764 lines - 10 Removals

```
import inspect
import logging
import uuid
from typing import Iterable, Set, Tuple, Union, List

import pandas as pd



from misc.logger import CustomLogger
from misc.decorators import timing, suppress_tracking
from misc.utils import extract_used_features, keys_mapping
from prov_acquisition.repository.neo4j import Neo4jFactory, Neo4jConnector

from prov_acquisition.prov_libraries.state import GlobalState, DataFrameState


class ProvenanceTracker:
    """
    Class that tracks changes in dataframes and traces provenance.
    """

    def __init__(self) -> None:

        self.logger = CustomLogger('ProvenanceTracker')
        self.logger.set_level(logging.DEBUG)

        self.__dataframe_tracking = True
        self.enable_dataframe_warning_msg = True

        self.global_state = GlobalState()

        self.neo4j = Neo4jFactory.create_neo4j_queries(uri="bolt://localhost",
                                       user="neo4j",
                                       pwd="password")
        self.neo4j.delete_all()

    @property
    def dataframe_tracking(self) -> bool:
        return self.__dataframe_tracking

    @dataframe_tracking.setter
    def dataframe_tracking(self, value: bool) -> None:
        if self.enable_dataframe_warning_msg:
            if value:
                self.logger.warning(f' Wrapper dataframe provenance tracker was enable!')
            else:
                self.logger.warning(
                    f' Wrapper dataframe provenance tracker was disable! Please use track_provenance method for tracking provenance.')
        self.__dataframe_tracking = value

    @suppress_tracking
    def _wrapper_track_provenance(self, f, tracker_id: str):
        """
        Wrapper function to track provenance.
```

830 lines + 78 Additions

```
import inspect
import logging
import uuid
from typing import Iterable, Set, Tuple, Union, List

import pandas as pd
import numpy as np
from numbers import Number

from misc.logger import CustomLogger
from misc.decorators import timing, suppress_tracking
from misc.utils import extract_used_features, keys_mapping
from prov_acquisition.repository.neo4j import Neo4jFactory, Neo4jConnector

from prov_acquisition.prov_libraries.state import GlobalState, DataFrameState


class ProvenanceTracker:
    """
    Class that tracks changes in dataframes and traces provenance.
    """

    def __init__(self) -> None:

        self.logger = CustomLogger('ProvenanceTracker')
        self.logger.set_level(logging.DEBUG)

        self.__dataframe_tracking = True
        self.enable_dataframe_warning_msg = True

        self.global_state = GlobalState()

        self.neo4j = Neo4jFactory.create_neo4j_queries(uri="bolt://localhost",
                                       user="neo4j",
                                       pwd="password")
        self.neo4j.delete_all()

    @property
    def dataframe_tracking(self) -> bool:
        return self.__dataframe_tracking

    @dataframe_tracking.setter
    def dataframe_tracking(self, value: bool) -> None:
        if self.enable_dataframe_warning_msg:
            if value:
                self.logger.warning(f' Wrapper dataframe provenance tracker was enable!')
            else:
                self.logger.warning(
                    f' Wrapper dataframe provenance tracker was disable! Please use track_provenance method for tracking provenance.')
        self.__dataframe_tracking = value

    @suppress_tracking
    def _wrapper_track_provenance(self, f, tracker_id: str):
        """
        Wrapper function to track provenance.
```

```
        If a method of the TrackedDataframe class is invoked,
        then the wrap function for capturing provenance will be run.

        """

        def wrap(*args, **kwargs):

            dataframe_state = self.global_state.dataframes_to_state[tracker_id]

            # print('args', args)
            # print("kwargs", kwargs)

            if callable(f):

                self.logger.info(f' Invoking {f.__name__} function')
                calling_function = inspect.stack()[1].function
                code = inspect.stack()[1].code_context[0].strip(' ').strip('\n')

                used_features = None
                if f.__name__ == '__item__':
                    used_features = extract_used_features(code, dataframe_state.df_input.columns)
                    column_to_add = args[1]

                if f.__name__ == 'merge':
                    kwargs['indicator'] = True

                result = f(*args, **kwargs)

                if hasattr(dataframe_state.df_input, calling_function):
                    self.logger.info(
                        f' The function {f.__name__} was called by {calling_function} function. Skipping data provenance phase.')
                    return result

                dataframe_state.df_output = result

                # If the result is a dataframe then the provenance is captured.
                if isinstance(result, pd.DataFrame) or result is None:

                    # In case of inplace operation.
                    if result is None:
                        dataframe_state.df_output = dataframe_state.df_input
                        dataframe_state.df_input = dataframe_state.df_input_copy

                    if f.__name__ == '__setitem__':
                        dataframe_state.df_output = args[0]

                    if not self.__dataframe_tracking:
                        self.logger.warning(
                            f' Wrapper dataframe provenance is disable! Data provenance will not be caught.')
                        self.__prepare_for_next_operation(dataframe_state=dataframe_state,

                                                          update_df_input=False, save=False)
```


```
        If a method of the TrackedDataframe class is invoked,
        then the wrap function for capturing provenance will be run.

        """

        def wrap(*args, **kwargs):

            dataframe_state = self.global_state.dataframes_to_state[tracker_id]

            # print('args', args)
            # print("kwargs", kwargs)

            if callable(f):

                self.logger.info(f' Invoking {f.__name__} function')
                calling_function = inspect.stack()[1].function
                code = inspect.stack()[1].code_context[0].strip(' ').strip('\n')

                used_features = None
                if f.__name__ == '__item__':
                    used_features = extract_used_features(code, dataframe_state.df_input.columns)
                    column_to_add = args[1]

                if f.__name__ == 'merge':
                    kwargs['indicator'] = True

                result = f(*args, **kwargs)

                if hasattr(dataframe_state.df_input, calling_function):
                    self.logger.info(
                        f' The function {f.__name__} was called by {calling_function} function. Skipping data provenance phase.')
                    return result

                dataframe_state.df_output = result

                # If the result is a dataframe then the provenance is captured.
                if isinstance(result, pd.DataFrame) or result is None:

                    # In case of inplace operation.
                    if result is None:
                        dataframe_state.df_output = dataframe_state.df_input
                        dataframe_state.df_input = dataframe_state.df_input_copy

                    if f.__name__ == '__setitem__':
                        dataframe_state.df_output = args[0]

                    if not self.__dataframe_tracking:
                        self.logger.warning(
                            f' Wrapper dataframe provenance is disable! Data provenance will not be caught.')
