PulseDBTransformer Functions - Detailed Explanation

This document provides a comprehensive breakdown of each function in the `pulse_to_postgresql.py` script, including code snippets and in-depth explanations of the implementation details.

Table of Contents

---

init

```
def __init__(self, pg_host='localhost', pg_port=5432, pg_dbname='em_pulse_data',
pg_user='postgres', pg_password='postgres'):
    # SQL Server connection parameters
    self.server = '10.20.2.10'
    self.database = 'pulse'
    self.username = 'Pulse_RO'
    self.password = 'PD@T@r3@der'
    self.connection_string = f"DRIVER={{ODBC Driver 17 for SQL Server}};SERVER=
{self.server};DATABASE={self.database};UID={self.username};PWD={self.password}"
    self.engine = create_engine("mssql+pyodbc:///?odbc_connect=" +
self.connection_string)

    # Target PostgreSQL connection parameters
    self.pg_host = pg_host
    self.pg_port = pg_port
    self.pg_dbname = pg_dbname
    self.pg_user = pg_user
    self.pg_password = pg_password
    self.pg_engine = create_engine(f"postgresql://{pg_user}:
{pg_password}@{pg_host}:{pg_port}/{pg_dbname}")

    # Mill names and sensor tags
    self.mills = ['Mill01', 'Mill02', 'Mill03', 'Mill04', 'Mill05', 'Mill06',
                 'Mill07', 'Mill08', 'Mill09', 'Mill10', 'Mill11', 'Mill12']
    # self.mills = ['Mill01']  # Uncomment for testing with a single mill

    # SQL tags dictionary (snippet - full dictionary is much larger)
```

```
        self.sql_tags = {
            'Ore': {"485": "Mill01", "488": "Mill02", "491": "Mill03", "494": "Mill04",
                    "497": "Mill05", "500": "Mill06", "455": "Mill07", "467": "Mill08",
                    "476": "Mill09", "479": "Mill10", "482": "Mill11", "3786":
    "Mill12"},
            # Other features follow similar pattern...
        }

        # Table names from SQL Server
        self.table_names = ['LoggerValues',]

        # Initialize timestamp filter to None (get all data)
        self.filter_timestamp = None
```

Purpose

Initializes a new PulseDBTransformer object that connects to both SQL Server (source) and PostgreSQL (target) databases. This constructor sets up all necessary connections, configuration parameters, and data structures needed for the ETL process.

Parameters

- `pg_host`: PostgreSQL server hostname (default: 'localhost')
- `pg_port`: PostgreSQL server port (default: 5432)
- `pg_dbname`: PostgreSQL database name (default: 'em_pulse_data')
- `pg_user`: PostgreSQL username (default: 'postgres')
- `pg_password`: PostgreSQL password (default: 'postgres')

Process Breakdown

1. SQL Server Connection Setup

```
# SQL Server connection parameters
self.server = '10.20.2.10'
self.database = 'pulse'
self.username = 'Pulse_RO'
self.password = 'PD@T@r3@der'
```

These hardcoded parameters specify the connection details for the source SQL Server database. The user 'Pulse_RO' appears to be a read-only user, which is appropriate for ETL processes.

```
self.connection_string = f"DRIVER={{ODBC Driver 17 for SQL Server}};SERVER=
{self.server};DATABASE={self.database};UID={self.username};PWD={self.password}"
```

This builds an ODBC connection string for SQL Server. The double braces `{{}}` around "ODBC Driver 17 for SQL Server" are necessary to escape the curly braces in f-strings.

```
self.engine = create_engine("mssql+pyodbc:///?odbc_connect=" +
self.connection_string)
```

Creates a SQLAlchemy engine using the pyodbc connector. The engine is the main entry point for SQLAlchemy's SQL functionality and allows for executing SQL statements.

2. PostgreSQL Connection Setup

```
# Target PostgreSQL connection parameters
self.pg_host = pg_host
self.pg_port = pg_port
self.pg_dbname = pg_dbname
self.pg_user = pg_user
self.pg_password = pg_password
self.pg_engine = create_engine(f"postgresql://{pg_user}:{pg_password}@{pg_host}:
{pg_port}/{pg_dbname}")
```

These lines store the PostgreSQL connection parameters and create a SQLAlchemy engine for the target database. Unlike the SQL Server connection, these parameters are passed as arguments to the constructor, making the code more flexible.

3. Mill Configuration

```
# Mill names and sensor tags
self.mills = ['Mill01', 'Mill02', 'Mill03', 'Mill04', 'Mill05', 'Mill06',
             'Mill07', 'Mill08', 'Mill09', 'Mill10', 'Mill11', 'Mill12']
# self.mills = ['Mill01']  # Uncomment for testing with a single mill
```

This defines the list of mills to process. The commented line provides an easy way to limit processing to a single mill for testing or troubleshooting.

