get_options()` (`butterfree.configs.db.metastore_config.MetastoreConfig` method), 28

- `get_options()` (*butterfree.configs.db.MetastoreConfig* method), 32
`get_output_columns()` (*butterfree.transform.features.Feature* method), 100
`get_output_columns()` (*butterfree.transform.features.feature.Feature* method), 98
`get_path_with_partitions()` (*butterfree.configs.db.metastore_config.MetastoreConfig* method), 28
`get_path_with_partitions()` (*butterfree.configs.db.MetastoreConfig* method), 32
`get_schema()` (*butterfree.clients.abstract_client.AbstractClient* method), 16
`get_schema()` (*butterfree.clients.AbstractClient* method), 20
`get_schema()` (*butterfree.clients.cassandra_client.CassandraClient* method), 17
`get_schema()` (*butterfree.clients.CassandraClient* method), 21
`get_schema()` (*butterfree.clients.spark_client.SparkClient* method), 18
`get_schema()` (*butterfree.clients.SparkClient* method), 21
`get_schema()` (*butterfree.transform.aggregated_feature_set.AggregatedFeatureSet* method), 123
`get_schema()` (*butterfree.transform.feature_set.FeatureSet* method), 126
`get_schema()` (*butterfree.transform.FeatureSet* method), 129
`get_variable()` (in module *butterfree.configs.environment*), 33
`group_by_columns` (in module *butterfree.extract.pre_processing*), 49
`group_by_columns` (in module *butterfree.extract.pre_processing.pivot_transform*), 43
- ## H
- `h3_resolutions` (*butterfree.transform.transformations.h3_transform.H3HashTransform* attribute), 105
H3HashTransform (class in *butterfree.transform.transformations.h3_transform*), 104
- HistoricalFeatureStoreWriter* (class in *butterfree.load.writers*), 77
HistoricalFeatureStoreWriter (class in *butterfree.load.writers.historical_feature_store_writer*), 69
Hook (class in *butterfree.hooks*), 67
Hook (class in *butterfree.hooks.hook*), 65
HookableComponent (class in *butterfree.hooks*), 67
HookableComponent (class in *butterfree.hooks.hookable_component*), 66
`host` (*butterfree.clients.cassandra_client.CassandraClient* attribute), 16
`host` (*butterfree.clients.CassandraClient* attribute), 20
`host` (*butterfree.configs.db.cassandra_config.CassandraConfig* attribute), 25
`host` (*butterfree.configs.db.CassandraConfig* attribute), 29
`host()` (*butterfree.configs.db.cassandra_config.CassandraConfig* property), 25
`host()` (*butterfree.configs.db.CassandraConfig* property), 30
- ## I
- `id` (*butterfree.extract.readers.file_reader.FileReader* attribute), 52
`id` (*butterfree.extract.readers.FileReader* attribute), 57
`id` (*butterfree.extract.readers.kafka_reader.KafkaReader* attribute), 53
`id` (*butterfree.extract.readers.KafkaReader* attribute), 58
`id` (*butterfree.extract.readers.reader.Reader* attribute), 55
`id` (*butterfree.extract.readers.table_reader.TableReader* attribute), 56
`id` (*butterfree.extract.readers.TableReader* attribute), 59
IncrementalStrategy (class in *butterfree.dataframe_service*), 37
IncrementalStrategy (class in *butterfree.dataframe_service.incremental_strategy*), 34
`input()` (*butterfree.validations.validation.Validation* method), 130
INTEGER (*butterfree.constants.data_type.DataType* attribute), 34
INTEGER (*butterfree.constants.DataType* attribute), 34
`is_regex` (*butterfree.transform.transformations.stack_transform.StackTransform* attribute), 109
`is_regex` (*butterfree.transform.transformations.StackTransform* attribute), 115
- ## J
- `json_transform()` (in module *butterfree.load.processing*), 69

`json_transform()` (in module `butterfree.load.processing.json_transform`), 68

K

`KAFKA_COLUMNS` (`butterfree.extract.readers.kafka_reader.KafkaReader` attribute), 54

`KAFKA_COLUMNS` (`butterfree.extract.readers.KafkaReader` attribute), 59

`kafka_connection_string` (`butterfree.configs.db.kafka_config.KafkaConfig` attribute), 26

`kafka_connection_string` (`butterfree.configs.db.KafkaConfig` attribute), 30

`kafka_connection_string()` (`butterfree.configs.db.kafka_config.KafkaConfig` property), 27