                        self.__prepare_for_next_operation(dataframe_state=dataframe_state,

                                                          update_df_input=False, save=False)
```

```
                        if result is None:
                            return result
                        else:
                            return self.create_tracked_dataframe(result, tracker_id=tracker_id)

                    # Preparing metadata for provenance capture.
                    self.global_state.update_basic_property(description=f.__name__,
                                                            code=code,
                                                            code_line=inspect.stack()[1].lineno,
                                                            function=f.__name__)
                    self.global_state.operation_number += 1

                    dataframe_state.update_hash_df_output()
                    dataframe_state.update_hash_df_output_common_index()

                    # Capture the provenance

                    if self.global_state.function == 'merge':

                        right_df_input = kwargs['right'] if 'right' in kwargs else args[1] if len(args) >= 1 else None

                        on = kwargs['on'] if 'on' in kwargs else args[3] if len(args) >= 4 else None
                        left_on = kwargs['left_on'] if 'left_on' in kwargs else args[4] if len(args) >= 5 else None
                        right_on = kwargs['right_on'] if 'right_on' in kwargs else args[5] if len(args) >= 6 else None
                        suffixes = kwargs['suffixes'] if 'suffixes' in kwargs else '_x', '_y'

                        if on:
                            left_on = on
                            right_on = on

                        self.global_state.operation_number -= 1

                        dataframe_state_right = self.global_state.dataframes_to_state[right_df_input.tracker_id]
                        dataframe_state.update_hash_row()
                        dataframe_state_right.update_hash_row()
                        self.global_state.operation_number += 1

                        self.__get_prov_join(dataframe_state_left=dataframe_state,
                                             dataframe_state_right=dataframe_state_right, left_keys=set(left_on),
                                             right_keys=set(right_on), suffixes=suffixes)
                    else:
```

```
                        if result is None:
                            return result
                        else:
                            return self.create_tracked_dataframe(result, tracker_id=tracker_id)

                    # Preparing metadata for provenance capture.
                    self.global_state.update_basic_property(description=f.__name__,
                                                            code=code,
                                                            code_line=inspect.stack()[1].lineno,
                                                            function=f.__name__)
                    self.global_state.operation_number += 1

                    dataframe_state.update_hash_df_output()
                    dataframe_state.update_hash_df_output_common_index()

                    # Capture the provenance

                    if self.global_state.function == 'merge':

                        right_df_input = kwargs['right'] if 'right' in kwargs else args[1] if len(args) >= 1 else None

                        on = kwargs['on'] if 'on' in kwargs else args[3] if len(args) >= 4 else None
                        left_on = kwargs['left_on'] if 'left_on' in kwargs else args[4] if len(args) >= 5 else None
                        right_on = kwargs['right_on'] if 'right_on' in kwargs else args[5] if len(args) >= 6 else None
                        suffixes = kwargs['suffixes'] if 'suffixes' in kwargs else '_x', '_y'

                        if on:
                            left_on = on
                            right_on = on

                        self.global_state.operation_number -= 1

                        dataframe_state_right = self.global_state.dataframes_to_state[right_df_input.tracker_id]
                        dataframe_state.update_hash_row()
                        dataframe_state_right.update_hash_row()
                        self.global_state.operation_number += 1

                        self.__get_prov_join(dataframe_state_left=dataframe_state,
                                             dataframe_state_right=dataframe_state_right, left_keys=set(left_on),
                                             right_keys=set(right_on), suffixes=suffixes)
                    else:
```

```
                        feature_mapping = self.__get_prov_feature_rename(dataframe_state=dataframe_state)

                        self.__get_prov_space_transformation(dataframe_state=dataframe_state,
                                                             feature_mapping=feature_mapping)

                        self.__get_prov_value_change(dataframe_state=dataframe_state, extra_used_features=used_features)

                    self.global_state.print_current_activities_info()

                    self.__prepare_for_next_operation(dataframe_state=dataframe_state,
                                                      update_df_input=True)

                if result is None:
                    return result
                else:
                    return self.create_tracked_dataframe(result, tracker_id=tracker_id)

        return wrap

    @suppress_tracking
    @timing
    def __get_prov_join(self, dataframe_state_left: DataFrameState, dataframe_state_right: DataFrameState,
                        left_keys: Set[str], right_keys: Set[str], suffixes: Tuple[str],
                        _merge_feature: bool = False) -> None:

        """
        Captures the provenance related to the join operation.
        Known issues to address: The merge operation can convert integer columns to float columns if they contain null values.
        Changing the type changes the hash of the row, resulting in missing corresponding indices.

        :param dataframe_state_left: The first input dataframe state.
        :param dataframe_state_right: The second input dataframe state.
        :param left_keys: Set of keys used for the left dataframe.
        :param right_keys: Set of keys used for the right dataframe.
        :param suffixes: Suffixes for common keys.
        :param _merge_feature: Indicates if the merge feature has been previously generated for provenance.

        """

        function_name = "Join"

        left_df_input = dataframe_state_left.df_input_copy
        right_df_input = dataframe_state_right.df_input_copy
        df_output = dataframe_state_left.df_output

```

```
                        feature_mapping = self.__get_prov_feature_rename(dataframe_state=dataframe_state)

                        self.__get_prov_space_transformation(dataframe_state=dataframe_state,
                                                             feature_mapping=feature_mapping)

                        self.__get_prov_value_change(dataframe_state=dataframe_state, extra_used_features=used_features)

                    self.global_state.print_current_activities_info()

                    self.__prepare_for_next_operation(dataframe_state=dataframe_state,
                                                      update_df_input=True)

                if result is None:
                    return result
                else:
                    return self.create_tracked_dataframe(result, tracker_id=tracker_id)

        return wrap

    @suppress_tracking
    @timing
    def __get_prov_join(self, dataframe_state_left: DataFrameState, dataframe_state_right: DataFrameState,
                        left_keys: Set[str], right_keys: Set[str], suffixes: Tuple[str],
                        _merge_feature: bool = False) -> None:

        """
        Captures the provenance related to the join operation.
        Known issues to address: The merge operation can convert integer columns to float columns if they contain null values.
        Changing the type changes the hash of the row, resulting in missing corresponding indices.

        :param dataframe_state_left: The first input dataframe state.
        :param dataframe_state_right: The second input dataframe state.
        :param left_keys: Set of keys used for the left dataframe.
        :param right_keys: Set of keys used for the right dataframe.
        :param suffixes: Suffixes for common keys.
        :param _merge_feature: Indicates if the merge feature has been previously generated for provenance.