`kafka_connection_string()` (`butterfree.configs.db.KafkaConfig` property), 31

`kafka_topic` (`butterfree.configs.db.kafka_config.KafkaConfig` attribute), 26

`kafka_topic` (`butterfree.configs.db.KafkaConfig` attribute), 30

`kafka_topic()` (`butterfree.configs.db.kafka_config.KafkaConfig` property), 27

`kafka_topic()` (`butterfree.configs.db.KafkaConfig` property), 31

`KafkaConfig` (class in `butterfree.configs.db`), 30

`KafkaConfig` (class in `butterfree.configs.db.kafka_config`), 26

`KafkaReader` (class in `butterfree.extract.readers`), 58

`KafkaReader` (class in `butterfree.extract.readers.kafka_reader`), 53

`KeyFeature` (class in `butterfree.transform.features`), 100

`KeyFeature` (class in `butterfree.transform.features.key_feature`), 98

`keys` (`butterfree.transform.feature_set.FeatureSet` attribute), 124

`keys` (`butterfree.transform.FeatureSet` attribute), 127

`keys()` (`butterfree.transform.feature_set.FeatureSet` property), 126

`keys()` (`butterfree.transform.FeatureSet` property), 129

`keys_columns()` (`butterfree.transform.feature_set.FeatureSet` property), 126

`keys_columns()` (`butterfree.transform.FeatureSet` property), 129

`keyspace` (`butterfree.clients.cassandra_client.CassandraClient` attribute), 16

`keyspace` (`butterfree.clients.CassandraClient` attribute), 20

`keyspace` (`butterfree.configs.db.cassandra_config.CassandraConfig` attribute), 25

`keyspace` (`butterfree.configs.db.CassandraConfig` attribute), 29

`keyspace()` (`butterfree.configs.db.cassandra_config.CassandraConfig` property), 25

`keyspace()` (`butterfree.configs.db.CassandraConfig` property), 30

`kind` (`butterfree.migrations.database_migration.database_migration.Diff` attribute), 85

`kind` (`butterfree.migrations.database_migration.Diff` attribute), 86

L

`lat` (in module `butterfree.transform.transformations.h3_transform`), 106

`lat_column` (`butterfree.transform.transformations.h3_transform.H3Hash` attribute), 105

`lng` (in module `butterfree.transform.transformations.h3_transform`), 106

`lng_column` (`butterfree.transform.transformations.h3_transform.H3Hash` attribute), 105

`local_dc()` (`butterfree.configs.db.cassandra_config.CassandraConfig` property), 25

`local_dc()` (`butterfree.configs.db.CassandraConfig` property), 30

M

`mask` (`butterfree.transform.features.timestamp_feature.TimestampFeature` attribute), 99

`mask` (`butterfree.transform.features.TimestampFeature` attribute), 101

`Metadata` (class in `butterfree.reports`), 95

`Metadata` (class in `butterfree.reports.metadata`), 93

`MetastoreConfig` (class in `butterfree.configs.db`), 31

`MetastoreConfig` (class in `butterfree.configs.db.metastore_config`), 27

`MetastoreMigration` (class in `butterfree.migrations.database_migration`), 86

`MetastoreMigration` (class in `butterfree.migrations.database_migration.metastore_migration`), 85

`mock_type` (in module `butterfree.extract.pre_processing`), 49

`mock_type` (in module `butterfree.extract.pre_processing.pivot_transform`), 43

`mock_value` (in module `butterfree.extract.pre_processing`), 49

`mock_value` (in module `butterfree.extract.pre_processing.pivot_transform`), 43
`mode` (`butterfree.configs.db.cassandra_config.CassandraConfig` attribute), 25
`mode` (`butterfree.configs.db.CassandraConfig` attribute), 29
`mode` (`butterfree.configs.db.kafka_config.KafkaConfig` attribute), 26
`mode` (`butterfree.configs.db.KafkaConfig` attribute), 31
`mode` (`butterfree.configs.db.metastore_config.MetastoreConfig` attribute), 27
`mode` (`butterfree.configs.db.MetastoreConfig` attribute), 32
`mode` () (`butterfree.configs.db.abstract_config.AbstractWriteConfig` property), 24
`mode` () (`butterfree.configs.db.AbstractWriteConfig` property), 28
`mode` () (`butterfree.configs.db.cassandra_config.CassandraConfig` property), 25
`mode` () (`butterfree.configs.db.CassandraConfig` property), 30
`mode` () (`butterfree.configs.db.kafka_config.KafkaConfig` property), 27
`mode` () (`butterfree.configs.db.KafkaConfig` property), 31
`mode` () (`butterfree.configs.db.metastore_config.MetastoreConfig` property), 28
`mode` () (`butterfree.configs.db.MetastoreConfig` property), 32
module
 `butterfree`, 131
 `butterfree.clients`, 20
 `butterfree.clients.abstract_client`, 16
 `butterfree.clients.cassandra_client`, 16
 `butterfree.clients.spark_client`, 17
 `butterfree.configs`, 33
 `butterfree.configs.db`, 28
 `butterfree.configs.db.abstract_config`, 23
 `butterfree.configs.db.cassandra_config`, 24
 `butterfree.configs.db.kafka_config`, 26
 `butterfree.configs.db.metastore_config`, 27
 `butterfree.configs.environment`, 33
 `butterfree.configs.logger`, 33
 `butterfree.constants`, 34
 `butterfree.constants.columns`, 33
 `butterfree.constants.data_type`, 33
 `butterfree.constants.migrations`, 34
 `butterfree.constants.spark_constants`, 34
 `butterfree.constants.window_definitions`, 34
 `butterfree.dataframe_service`, 37
 `butterfree.dataframe_service.incremental_strategy`, 34
 `butterfree.dataframe_service.partitioning`, 36
 `butterfree.dataframe_service.repartition`, 36
 `butterfree.extract`, 62
 `butterfree.extract.pre_processing`, 46
 `butterfree.extract.pre_processing.explode_json`, 39
 `butterfree.extract.pre_processing.filter_transform`, 41
 `butterfree.extract.pre_processing.forward_fill`, 41
 `butterfree.extract.pre_processing.pivot_transform`, 43
 `butterfree.extract.pre_processing.replace_transform`, 45
 `butterfree.extract.readers`, 57
 `butterfree.extract.readers.file_reader`, 52
 `butterfree.extract.readers.kafka_reader`, 53
 `butterfree.extract.readers.reader`, 55
 `butterfree.extract.readers.table_reader`, 56
 `butterfree.extract.source`, 60
 `butterfree.hooks`, 67
 `butterfree.hooks.hook`, 65
 `butterfree.hooks.hookable_component`, 66
 `butterfree.hooks.schema_compatibility`, 64
 `butterfree.hooks.schema_compatibility.cassandra`, 63
 `butterfree.hooks.schema_compatibility.spark_table`, 63
 `butterfree.load`, 83
 `butterfree.load.processing`, 69
 `butterfree.load.processing.json_transform`, 68
 `butterfree.load.sink`, 82
 `butterfree.load.writers`, 77
 `butterfree.load.writers.historical_feature_store`, 69
 `butterfree.load.writers.online_feature_store_writer`, 72
 `butterfree.load.writers.writer`, 75

butterfree.migrations, 87
 butterfree.migrations.database_migration, 86
 butterfree.migrations.database_migration.cassandra_migration, 84
 butterfree.migrations.database_migration.data_migration, 84
 butterfree.migrations.database_migration.metadata_migration, 85
 butterfree.pipelines, 90
 butterfree.pipelines.feature_set_pipeline, 87
 butterfree.reports, 95
 butterfree.reports.metadata, 93
 butterfree.testing, 97
 butterfree.testing.dataframe, 96
 butterfree.transform, 127
 butterfree.transform.aggregated_feature_set, 120
 butterfree.transform.feature_set, 124
 butterfree.transform.features, 100
 butterfree.transform.features.feature, 97
 butterfree.transform.features.key_feature, 98
 butterfree.transform.features.timestamp_feature, 98
 butterfree.transform.transformations, 110
 butterfree.transform.transformations.aggregated_transform, 102
 butterfree.transform.transformations.custom_transform, 103
 butterfree.transform.transformations.h3_transform, 104
 butterfree.transform.transformations.spark_function_transform, 106
 butterfree.transform.transformations.sql_expression_transform, 107
 butterfree.transform.transformations.stack_transform, 108
 butterfree.transform.transformations.transform_component, 110
 butterfree.transform.utils, 119
 butterfree.transform.utils.date_range, 117
 butterfree.transform.utils.function, 117
 butterfree.transform.utils.window_spec, 118
 butterfree.validations, 131
 butterfree.validations.basic_validation, 130
 butterfree.validations.validation, 130
 OnlineFeatureStoreWriter (class in butterfree.load.writers), 79
 OnlineFeatureStoreWriter (class in butterfree.load.writers.online_feature_store_writer), 79
 order_by (in module butterfree.extract.pre_processing), 47
 order_by (in module butterfree.extract.pre_processing.forward_fill_transform), 41
 output_columns () (butterfree.transform.transformations.aggregated_transform.AggregatedTransform property), 103
 output_columns () (butterfree.transform.transformations.AggregatedTransform property), 111
 output_columns () (butterfree.transform.transformations.custom_transform.CustomTransform property), 104
 output_columns () (butterfree.transform.transformations.CustomTransform property), 112
 output_columns () (butterfree.transform.transformations.h3_transform.H3HashTransform property), 105
 output_columns () (butterfree.transform.transformations.spark_function_transform.SparkFunctionTransform property), 107