        """

        function_name = "Join"

        left_df_input = dataframe_state_left.df_input_copy
        right_df_input = dataframe_state_right.df_input_copy
        df_output = dataframe_state_left.df_output

```

```
        used_features = set()

        left_suffix = suffixes[0]
        right_suffix = suffixes[1]

        # Get columns of left, right, and output datafra
mes
        left_columns = left_df_input.columns
        right_columns = right_df_input.columns
        output_columns = df_output.columns.difference
(['_merge'])

        # Identify common keys and columns
        common_keys = left_keys.intersection(right_keys)
        common_columns = left_columns.intersection(right
_columns).difference(common_keys)

        # Convert output dataframe to dictionary of reco
rds
        records = df_output.to_dict('index')

        generated_entities = []
        used_entities = []
        index_col_to_input_entities = {}

        # Iterate over each record in the output datafra
me
        for index, row in records.items():

            # Calculate hash values for the left and rig
ht rows
            left_hash_row = sum(
                [hash(str(row[e + left_suffix])) if e in
common_columns else hash(str(row[e])) for e in
                 left_df_input.columns])
            right_hash_row = sum(
                [hash(str(row[e + right_suffix])) if e i
n common_columns else hash(str(row[e])) for e in
                 right_df_input.columns])

            # Iterate over output columns
            for col_name in output_columns:

                output_value = row[col_name]

                # Create generated entity for the output
value
                generated_entity = self.global_state.cre
ate_entity(value=output_value, feature_name=col_name,

index=index,

instance=self.global_state.operation_number)
                generated_entities.append(generated_enti
ty['id'])
                index_col_to_input_entities[(index, col_
name)] = generated_entity

                # Process left-only or both cases
                if row['_merge'] == 'left_only' or row
['_merge'] == 'both':

                    set_of_indexes = dataframe_state_lef
t.hash_rows_to_indexes.get(left_hash_row, set())

                    for left_index in set_of_indexes:

```

```
        used_features = set()

        left_suffix = suffixes[0]
        right_suffix = suffixes[1]

        # Get columns of left, right, and output datafra
mes
        left_columns = left_df_input.columns
        right_columns = right_df_input.columns
        output_columns = df_output.columns.difference
(['_merge'])

        # Identify common keys and columns
        common_keys = left_keys.intersection(right_keys)
        common_columns = left_columns.intersection(right
_columns).difference(common_keys)

        # Convert output dataframe to dictionary of reco
rds
        records = df_output.to_dict('index')

        generated_entities = []
        used_entities = []
        index_col_to_input_entities = {}

        # Iterate over each record in the output datafra
me
        for index, row in records.items():

            # Calculate hash values for the left and rig
ht rows
            left_hash_row = sum(
                [hash(str(row[e + left_suffix])) if e in
common_columns else hash(str(row[e])) for e in
                 left_df_input.columns])
            right_hash_row = sum(
                [hash(str(row[e + right_suffix])) if e i
n common_columns else hash(str(row[e])) for e in
                 right_df_input.columns])

            # Iterate over output columns
            for col_name in output_columns:

                output_value = row[col_name]

                # Create generated entity for the output
value
                generated_entity = self.global_state.cre
ate_entity(value=output_value, feature_name=col_name,

index=index,

instance=self.global_state.operation_number)
                generated_entities.append(generated_enti
ty['id'])
                index_col_to_input_entities[(index, col_
name)] = generated_entity

                # Process left-only or both cases
                if row['_merge'] == 'left_only' or row
['_merge'] == 'both':

                    set_of_indexes = dataframe_state_lef
t.hash_rows_to_indexes.get(left_hash_row, set())

                    for left_index in set_of_indexes:

```

```
                        if col_name in left_columns or col_name.removesuffix(
                                left_suffix) in common_columns or col_name in common_keys:

                            # Get and remove the used entity from the left dataframe state
                            used_entity = dataframe_state_left.index_col_to_input_entities.pop(
                                (left_index, col_name.removesuffix(left_suffix)), None)

                            if used_entity is None:
                                continue

                            used_features.add(col_name)
                            used_entities.append(used_entity['id'])

                            # Create derivation relation between used and generated entities
                            self.global_state.create_derivation(used_ent=used_entity['id'],
                                                                gen_ent=generated_entity['id'])

                # Process right-only or both cases
                if row['_merge'] == 'right_only' or row['_merge'] == 'both':
                    set_of_indexes = dataframe_state_right.hash_rows_to_indexes.get(right_hash_row, set())

                    for right_index in set_of_indexes:
                        if col_name in right_columns or col_name.removesuffix(
                                right_suffix) in common_columns or col_name in common_keys:

                            # Get and remove the used entity from the right dataframe state
                            used_entity = dataframe_state_right.index_col_to_input_entities.pop(
                                (right_index, col_name.removesuffix(right_suffix)), None)

                            if used_entity is None:
                                continue

                            used_features.add(col_name)
                            used_entities.append(used_entity['id'])

                            # Create derivation relation between used and generated entities
                            self.global_state.create_derivation(used_ent=used_entity['id'],
                                                                gen_ent=generated_entity['id'])

        # Collect invalidated entities
        invalidated = []

        for index in dataframe_state_left.index_col_to_input_entities:
            invalidated.append(dataframe_state_left.index_col_to_input_entities[index]['id'])

```


```
                        if col_name in left_columns or col_name.removesuffix(
                                left_suffix) in common_columns or col_name in common_keys:

                            # Get and remove the used entity from the left dataframe state
                            used_entity = dataframe_state_left.index_col_to_input_entities.pop(
                                (left_index, col_name.removesuffix(left_suffix)), None)

                            if used_entity is None:
                                continue

                            used_features.add(col_name)
                            used_entities.append(used_entity['id'])

                            # Create derivation relation between used and generated entities
                            self.global_state.create_derivation(used_ent=used_entity['id'],
                                                                gen_ent=generated_entity['id'])

                # Process right-only or both cases
                if row['_merge'] == 'right_only' or row['_merge'] == 'both':
                    set_of_indexes = dataframe_state_right.hash_rows_to_indexes.get(right_hash_row, set())

                    for right_index in set_of_indexes:
                        if col_name in right_columns or col_name.removesuffix(
                                right_suffix) in common_columns or col_name in common_keys:

                            # Get and remove the used entity from the right dataframe state
                            used_entity = dataframe_state_right.index_col_to_input_entities.pop(
                                (right_index, col_name.removesuffix(right_suffix)), None)

                            if used_entity is None:
                                continue

                            used_features.add(col_name)
                            used_entities.append(used_entity['id'])

                            # Create derivation relation between used and generated entities
                            self.global_state.create_derivation(used_ent=used_entity['id'],
                                                                gen_ent=generated_entity['id'])

        # Collect invalidated entities
        invalidated = []

        for index in dataframe_state_left.index_col_to_input_entities:
            invalidated.append(dataframe_state_left.index_col_to_input_entities[index]['id'])

```

```
        for index in dataframe_state_right.index_col_to_input_entities:
            invalidated.append(dataframe_state_right.index_col_to_input_entities[index]['id'])

        invalidated.extend(used_entities)

        # Create activity and relation in the global state
        act_id = self.global_state.create_activity(function_name=function_name, used_features=list(used_features),
                                                   description=self.global_state.description,
                                                   code=self.global_state.code, code_line=self.global_state.code_line,
                                                   tracker_id=dataframe_state_left.tracker_id)

        self.global_state.create_relation(act_id=act_id, generated=generated_entities, used=used_entities,
                                          invalidated=invalidated,
                                          same=False)

        # Update the index_col_to_input_entities for the left and right dataframe states
        dataframe_state_left.index_col_to_input_entities = index_col_to_input_entities
        dataframe_state_right.index_col_to_input_entities = {}

        # Remove the '_merge' column from the output dataframe if the merge feature is not required
        if not _merge_feature:
            del df_output['_merge']

    @suppress_tracking
    @timing
    def __get_prov_space_transformation(self, dataframe_state: DataFrameState,
                                        feature_mapping: dict = {}) -> None:
        """
        Captures the provenance related to a change in the dataframe's dimensionality.
        This function uses indexes and can only be used if the operation for capturing provenance does not involve reindexing.

        Types of operations captured by this method:
            - Feature Selection: One or more features are removed.
            - Feature Augmentation: One or more features are added.
            - Instance Drop: One or more records are removed.
            - Instance Generation: One or more records are added.
            - Dimensionality Reduction: Features and records are added/removed. The overall number of removed features and records is greater than those added.
            - Space Augmentation: Features and records are added/removed. The overall number of added features and records is greater than those removed.