output_columns() (butterfree.transform.transformations.SparkFunctionTransform property), 115
 output_columns() (butterfree.transform.transformations.sql_expression_transform.SQLExpressionTransform property), 108
 output_columns() (butterfree.transform.transformations.SQLExpressionTransform property), 113
 output_columns() (butterfree.transform.transformations.stack_transform.StackTransform property), 110
 output_columns() (butterfree.transform.transformations.StackTransform property), 116
 output_columns() (butterfree.transform.transformations.transform_component.TransformComponent property), 110
 output_columns() (butterfree.transform.transformations.TransformComponent property), 116

P

parent (butterfree.transform.transformations.transform_component.TransformComponent attribute), 110
 parent (butterfree.transform.transformations.TransformComponent attribute), 116
 parent () (butterfree.transform.transformations.transform_component.TransformComponent property), 110
 parent () (butterfree.transform.transformations.TransformComponent property), 117
 PARTITION_BY (butterfree.load.writers.historical_feature_store_writer.HistoricalFeatureStoreWriter attribute), 72
 PARTITION_BY (butterfree.load.writers.HistoricalFeatureStoreWriter attribute), 78
 partition_by (in module butterfree.extract.pre_processing), 47
 partition_by (in module butterfree.extract.pre_processing.forward_fill_transform), 41
 password (butterfree.clients.cassandra_client.CassandraClient attribute), 16
 password (butterfree.clients.CassandraClient attribute), 20
 password (butterfree.configs.db.cassandra_config.CassandraConfig attribute), 24
 password (butterfree.configs.db.CassandraConfig attribute), 29
 password () (butterfree.configs.db.cassandra_config.CassandraConfig property), 25
 password () (butterfree.configs.db.CassandraConfig property), 30
 path (butterfree.configs.db.metastore_config.MetastoreConfig attribute), 27
 path (butterfree.configs.db.MetastoreConfig attribute), 32
 path (butterfree.extract.readers.FileReader attribute), 57
 path () (butterfree.configs.db.metastore_config.MetastoreConfig property), 28
 pipeline (butterfree.reports.Metadata attribute), 95
 pipeline (butterfree.reports.metadata.Metadata attribute), 93
 pivot () (in module butterfree.extract.pre_processing), 49
 pivot () (in module butterfree.extract.pre_processing.pivot_transform), 43
 pivot_column (in module butterfree.extract.pre_processing), 49
 pivot_column (in module butterfree.extract.pre_processing.pivot_transform), 43
 post_hooks (butterfree.hooks.hookable_component.HookableComponent attribute), 66
 post_hooks (butterfree.hooks.HookableComponent attribute), 67
 post_hooks () (butterfree.hooks.hookable_component.HookableComponent property), 66
 post_hooks () (butterfree.hooks.HookableComponent property), 68
 pre_hooks (butterfree.hooks.hookable_component.HookableComponent attribute), 66
 pre_hooks (butterfree.hooks.HookableComponent attribute), 67
 pre_hooks () (butterfree.hooks.hookable_component.HookableComponent property), 66
 pre_hooks () (butterfree.hooks.HookableComponent property), 68
 primary_key (butterfree.clients.cassandra_client.CassandraColumn attribute), 17
 query (butterfree.clients.cassandra_client.CassandraClient attribute), 17
 query (butterfree.clients.CassandraClient attribute), 21
 query (butterfree.extract.Source attribute), 62
 query (butterfree.extract.source.Source attribute), 60