```


```
        for index in dataframe_state_right.index_col_to_input_entities:
            invalidated.append(dataframe_state_right.index_col_to_input_entities[index]['id'])

        invalidated.extend(used_entities)

        # Create activity and relation in the global state
        act_id = self.global_state.create_activity(function_name=function_name, used_features=list(used_features),
                                                   description=self.global_state.description,
                                                   code=self.global_state.code, code_line=self.global_state.code_line,
                                                   tracker_id=dataframe_state_left.tracker_id)

        self.global_state.create_relation(act_id=act_id, generated=generated_entities, used=used_entities,
                                          invalidated=invalidated,
                                          same=False)

        # Update the index_col_to_input_entities for the left and right dataframe states
        dataframe_state_left.index_col_to_input_entities = index_col_to_input_entities
        dataframe_state_right.index_col_to_input_entities = {}

        # Remove the '_merge' column from the output dataframe if the merge feature is not required
        if not _merge_feature:
            del df_output['_merge']

    @suppress_tracking
    @timing
    def __get_prov_space_transformation(self, dataframe_state: DataFrameState,
                                        feature_mapping: dict = {}) -> None:
        """
        Captures the provenance related to a change in the dataframe's dimensionality.
        This function uses indexes and can only be used if the operation for capturing provenance does not involve reindexing.

        Types of operations captured by this method:
            - Feature Selection: One or more features are removed.
            - Feature Augmentation: One or more features are added.
            - Instance Drop: One or more records are removed.
            - Instance Generation: One or more records are added.
            - Dimensionality Reduction: Features and records are added/removed. The overall number of removed features and records is greater than those added.
            - Space Augmentation: Features and records are added/removed. The overall number of added features and records is greater than those removed.
```

```
            - Space Transformation: Features and records are added/removed. In this case, there can be a reduction in dimensionality for one axis and a space augmentation for the other.
        :param dataframe_state: Input and output DataFrame state.
        :param feature_mapping: Mapping between the features of the output DataFrame and the input DataFrame.
        :return: None
        """

        function_name1 = "Feature Selection"
        function_name2 = "Feature Augmentation"

        function_name3 = "Instance Drop"
        function_name4 = "Instance Generation"

        function_name5 = "Dimensionality Reduction"
        function_name6 = "Space Augmentation"
        function_name7 = "Space Transformation"


        df_input = dataframe_state.df_input_copy
        df_output = dataframe_state.df_output

        dropped_rows = df_input.index.difference(df_output.index)
        dropped_cols = df_input.columns.difference(df_output.columns)
        augs_rows = df_output.index.difference(df_input.index)
        augs_cols = df_output.columns.difference(df_input.columns)
        int_rows = df_output.index.intersection(df_input.index)

        used_entities = []
        generated_entities = []
        used_cols = set()

        self.logger.info(f' Dropped cols: {dropped_cols}')
        self.logger.info(f' Dropped rows: {dropped_rows}')
        self.logger.info(f' Generated rows: {augs_rows}')
        self.logger.info(f' Generated cols: {augs_cols}')

        output_values = df_output.to_numpy()

        # Determine the type of function
        function_name = function_name7

        if len(dropped_cols) > 0 and len(augs_cols) == 0 and len(dropped_rows) == 0 and len(augs_rows) == 0:
            function_name = function_name1
        if len(dropped_cols) == 0 and len(augs_cols) > 0 and len(dropped_rows) == 0 and len(augs_rows) == 0:
            function_name = function_name2
```

```
            - Space Transformation: Features and records are added/removed. In this case, there can be a reduction in dimensionality for one axis and a space augmentation for the other.
            - One-Hot Encoding: Multiple binary features added representing categorical values of another feature.
        :param dataframe_state: Input and output DataFrame state.
        :param feature_mapping: Mapping between the features of the output DataFrame and the input DataFrame.
        :return: None
        """

        function_name1 = "Feature Selection"
        function_name2 = "Feature Augmentation"

        function_name3 = "Instance Drop"
        function_name4 = "Instance Generation"

        function_name5 = "Dimensionality Reduction"
        function_name6 = "Space Augmentation"
        function_name7 = "Space Transformation"

        function_name8 = "One-Hot Encoding"

        df_input = dataframe_state.df_input_copy
        df_output = dataframe_state.df_output

        dropped_rows = df_input.index.difference(df_output.index)
        dropped_cols = df_input.columns.difference(df_output.columns)
        augs_rows = df_output.index.difference(df_input.index)
        augs_cols = df_output.columns.difference(df_input.columns)
        int_rows = df_output.index.intersection(df_input.index)

        used_entities = []
        generated_entities = []
        used_cols = set()

        self.logger.info(f' Dropped cols: {dropped_cols}')
        self.logger.info(f' Dropped rows: {dropped_rows}')
        self.logger.info(f' Generated rows: {augs_rows}')
        self.logger.info(f' Generated cols: {augs_cols}')

        output_values = df_output.to_numpy()

        # Determine the type of function
        function_name = function_name7

        if len(dropped_cols) > 0 and len(augs_cols) == 0 and len(dropped_rows) == 0 and len(augs_rows) == 0:
            function_name = function_name1
        if len(dropped_cols) == 0 and len(augs_cols) > 0 and len(dropped_rows) == 0 and len(augs_rows) == 0:
            if set(pd.unique(df_output[augs_cols].values.ravel())) == set([0, 1]) and len(augs_cols) > 1:
                function_name = function_name8
```

```
        if len(dropped_cols) == 0 and len(augs_cols) == 0 and len(dropped_rows) > 0 and len(augs_rows) == 0:
            function_name = function_name3
        if len(dropped_cols) == 0 and len(augs_cols) == 0 and len(dropped_rows) == 0 and len(augs_rows) > 0:
            function_name = function_name4

        if (len(dropped_cols) > len(augs_cols)) and len(dropped_rows) > len(augs_cols):
            function_name = function_name5
        if (len(dropped_cols) < len(augs_cols)) and len(dropped_rows) < len(augs_cols):
            function_name = function_name6

        # Iterate over the removed rows to find values deleted due to an Instance Drop operation
        for index in dropped_rows:
            for _, col_name in feature_mapping.items():
                used_entity = dataframe_state.index_col_to_input_entities.pop((index, col_name), None)
                if used_entity:
                    used_cols.add(col_name)
                    used_entities.append(used_entity['id'])

        # Iterate over the added rows to find values added due to an Instance Generation operation
        for index in augs_rows:
            i = df_output.index.get_loc(index)
            for col_name, _ in feature_mapping.items():
                col = df_output.columns.get_loc(col_name)
                output_value = output_values[i, col]
                generated_entity = self.global_state.create_entity(value=output_value, feature_name=col_name,
                                                                   index=index,
                                                                   instance=self.global_state.operation_number)
                dataframe_state.index_col_to_input_entities[(index, col_name)] = generated_entity
                generated_entities.append(generated_entity['id'])

        # Iterate over the remaining rows to find values added/removed due to feature removal/addition
        for index in int_rows:

            i = df_output.index.get_loc(index)

            for col_name in dropped_cols:

                if col_name in set(feature_mapping.values()):
                    continue

                used_entity = dataframe_state.index_col_to_input_entities.pop((index, col_name), None)
                if used_entity:
                    used_cols.add(col_name)
                    used_entities.append(used_entity['id'])

            for col_name in augs_cols:

```

```
            else:
                function_name = function_name2

        if len(dropped_cols) == 0 and len(augs_cols) == 0 and len(dropped_rows) > 0 and len(augs_rows) == 0:
            function_name = function_name3
        if len(dropped_cols) == 0 and len(augs_cols) == 0 and len(dropped_rows) == 0 and len(augs_rows) > 0:
            function_name = function_name4

        if (len(dropped_cols) > len(augs_cols)) and len(dropped_rows) > len(augs_rows):
            function_name = function_name5
        if (len(dropped_cols) < len(augs_cols)) and len(dropped_rows) < len(augs_rows):
            function_name = function_name6

        # Iterate over the removed rows to find values deleted due to an Instance Drop operation
        for index in dropped_rows:
            for col_name in df_input.columns:
                used_entity = dataframe_state.index_col_to_input_entities.pop((index, col_name), None)
                if used_entity:
                    used_cols.add(col_name)
                    used_entities.append(used_entity['id'])

        # Iterate over the added rows to find values added due to an Instance Generation operation
        for index in augs_rows:
            i = df_output.index.get_loc(index)
            for col_name, _ in feature_mapping.items():
                col = df_output.columns.get_loc(col_name)
                output_value = output_values[i, col]
                generated_entity = self.global_state.create_entity(value=output_value, feature_name=col_name,
                                                                   index=index,
                                                                   instance=self.global_state.operation_number)
                dataframe_state.index_col_to_input_entities[(index, col_name)] = generated_entity
                generated_entities.append(generated_entity['id'])

        # Iterate over the remaining rows to find values added/removed due to feature removal/addition
        for index in int_rows:

            i = df_output.index.get_loc(index)

            for col_name in dropped_cols:

                if col_name in set(feature_mapping.values()):
                    continue

                used_entity = dataframe_state.index_col_to_input_entities.pop((index, col_name), None)
                if used_entity:
                    used_cols.add(col_name)
                    used_entities.append(used_entity['id'])

            for col_name in augs_cols:

```

```
                if col_name in feature_mapping:
                    continue

                col = df_output.columns.get_loc(col_name)
                output_value = output_values[i, col]
                generated_entity = self.global_state.create_entity(value=output_value, feature_name=col_name,
                                                                   index=index,
                                                                   instance=self.global_state.operation_number)
                dataframe_state.index_col_to_input_entities[(index, col_name)] = generated_entity
                generated_entities.append(generated_entity['id'])



        if len(generated_entities) > 0 or len(used_entities) > 0:
            act_id = self.global_state.create_activity(function_name, used_features=list(used_cols),
                                                       description=self.global_state.description,
                                                       code=self.global_state.code,
                                                       code_line=self.global_state.code_line,
                                                       generated_records=len(augs_rows) > 0,
                                                       generated_features=list(augs_cols),
                                                       deleted_records=len(dropped_rows) > 0,
                                                       deleted_used_features=list(dropped_cols),
                                                       tracker_id=dataframe_state.tracker_id)

            self.global_state.create_relation(act_id=act_id, generated=generated_entities, used=used_entities,
                                              invalidated=None, same=True)
```

```
                if col_name in feature_mapping:
                    continue

                col = df_output.columns.get_loc(col_name)
                output_value = output_values[i, col]
                generated_entity = self.global_state.create_entity(value=output_value, feature_name=col_name,
                                                                   index=index,
                                                                   instance=self.global_state.operation_number)
                dataframe_state.index_col_to_input_entities[(index, col_name)] = generated_entity
                generated_entities.append(generated_entity['id'])

                # if function_name == function_name8:
                #     col_val = pd.Series({col:df_input[col].unique() for col in df_input})

                #     for col_name2 in col_val.index:
                #         used_entity = dataframe_state.index_col_to_input_entities[(index, col_name2)]

                #         if len(set(augs_cols)) == len(np.unique(col_val[col_name2])):

                #             if set(augs_cols) == set(col_val[col_name2]):
                #                 used_cols.add(col_name2)
                #                 used_entities.append(used_entity['id'])
                #                 self.global_state.create_derivation(used_ent=used_entity['id'], gen_ent=generated_entity['id'])

        if len(generated_entities) > 0 or len(used_entities) > 0:
            act_id = self.global_state.create_activity(function_name, used_features=list(used_cols),
                                                       description=self.global_state.description,
                                                       code=self.global_state.code,
                                                       code_line=self.global_state.code_line,
                                                       generated_records=len(augs_rows) > 0,
                                                       generated_features=list(augs_cols),
                                                       deleted_records=len(dropped_rows) > 0,
                                                       deleted_used_features=list(dropped_cols),
                                                       tracker_id=dataframe_state.tracker_id)

            if function_name == function_name8:

                self.global_state.create_relation(act_id=act_id, generated=generated_entities, used=used_entities,
```

```python

    @suppress_tracking
    @timing
    def __get_prov_feature_rename(self, dataframe_state: DataFrameState) -> dict:
        """
        Captures the provenance related to the renaming of one or more features.

        :param dataframe_state: Input and output DataFrame state.
        :return: dict - Mapping between the features of the output DataFrame and the input DataFrame.
        """

        function_name = "Feature Rename"

        df_input = dataframe_state.df_input_copy
        df_output = dataframe_state.df_output

        int_rows = df_output.index.intersection(df_input.index)

        used_entities = []
        generated_entities = []
        used_features = set()
        generated_features = set()

        output_values = df_output.to_numpy()

        hash_df_output_common_index = dataframe_state.hash_df_output_common_index.to_dict()
        hash_df_input = dataframe_state.hash_df_input.to_dict()

        feature_mapping = keys_mapping(hash_df_output_common_index, hash_df_input)

        # Iterate over the intersecting rows to find feature rename operations
        for index in int_rows:

            i = df_output.index.get_loc(index)

            for col_name1, col_name2 in feature_mapping.items():

                if col_name1 == col_name2:
                    continue

                col = df_output.columns.get_loc(col_name1)
                output_value = output_values[i, col]

                used_entity = dataframe_state.index_col_to_input_entities.pop((index, col_name2), None)
                if used_entity:
                    generated_entity = self.global_state.create_entity(value=output_value, feature_name=col_name1,
```

```python
                                                      invalidated=None, same=False)
            else:
                self.global_state.create_relation(act_id=act_id, generated=generated_entities, used=used_entities,
                                                  invalidated=None, same=True)

    @suppress_tracking
    @timing
    def __get_prov_feature_rename(self, dataframe_state: DataFrameState) -> dict:
        """
        Captures the provenance related to the renaming of one or more features.

        :param dataframe_state: Input and output DataFrame state.
        :return: dict - Mapping between the features of the output DataFrame and the input DataFrame.