R

- `read()` (*butterfree.clients.spark_client.SparkClient* method), 18
- `read()` (*butterfree.clients.SparkClient* method), 21
- `read_consistency_level` (*butterfree.configs.db.cassandra_config.CassandraConfig* attribute), 25
- `read_consistency_level` (*butterfree.configs.db.CassandraConfig* attribute), 29
- `read_consistency_level()` (*butterfree.configs.db.cassandra_config.CassandraConfig* property), 25
- `read_consistency_level()` (*butterfree.configs.db.CassandraConfig* property), 30
- `read_table()` (*butterfree.clients.spark_client.SparkClient* method), 18
- `read_table()` (*butterfree.clients.SparkClient* method), 22
- `Reader` (class in *butterfree.extract.readers.reader*), 55
- `readers` (*butterfree.extract.Source* attribute), 62
- `readers` (*butterfree.extract.source.Source* attribute), 60
- `repartition_df()` (in module *butterfree.dataframe_service*), 38
- `repartition_df()` (in module *butterfree.dataframe_service.repartition*), 36
- `repartition_sort_df()` (in module *butterfree.dataframe_service*), 39
- `repartition_sort_df()` (in module *butterfree.dataframe_service.repartition*), 36
- `replace()` (in module *butterfree.extract.pre_processing*), 51
- `replace()` (in module *butterfree.extract.pre_processing.replace_transform*), 45
- `resolution` (in module *butterfree.transform.transformations.h3_transform*), 106
- `run()` (*butterfree.hooks.Hook* method), 67
- `run()` (*butterfree.hooks.hook.Hook* method), 65
- `run()` (*butterfree.hooks.schema_compatibility.cassandra_table_schema_compatibility_hook.CassandraTableSchemaCompatibilityHook* method), 63
- `run()` (*butterfree.hooks.schema_compatibility.CassandraTableSchemaCompatibilityHook* method), 64
- `run()` (*butterfree.hooks.schema_compatibility.spark_table_schema_compatibility_hook.SparkTableSchemaCompatibilityHook* method), 64
- `run()` (*butterfree.hooks.schema_compatibility.SparkTableSchemaCompatibilityHook* method), 65
- `run()` (*butterfree.pipelines.feature_set_pipeline.FeatureSetPipeline* method), 89
- `run()` (*butterfree.pipelines.FeatureSetPipeline* method), 92
- `run_for_date()` (*butterfree.pipelines.feature_set_pipeline.FeatureSetPipeline* method), 90
- `run_for_date()` (*butterfree.pipelines.FeatureSetPipeline* method), 93
- `run_post_hooks()` (*butterfree.hooks.hookable_component.HookableComponent* method), 66
- `run_post_hooks()` (*butterfree.hooks.HookableComponent* method), 68
- `run_pre_hooks()` (*butterfree.hooks.hookable_component.HookableComponent* method), 66
- `run_pre_hooks()` (*butterfree.hooks.HookableComponent* method), 68

S

- `save` (*butterfree.reports.Metadata* attribute), 95
- `save` (*butterfree.reports.metadata.Metadata* attribute), 93
- `schema` (*butterfree.extract.readers.file_reader.FileReader* attribute), 52
- `schema` (*butterfree.extract.readers.FileReader* attribute), 57
- `sink` (*butterfree.pipelines.feature_set_pipeline.FeatureSetPipeline* attribute), 87
- `sink` (*butterfree.pipelines.FeatureSetPipeline* attribute), 90
- `Sink` (class in *butterfree.load*), 83
- `Sink` (class in *butterfree.load.sink*), 82
- `sink()` (*butterfree.pipelines.feature_set_pipeline.FeatureSetPipeline* property), 90
- `sink()` (*butterfree.pipelines.FeatureSetPipeline* property), 93
- `source` (*butterfree.pipelines.feature_set_pipeline.FeatureSetPipeline* attribute), 87
- `source` (*butterfree.pipelines.FeatureSetPipeline* attribute), 90
- `Source` (class in *butterfree.extract*), 62
- `Source` (class in *butterfree.extract.source*), 60
- `source()` (*butterfree.pipelines.feature_set_pipeline.FeatureSetPipeline* property), 90
- `source()` (*butterfree.pipelines.FeatureSetPipeline* property), 93
- `spark_client` (*butterfree.hooks.schema_compatibility.spark_table_schema_compatibility_hook.SparkTableSchemaCompatibilityHook* attribute), 64
- `spark_client` (*butterfree.hooks.schema_compatibility.SparkTableSchemaCompatibilityHook* attribute), 64
- `spark_client` (*butterfree.hooks.schema_compatibility.spark_table_schema_compatibility_hook.SparkTableSchemaCompatibilityHook* attribute), 65
- `spark_client` (*butterfree.hooks.schema_compatibility.SparkTableSchemaCompatibilityHook* attribute), 65