        """

        function_name = "Feature Rename"

        df_input = dataframe_state.df_input_copy
        df_output = dataframe_state.df_output

        int_rows = df_output.index.intersection(df_input.index)

        used_entities = []
        generated_entities = []
        used_features = set()
        generated_features = set()

        output_values = df_output.to_numpy()

        hash_df_output_common_index = dataframe_state.hash_df_output_common_index.to_dict()
        hash_df_input = dataframe_state.hash_df_input.to_dict()

        feature_mapping = keys_mapping(hash_df_output_common_index, hash_df_input)

        # Iterate over the intersecting rows to find feature rename operations
        for index in int_rows:

            i = df_output.index.get_loc(index)

            for col_name1, col_name2 in feature_mapping.items():

                if col_name1 == col_name2:
                    continue

                col = df_output.columns.get_loc(col_name1)
                output_value = output_values[i, col]

                used_entity = dataframe_state.index_col_to_input_entities.pop((index, col_name2), None)
                if used_entity:
                    generated_entity = self.global_state.create_entity(value=output_value, feature_name=col_name1,
```

```
                    index=index,
                    instance=self.global_state.operation_number)
                    generated_entities.append(generated_entity['id'])
                    used_entities.append(used_entity['id'])
                    generated_features.add(col_name1)
                    used_features.add(col_name2)

                    dataframe_state.index_col_to_input_entities[(index, col_name1)] = generated_entity
                    self.global_state.create_derivation(used_ent=used_entity['id'], gen_ent=generated_entity['id'])

        if len(generated_features) > 0:
            self.logger.info(f' Feature rename detect: {feature_mapping}')
            act_id = self.global_state.create_activity(function_name, used_features=list(used_features),
                                                       description=self.global_state.description,
                                                       code=self.global_state.code,
                                                       code_line=self.global_state.code_line,
                                                       generated_features=list(generated_features),
                                                       tracker_id=dataframe_state.tracker_id)

            self.global_state.create_relation(act_id=act_id, generated=generated_entities, used=used_entities,
                                              invalidated=[],
                                              same=True)

        return feature_mapping

    @suppress_tracking
    @timing
    def __get_prov_value_change(self, dataframe_state, extra_used_features: set = None) -> None:

        """
        Captures the provenance related to a change in the values of the DataFrame.
        The type of generated activity can be of two types:
        Value Transformation: generic case.
        Imputation: the DataFrame column has undergone a replacement of null values.


        :param dataframe_state: Input and output DataFrame state.
        :param extra_used_features: Extra features used in the input. Indicates additional features that contribute to the generation of new entities.
        :return: None

        """

        self.logger.info(f' Check for value change...')
```

```
                    index=index,
                    instance=self.global_state.operation_number)
                    generated_entities.append(generated_entity['id'])
                    used_entities.append(used_entity['id'])
                    generated_features.add(col_name1)
                    used_features.add(col_name2)

                    dataframe_state.index_col_to_input_entities[(index, col_name1)] = generated_entity
                    self.global_state.create_derivation(used_ent=used_entity['id'], gen_ent=generated_entity['id'])

        if len(generated_features) > 0:
            self.logger.info(f' Feature rename detect: {feature_mapping}')
            act_id = self.global_state.create_activity(function_name, used_features=list(used_features),
                                                       description=self.global_state.description,
                                                       code=self.global_state.code,
                                                       code_line=self.global_state.code_line,
                                                       generated_features=list(generated_features),
                                                       tracker_id=dataframe_state.tracker_id)

            self.global_state.create_relation(act_id=act_id, generated=generated_entities, used=used_entities,
                                              invalidated=[],
                                              same=True)

        return feature_mapping

    @suppress_tracking
    @timing
    def __get_prov_value_change(self, dataframe_state, extra_used_features: set = None) -> None:

        """
        Captures the provenance related to a change in the values of the DataFrame.
        The type of generated activity can be of two types:
        Value Transformation: generic case.
        Imputation: the DataFrame column has undergone a replacement of null values.
        Ordinal Encoding: the DataFrame column has been encoded.

        :param dataframe_state: Input and output DataFrame state.
        :param extra_used_features: Extra features used in the input. Indicates additional features that contribute to the generation of new entities.
        :return: None

        """

        self.logger.info(f' Check for value change...')
```

```
        function_name1 = "Value Transformation"
        function_name2 = "Imputation"


        df_input = dataframe_state.df_input_copy
        df_output = dataframe_state.df_output

        int_columns = df_output.columns.intersection(df_input.columns)

        generated_entities = []
        used_entities = []
        extra_used_entities = []
        imp_cols = set()
        trans_cols = set()

        values_output = df_output[int_columns].to_numpy()




        if extra_used_features is None:
            used_features = set()

        for i in df_output.index:

            index = df_output.index.get_loc(i)




            for col_name in int_columns:

                col = int_columns.get_loc(col_name)
                new_value = values_output[index][col]

                used_entity = dataframe_state.index_col_to_input_entities.get((index, col_name), None)

                if used_entity is None:
                    continue

                old_value = used_entity['value']

                if new_value != old_value:
                    if pd.isnull(old_value) and pd.isnull(new_value):
                        continue
                    elif pd.isnull(old_value):
                        imp_cols.add(col_name)
```

```
        function_name1 = "Value Transformation"
        function_name2 = "Imputation"
        function_name3 = "Ordinal Encoding"

        df_input = dataframe_state.df_input_copy
        df_output = dataframe_state.df_output

        int_columns = df_output.columns.intersection(df_input.columns)

        generated_entities = []
        used_entities = []
        extra_used_entities = []
        imp_cols = set()
        trans_cols = set()
        enc_cols = set()
        values_output = df_output[int_columns].to_numpy()

        dropped_rows = df_input.index.difference(df_output.index)
        drop_index = set()
        for j in range(0, len(dropped_rows)):
            drop_index.add(df_input.index.get_loc(dropped_rows[j]))
        drop_mod = sum(j < df_input.index for j in drop_index)
        drop_mod = np.delete(drop_mod, list(drop_index))

        if extra_used_features is None:
            used_features = set()

        for i in df_output.index:

            index = df_output.index.get_loc(i)

            if len(drop_index) > 0:
                index_in = index + drop_mod[index]
            else:
                index_in = index

            for col_name in int_columns:

                col = int_columns.get_loc(col_name)
                new_value = values_output[index][col]

                used_entity = dataframe_state.index_col_to_input_entities.get((index_in, col_name), None)

                if used_entity is None:
                    continue

                old_value = used_entity['value']

                if new_value != old_value:
                    if pd.isnull(old_value) and pd.isnull(new_value):
                        continue
                    elif pd.isnull(old_value):
                        imp_cols.add(col_name)
                    elif type(old_value) == str and isinstance(new_value, Number):
                        if new_value.is_integer():
                            if old_value.isdigit():
                                if int(old_value) == new_value:
                                    continue
```

```
                    else:
                        trans_cols.add(col_name)

                    entity = self.global_state.create_entity(value=new_value, feature_name=col_name, index=index,
                                                             instance=self.global_state.operation_number)
                    dataframe_state.index_col_to_input_entities[(index, col_name)] = entity

                    generated_entities.append(entity['id'])
                    used_entities.append(used_entity['id'])

                    self.global_state.create_derivation(used_ent=used_entity['id'], gen_ent=entity['id'])

                    # Extra features to add
                    for used_feature in used_features:
                        used_entity = dataframe_state.index_col_to_input_entities.get((index, used_feature), None)

                        if used_entity is None:
                            continue

                        extra_used_entities.append(used_entity)
                        self.global_state.create_derivation(used_ent=used_entity['id'], gen_ent=entity['id'])

        imp_cols = imp_cols.difference(trans_cols)

        if len(imp_cols) > 0:
```

```
                            else:
                                enc_cols.add(col_name)
                    else:
                        trans_cols.add(col_name)

                    entity = self.global_state.create_entity(value=new_value, feature_name=col_name, index=index,
                                                             instance=self.global_state.operation_number)
                    dataframe_state.index_col_to_input_entities[(index, col_name)] = entity

                    generated_entities.append(entity['id'])
                    used_entities.append(used_entity['id'])

                    self.global_state.create_derivation(used_ent=used_entity['id'], gen_ent=entity['id'])

                    # Extra features to add
                    for used_feature in used_features:
                        used_entity = dataframe_state.index_col_to_input_entities.get((index, used_feature), None)

                        if used_entity is None:
                            continue

                        extra_used_entities.append(used_entity)
                        self.global_state.create_derivation(used_ent=used_entity['id'], gen_ent=entity['id'])

        imp_cols = imp_cols.difference(trans_cols)

        if len(enc_cols) > 0:
            self.logger.info(f' Encoding detect on {enc_cols} columns')
            enc_cols = enc_cols.union(used_features)
            extra_used_entities.extend(used_entities)
            act_id = self.global_state.create_activity(function_name=function_name3, used_features=list(enc_cols),
                                                       description=self.global_state.description,
                                                       code=self.global_state.code,
                                                       code_line=self.global_state.code_line,
                                                       tracker_id=dataframe_state.tracker_id)
            self.global_state.create_relation(act_id=act_id, generated=generated_entities,
                                              used=used_entities if len(
                                                  extra_used_entities) == 0 else extra_used_entities,
                                              invalidated=None, same=len(extra_used_entities) == len(used_entities))

        if len(imp_cols) > 0:
```

```
            self.logger.info(f' Imputation detect on {imp_cols} columns')
            act_id = self.global_state.create_activity(function_name=function_name2, used_features=list(imp_cols),
                                                       description=self.global_state.description,
                                                       code=self.global_state.code,
                                                       code_line=self.global_state.code_line,
                                                       tracker_id=dataframe_state.tracker_id)
            self.global_state.create_relation(act_id=act_id, generated=generated_entities, used=None,
                                              invalidated=None, same=True)

        if len(trans_cols) > 0:
            self.logger.info(f' Value transformation detect on {trans_cols} columns')
            trans_cols = trans_cols.union(used_features)
            extra_used_entities.extend(used_entities)
            act_id = self.global_state.create_activity(function_name=function_name1, used_features=list(trans_cols),
                                                       description=self.global_state.description,
                                                       code=self.global_state.code,
                                                       code_line=self.global_state.code_line,
                                                       tracker_id=dataframe_state.tracker_id)
            self.global_state.create_relation(act_id=act_id, generated=generated_entities,
                                              used=used_entities if len(
                                                  extra_used_entities) == 0 else extra_used_entities,
                                              invalidated=None, same=len(extra_used_entities) == len(used_entities))
        else:
            self.logger.info(f' Value transformation and Imputation not detected')


    @suppress_tracking
    @timing
    def check_equals_dataframe(self, feature_mapping: dict)-> bool:
        """
        Check if the df_input and df_output dataframes are equal.

        """

        function_name = "Check Equals Dataframe"

        result = False
        hash_df_output = self.hash_df_output.copy()
        hash_df_output.rename(feature_mapping)
        if self._df_output is not None and self.hash_df_output is not None:
```

```
            self.logger.info(f' Imputation detect on {imp_cols} columns')
            act_id = self.global_state.create_activity(function_name=function_name2, used_features=list(imp_cols),
                                                       description=self.global_state.description,
                                                       code=self.global_state.code,
                                                       code_line=self.global_state.code_line,
                                                       tracker_id=dataframe_state.tracker_id)
            self.global_state.create_relation(act_id=act_id, generated=generated_entities, used=None,
                                              invalidated=None, same=True)

        if len(trans_cols) > 0:
            self.logger.info(f' Value transformation detect on {trans_cols} columns')
            trans_cols = trans_cols.union(used_features)
            extra_used_entities.extend(used_entities)
            act_id = self.global_state.create_activity(function_name=function_name1, used_features=list(trans_cols),
                                                       description=self.global_state.description,
                                                       code=self.global_state.code,
                                                       code_line=self.global_state.code_line,
                                                       tracker_id=dataframe_state.tracker_id)
            self.global_state.create_relation(act_id=act_id, generated=generated_entities,
                                              used=used_entities if len(
                                                  extra_used_entities) == 0 else extra_used_entities,
                                              invalidated=None, same=len(extra_used_entities) == len(used_entities))

        if len([enc_cols, imp_cols, trans_cols]) == 0:

            self.logger.info(f' Value transformation, Encoding and Imputation not detected')


    @suppress_tracking
    @timing
    def check_equals_dataframe(self, feature_mapping: dict)-> bool:
        """
        Check if the df_input and df_output dataframes are equal.

        """

        function_name = "Check Equals Dataframe"

        result = False
        hash_df_output = self.hash_df_output.copy()
        hash_df_output.rename(feature_mapping)
        if self._df_output is not None and self.hash_df_output is not None:
```

```
            result = self.hash_df_input.equals(hash_df_output)

        if result:
            self.logger.info(f' {function_name}: dataframe are equals!')

        return result

    def __prepare_for_next_operation(self, dataframe_state: DataFrameState, update_df_input: bool = True,
                                     save: bool = True) -> None:
        """
        Prepare for the next operation.

        """
        self.global_state.description = None

        dataframe_state.df_input = dataframe_state.df_output

        if update_df_input:
            dataframe_state.df_input_copy = dataframe_state.df_input.copy()
            dataframe_state.hash_df_input = dataframe_state.hash_df_output

        dataframe_state.hash_df_output = None

        if save:

            # Save to neo4j

            session = Neo4jConnector().create_session()

            self.neo4j.create_constraint(session=session)
            self.neo4j.add_activities(self.global_state.current_activities, session)
            self.neo4j.add_entities(self.global_state.current_entities)
            self.neo4j.add_derivations(self.global_state.current_derivations)
            self.neo4j.add_relations(self.global_state.current_relations)

            if self.global_state.last_activities:
                next_operations = [{'act_in_id': a_in['id'], 'act_out_id': a_out['id']}
                                   for a_out in self.global_state.current_activities
                                   for a_in in self.global_state.last_activities
                                   if a_out['tracker_id'] == a_in['tracker_id']]
                self.neo4j.add_next_operations(next_operations, session)

            self.global_state.last_activities = self.global_state.current_activities.copy()

            # Free memory
            del self.global_state.current_activities[:]
            del self.global_state.current_entities[:]
            del self.global_state.current_derivations[:]
            del self.global_state.current_relations[:]


```

```
            result = self.hash_df_input.equals(hash_df_output)

        if result:
            self.logger.info(f' {function_name}: dataframe are equals!')

        return result

    def __prepare_for_next_operation(self, dataframe_state: DataFrameState, update_df_input: bool = True,
                                     save: bool = True) -> None:
        """
        Prepare for the next operation.