`free.configs.db.CassandraConfig` property), 30

`stream_processing_time()` (`butterfree.configs.db.kafka_config.KafkaConfig` property), 27

`stream_processing_time()` (`butterfree.configs.db.KafkaConfig` property), 31

STRING (`butterfree.constants.data_type.DataType` attribute), 34

STRING (`butterfree.constants.DataType` attribute), 34

T

table (`butterfree.clients.abstract_client.AbstractClient` attribute), 16

table (`butterfree.clients.AbstractClient` attribute), 20

table (`butterfree.clients.cassandra_client.CassandraClient` attribute), 17

table (`butterfree.clients.CassandraClient` attribute), 21

table (`butterfree.clients.spark_client.SparkClient` attribute), 18

table (`butterfree.clients.SparkClient` attribute), 21

table (`butterfree.extract.readers.table_reader.TableReader` attribute), 56

table (`butterfree.extract.readers.TableReader` attribute), 59

table (`butterfree.hooks.schema_compatibility.cassandra_table_schema_compatibility_hook.CassandraTableSchemaCompatibilityHook` attribute), 63

table (`butterfree.hooks.schema_compatibility.CassandraTableSchemaCompatibilityHook` attribute), 64

table (`butterfree.hooks.schema_compatibility.spark_table_schema_compatibility_hook.SparkTableSchemaCompatibilityHook` attribute), 64

table (`butterfree.hooks.schema_compatibility.SparkTableSchemaCompatibilityHook` attribute), 65

TableReader (class in `butterfree.extract.readers`), 59

TableReader (class in `butterfree.extract.readers.table_reader`), 56

TIMESTAMP (`butterfree.constants.data_type.DataType` attribute), 34

TIMESTAMP (`butterfree.constants.DataType` attribute), 34

timestamp (`butterfree.transform.feature_set.FeatureSet` attribute), 124

timestamp (`butterfree.transform.FeatureSet` attribute), 127

timestamp () (`butterfree.transform.feature_set.FeatureSet` property), 127

timestamp () (`butterfree.transform.FeatureSet` property), 129

timestamp_column () (`butterfree.transform.feature_set.FeatureSet` property), 127

timestamp_column () (`butterfree.transform.FeatureSet` property), 129

TimestampFeature (class in `butterfree.transform.features`), 101

TimestampFeature (class in `butterfree.transform.features.timestamp_feature`), 98

to_json () (`butterfree.reports.Metadata` method), 96

to_json () (`butterfree.reports.metadata.Metadata` method), 94

to_markdown () (`butterfree.reports.Metadata` method), 96

to_markdown () (`butterfree.reports.metadata.Metadata` method), 94

topic (`butterfree.extract.readers.kafka_reader.KafkaReader` attribute), 53

topic (`butterfree.extract.readers.KafkaReader` attribute), 58

topic_options (`butterfree.extract.readers.kafka_reader.KafkaReader` attribute), 53

topic_options (`butterfree.extract.readers.KafkaReader` attribute), 58

transform () (`butterfree.transform.features.Feature` method), 100

transform () (`butterfree.transform.features.feature.Feature` method), 98

transform () (`butterfree.transform.features.timestamp_feature.TimestampFeature` method), 99

transform () (`butterfree.transform.features.TimestampFeature` method), 101

transform () (`butterfree.transform.transformations.aggregated_transform.AggregatedTransform` method), 103

transform () (`butterfree.transform.transformations.AggregatedTransform` method), 111

transform () (`butterfree.transform.transformations.custom_transform.CustomTransform` method), 104

transform () (`butterfree.transform.transformations.CustomTransform` method), 112

transform () (`butterfree.transform.transformations.h3_transform.H3HashTransform` method), 106

transform () (`butterfree.transform.transformations.spark_function_transform.SparkFunctionTransform` method), 107

transform () (`butterfree.transform.transformations.SparkFunctionTransform` method), 107