        """
        self.global_state.description = None

        dataframe_state.df_input = dataframe_state.df_output

        if update_df_input:
            dataframe_state.df_input_copy = dataframe_state.df_input.copy()
            dataframe_state.hash_df_input = dataframe_state.hash_df_output

        dataframe_state.hash_df_output = None

        if save:

            # Save to neo4j

            session = Neo4jConnector().create_session()

            self.neo4j.create_constraint(session=session)
            self.neo4j.add_activities(self.global_state.current_activities, session)
            self.neo4j.add_entities(self.global_state.current_entities)
            self.neo4j.add_derivations(self.global_state.current_derivations)
            self.neo4j.add_relations(self.global_state.current_relations)

            if self.global_state.last_activities:
                next_operations = [{'act_in_id': a_in['id'], 'act_out_id': a_out['id']}
                                   for a_out in self.global_state.current_activities
                                   for a_in in self.global_state.last_activities
                                   if a_out['tracker_id'] == a_in['tracker_id']]
                self.neo4j.add_next_operations(next_operations, session)

            self.global_state.last_activities = self.global_state.current_activities.copy()

            # Free memory
            del self.global_state.current_activities[:]
            del self.global_state.current_entities[:]
            del self.global_state.current_derivations[:]
            del self.global_state.current_relations[:]


```

```
    def create_tracked_dataframe(self, df: pd.DataFrame, tracker_id: str):
        """
        Create a tracked dataframe.

        """

        class DataFrameTracked(pd.DataFrame, metaclass=TrackedDataFrameMeta, tracker=self, tracker_id=tracker_id):
            pass

        return DataFrameTracked(df)

    def subscribe(self, df: Union[pd.DataFrame, List[pd.DataFrame]]) -> List[pd.DataFrame]:

        if isinstance(df, pd.DataFrame):
            # Case where only one dataframe is passed
            tracker_id = str(uuid.uuid4())
            dataframe_state = DataFrameState(tracker_id=tracker_id)

            df_tracked = self.create_tracked_dataframe(df=df, tracker_id=tracker_id)

            dataframe_state.df_input = df_tracked
            dataframe_state.df_input_copy = df_tracked.copy()

            self.global_state.add_dataframes(df_tracked.tracker_id, dataframe_state)
            self.global_state.init_prov_entities(tracker_id=df_tracked.tracker_id)

            return [df_tracked]

        elif isinstance(df, list):
            # Case where a list of dataframes is passed
            tracked_dfs = []
            for single_df in df:
                tracker_id = str(uuid.uuid4())
                dataframe_state = DataFrameState(tracker_id=tracker_id)

                df_tracked = self.create_tracked_dataframe(df=single_df, tracker_id=tracker_id)

                dataframe_state.df_input = df_tracked
                dataframe_state.df_input_copy = df_tracked.copy()

                self.global_state.add_dataframes(df_tracked.tracker_id, dataframe_state)
                self.global_state.init_prov_entities(tracker_id=df_tracked.tracker_id)

                tracked_dfs.append(df_tracked)

            return tracked_dfs

        else:
            raise ValueError("Invalid input format. Expected a single DataFrame or a list of DataFrames.")


class TrackedDataFrameMeta(type):
    """
    Defines the metaclass for the DataFrameTraked
```


```
    def create_tracked_dataframe(self, df: pd.DataFrame, tracker_id: str):
        """
        Create a tracked dataframe.

        """

        class DataFrameTracked(pd.DataFrame, metaclass=TrackedDataFrameMeta, tracker=self, tracker_id=tracker_id):
            pass

        return DataFrameTracked(df)

    def subscribe(self, df: Union[pd.DataFrame, List[pd.DataFrame]]) -> List[pd.DataFrame]:

        if isinstance(df, pd.DataFrame):
            # Case where only one dataframe is passed
            tracker_id = str(uuid.uuid4())
            dataframe_state = DataFrameState(tracker_id=tracker_id)

            df_tracked = self.create_tracked_dataframe(df=df, tracker_id=tracker_id)

            dataframe_state.df_input = df_tracked
            dataframe_state.df_input_copy = df_tracked.copy()

            self.global_state.add_dataframes(df_tracked.tracker_id, dataframe_state)
            self.global_state.init_prov_entities(tracker_id=df_tracked.tracker_id)

            return [df_tracked]

        elif isinstance(df, list):
            # Case where a list of dataframes is passed
            tracked_dfs = []
            for single_df in df:
                tracker_id = str(uuid.uuid4())
                dataframe_state = DataFrameState(tracker_id=tracker_id)

                df_tracked = self.create_tracked_dataframe(df=single_df, tracker_id=tracker_id)

                dataframe_state.df_input = df_tracked
                dataframe_state.df_input_copy = df_tracked.copy()

                self.global_state.add_dataframes(df_tracked.tracker_id, dataframe_state)
                self.global_state.init_prov_entities(tracker_id=df_tracked.tracker_id)

                tracked_dfs.append(df_tracked)

            return tracked_dfs

        else:
            raise ValueError("Invalid input format. Expected a single DataFrame or a list of DataFrames.")


class TrackedDataFrameMeta(type):
    """
    Defines the metaclass for the DataFrameTraked
```

```
    """

    def __new__(cls, name, bases, dct, tracker, tracker_id: str):
        """
        Every method (except exceptions) will be encapsulated.
        The wrapper function will take care of tracking the provenance

        """

        child = super().__new__(cls, name, bases, dct)

        setattr(child, 'tracker_id', tracker_id)

        exceptions = ['__init__', '_constructor_sliced', '_get_item_cache', '_clear_item_cache', '_ixs',
                      '_box_col_values', 'iterrows', '__repr__', '_info_repr', 'to_string', '__len__', 'itertuples',
                      'to_dict', '__getitem__', '_maybe_cache_changed', '_append', '_set_item', '_sanitize_column',
                      '_ensure_valid_index', '_set_item_mgr', '_iset_item_mgr', '_cmp_method', '_dispatch_frame_op',
                      '_construct_result', '_setitem_frame', 'isna', 'to_numpy', 'values', 'corr', 'isnull', 'nunique',
                      'select_dtypes', 'items']

        for base in bases:
            for field_name, field in base.__dict__.items():
                if callable(field):
                    if field_name not in exceptions and not isinstance(field, Iterable):
                        setattr(child, field_name, tracker._wrapper_track_provenance(field, tracker_id))

        return child

```

```
    """

    def __new__(cls, name, bases, dct, tracker, tracker_id: str):
        """
        Every method (except exceptions) will be encapsulated.
        The wrapper function will take care of tracking the provenance

        """

        child = super().__new__(cls, name, bases, dct)

        setattr(child, 'tracker_id', tracker_id)

        exceptions = ['__init__', '_constructor_sliced', '_get_item_cache', '_clear_item_cache', '_ixs',
                      '_box_col_values', 'iterrows', '__repr__', '_info_repr', 'to_string', '__len__', 'itertuples',
                      'to_dict', '__getitem__', '_maybe_cache_changed', '_append', '_set_item', '_sanitize_column',
                      '_ensure_valid_index', '_set_item_mgr', '_iset_item_mgr', '_cmp_method', '_dispatch_frame_op',
                      '_construct_result', '_setitem_frame', 'isna', 'to_numpy', 'values', 'corr', 'isnull', 'nunique',
                      'select_dtypes', 'items']

        for base in bases:
            for field_name, field in base.__dict__.items():
                if callable(field):
                    if field_name not in exceptions and not isinstance(field, Iterable):
                        setattr(child, field_name, tracker._wrapper_track_provenance(field, tracker_id))

        return child

```