- method*), 115
- `transform()` (*butterfree.transform.transformations.sql_expression_transform.SQLExpressionTransform* *method*), 108
- `transform()` (*butterfree.transform.transformations.custom_transform.CustomTransform* *method*), 113
- `transform()` (*butterfree.transform.transformations.stack_transform.StackTransform* *method*), 110
- `transform()` (*butterfree.transform.transformations.StackTransform* *method*), 116
- `transform()` (*butterfree.transform.transformations.transform_component.TransformComponent* *method*), 110
- `transform()` (*butterfree.transform.transformations.TransformComponent* *method*), 117
- `transformation` (*butterfree.transform.features.Feature* *attribute*), 100
- `transformation` (*butterfree.transform.features.feature.Feature* *attribute*), 97
- `transformation` (*butterfree.transform.features.key_feature.KeyFeature* *attribute*), 98
- `transformation` (*butterfree.transform.features.KeyFeature* *attribute*), 101
- `transformation` (*butterfree.transform.features.timestamp_feature.TimestampFeature* *attribute*), 99
- `transformation` (*butterfree.transform.features.TimestampFeature* *attribute*), 101
- `transformation()` (*butterfree.transform.features.Feature* *property*), 100
- `transformation()` (*butterfree.transform.features.feature.Feature* *property*), 98
- `transformations` (*butterfree.extract.readers.reader.Reader* *attribute*), 55
- `TransformComponent` (*class* in *butterfree.transform.transformations*), 116
- `TransformComponent` (*class* in *butterfree.transform.transformations.transform_component*), 110
- `transformer` (*butterfree.transform.transformations.custom_transform.CustomTransform* *attribute*), 103
- `transformer` (*butterfree.transform.transformations.CustomTransform* *attribute*), 103
- `transformer()` (*butterfree.transform.transformations.custom_transform.CustomTransform* *property*), 104
- `transformer()` (*butterfree.transform.transformations.CustomTransform* *property*), 113
- `translate()` (*butterfree.configs.db.abstract_config.AbstractWriteConfig* *method*), 24
- `translate()` (*butterfree.configs.db.AbstractWriteConfig* *method*), 24
- `translate()` (*butterfree.configs.db.cassandra_config.CassandraConfig* *method*), 26
- `translate()` (*butterfree.configs.db.CassandraConfig* *method*), 30
- `translate()` (*butterfree.configs.db.kafka_config.KafkaConfig* *method*), 27
- `translate()` (*butterfree.configs.db.KafkaConfig* *method*), 31
- `translate()` (*butterfree.configs.db.metastore_config.MetastoreConfig* *method*), 28
- `translate()` (*butterfree.configs.db.MetastoreConfig* *method*), 32
- `type` (*butterfree.clients.cassandra_client.CassandraColumn* *attribute*), 17
- `UnspecifiedVariableError`, 33
- `user` (*butterfree.clients.cassandra_client.CassandraClient* *attribute*), 16
- `user` (*butterfree.clients.CassandraClient* *attribute*), 20
- `username` (*butterfree.configs.db.cassandra_config.CassandraConfig* *attribute*), 24
- `username` (*butterfree.configs.db.CassandraConfig* *attribute*), 29
- `username()` (*butterfree.configs.db.cassandra_config.CassandraConfig* *property*), 26
- `username()` (*butterfree.configs.db.CassandraConfig* *property*), 30
- ## V
- `validate()` (*butterfree.load.Sink* *method*), 83
- `validate()` (*butterfree.load.sink.Sink* *method*), 82
- `validate()` (*butterfree.load.writers.historical_feature_store_writer.HistoricalFeatureStoreWriter* *method*), 72
- `validate()` (*butterfree.load.writers.HistoricalFeatureStoreWriter* *method*), 78

validate() (*butterfree.load.writers.online_feature_store_writer.OnlineFeatureStoreWriter* method), 75
 validate() (*butterfree.load.writers.OnlineFeatureStoreWriter* property), 118
 validate() (*butterfree.load.writers.writer.Writer* method), 76
 validate_column_ts() (*butterfree.validations.basic_validation.BasicValidation* method), 130
 validate_column_ts() (*butterfree.validations.BasicValidation* method), 131
 validate_df_is_empty() (*butterfree.validations.basic_validation.BasicValidation* method), 130
 validate_df_is_empty() (*butterfree.validations.BasicValidation* method), 131
 validate_df_is_spark_df() (*butterfree.validations.basic_validation.BasicValidation* method), 130
 validate_df_is_spark_df() (*butterfree.validations.BasicValidation* method), 131
 validation (*butterfree.load.Sink* attribute), 83
 validation (*butterfree.load.sink.Sink* attribute), 82
 Validation (class in *butterfree.validations.validation*), 130
 validation() (*butterfree.load.Sink* property), 83
 validation() (*butterfree.load.sink.Sink* property), 82
 validation_threshold (*butterfree.load.writers.historical_feature_store_writer.HistoricalFeatureStoreWriter* attribute), 70
 validation_threshold (*butterfree.load.writers.HistoricalFeatureStoreWriter* attribute), 77
 value (*butterfree.migrations.database_migration.database_migration.Difference* attribute), 85
 value (*butterfree.migrations.database_migration.Diff* attribute), 86
 value_schema (*butterfree.extract.readers.kafka_reader.KafkaReader* attribute), 53
 value_schema (*butterfree.extract.readers.KafkaReader* attribute), 58
 variable_name (*butterfree.configs.environment.UnspecifiedVariableError* attribute), 33
 Window (class in *butterfree.transform.utils*), 119
 Window (class in *butterfree.transform.utils.window_spec*), 118
 window_unit() (*butterfree.transform.utils.window_spec.FrameBoundaries* property), 118
 with() (*butterfree.extract.readers.reader.Reader* method), 55
 with() (*butterfree.load.writers.writer.Writer* method), 76
 with_distinct() (*butterfree.transform.aggregated_feature_set.AggregatedFeatureSet* method), 123
 with_forward_fill (in module *butterfree.extract.pre_processing*), 49
 with_forward_fill (in module *butterfree.extract.pre_processing.pivot_transform*), 43
 with_incremental_strategy() (*butterfree.extract.readers.reader.Reader* method), 55
 with_pivot() (*butterfree.transform.aggregated_feature_set.AggregatedFeatureSet* method), 124
 with_stack() (*butterfree.transform.transformations.h3_transform.H3HashTransform* method), 106
 with_window() (*butterfree.transform.transformations.spark_function_transform.SparkFunctionTransform* method), 107
 with_window() (*butterfree.transform.transformations.SparkFunctionTransform* method), 115
 with_windows() (*butterfree.transform.aggregated_feature_set.AggregatedFeatureSet* method), 124
 write() (*butterfree.load.writers.HistoricalFeatureStoreWriter* method), 79
 write() (*butterfree.load.writers.online_feature_store_writer.OnlineFeatureStoreWriter* method), 75
 write() (*butterfree.load.writers.OnlineFeatureStoreWriter* method), 81
 write() (*butterfree.load.writers.writer.Writer* method), 76
 write_consistency_level (*butterfree.configs.db.cassandra_config.CassandraConfig* attribute), 25
 write_consistency_level (*butterfree.configs.db.CassandraConfig* attribute), 29
 write_consistency_level() (*butterfree.configs.db.cassandra_config.CassandraConfig* attribute), 25

W

property), 26

`write_consistency_level()` (*butterfree.configs.db.CassandraConfig* *property*), 30

`write_dataframe()` (*butterfree.clients.spark_client.SparkClient* *static method*), 18

`write_dataframe()` (*butterfree.clients.SparkClient* *static method*), 22

`write_stream()` (*butterfree.clients.spark_client.SparkClient* *method*), 19

`write_stream()` (*butterfree.clients.SparkClient* *method*), 22

`write_table()` (*butterfree.clients.spark_client.SparkClient* *static method*), 19

`write_table()` (*butterfree.clients.SparkClient* *static method*), 23

`write_to_entity` (*butterfree.load.writers.online_feature_store_writer.OnlineFeatureStoreWriter* *attribute*), 73

`write_to_entity` (*butterfree.load.writers.OnlineFeatureStoreWriter* *attribute*), 79

`Writer` (*class in butterfree.load.writers.writer*), 75

`writers` (*butterfree.load.Sink* *attribute*), 83

`writers` (*butterfree.load.sink.Sink* *attribute*), 82

`writers()` (*butterfree.load.Sink* *property*), 83

`writers()` (*butterfree.load.sink.Sink* *property*), 82