4. SQL Tags Dictionary

The `self.sql_tags` dictionary is a complex mapping structure that serves as the cornerstone of the data transformation process:

```
# SQL tags dictionary (greatly simplified example)
self.sql_tags = {
    'Ore': {"485": "Mill01", "488": "Mill02", ... }
    # Many more features and mappings...
}
```

This multi-level dictionary has:

- **First level keys**: Feature names (e.g., 'Ore', 'WaterMill', etc.)
- **Second level**: Maps tag IDs (from SQL Server) to mill names

This structure allows the code to:

1. Query specific feature tags from SQL Server
2. Identify which mill each tag belongs to
3. Rename columns from numeric tag IDs to human-readable mill names

5. Table Names Configuration

```
# Table names from SQL Server
self.table_names = ['LoggerValues',]
```

Specifies which tables to query in SQL Server. By default, only the current 'LoggerValues' table is used. The commented section in the original code shows how this can be expanded to include historical archived tables (e.g., 'LoggerValues_Archive_Jan2025').

6. Timestamp Filter Initialization

```
# Initialize timestamp filter to None (get all data)
```

```
    self.filter_timestamp = None
```

This initializes the timestamp filter to `None`, meaning no filtering by default. This parameter becomes critical during incremental loading operations, where it's set to filter only data newer than what's already in PostgreSQL.

Implementation Details

- **Connection Engines**: The code uses SQLAlchemy for both database connections, providing a consistent API regardless of the underlying database system.
- **Configuration Flexibility**: While SQL Server parameters are hardcoded, PostgreSQL parameters are configurable through constructor parameters, allowing for different deployment scenarios.
- **Mill Configuration**: The explicit list of mills enables selective processing of specific mills if needed.
- **Extensible Table Structure**: The `table_names` list can be expanded to include historical data tables when needed.

Technical Notes

- The SQL Server connection uses ODBC Driver 17, which must be installed on the system where the script runs.
- The `self.sql_tags` dictionary is particularly important, as it defines what data is extracted and how it's transformed. Any changes to tag IDs in the source system would require updates to this dictionary.
- The timestamp filter (`self.filter_timestamp`) is initially `None` but gets set during incremental loading to filter data by timestamp.

---

read_sql_table

```
def read_sql_table(self, table_name, feature):
    """Read data from SQL Server for a specific feature"""
    tags = "LoggerTagID = " + " OR LoggerTagID =
".join(self.sql_tags[feature].keys())

    # Add timestamp filter if we're in append mode
    timestamp_filter = ""
    if hasattr(self, 'filter_timestamp') and self.filter_timestamp is not None:
        since_str = self.filter_timestamp.strftime('%Y-%m-%d %H:%M:%S')
        timestamp_filter = f" AND IndexTime > '{since_str}'"
        print(f"  - Filtering data after: {since_str}")

    query_str = f'SELECT IndexTime, LoggerTagID, Value FROM {table_name} WHERE
{tags}{timestamp_filter} ORDER BY IndexTime DESC'
    # print(f"  - SQL Query: {query_str}")

    query = pd.read_sql_query(query_str, self.engine)

    # If no data found, return empty dataframe
    if query.empty:
        print(f"  - No data found in {table_name} for {feature} with the current
filter")
        return pd.DataFrame()
    else:
        print(f"  - Found {len(query)} rows in {table_name} for {feature}")

    # Process the data
    query = query.drop_duplicates(subset='IndexTime', keep='last')
    df = query.pivot(index="IndexTime", columns="LoggerTagID", values="Value")
    df = df.ffill().bfill()  # Using newer pandas methods
    df = df.resample("1min").mean()

    # Rename columns to mill names
    df.columns = [self.sql_tags[feature][str(k)] for k in df.columns if str(k) in
self.sql_tags[feature]]
    df.index.names = ['TimeStamp']
    df.sort_index(axis=1, inplace=True)

    return df
```

Purpose

The `read_sql_table` function is responsible for extracting and transforming data from a specific SQL Server table for a given feature. This is a core data extraction function that handles SQL querying, timestamp filtering, and initial data transformation.

Parameters

- `table_name`: Name of the SQL Server table to query (e.g., 'LoggerValues')
- `feature`: Feature name (must be a key in `self.sql_tags`, e.g., 'Ore', 'WaterMill')

Process Breakdown

1. Building the SQL Query

```
tags = "LoggerTagID = " + " OR LoggerTagID = ".join(self.sql_tags[feature].keys())
```

This line constructs the SQL WHERE condition for filtering by LoggerTagID. It joins all the tag IDs for the specified feature with ' OR LoggerTagID = ' to create a condition like 'LoggerTagID = 485 OR LoggerTagID = 488 OR...'.

```
# Add timestamp filter if we're in append mode
timestamp_filter = ""
if hasattr(self, 'filter_timestamp') and self.filter_timestamp is not None:
    since_str = self.filter_timestamp.strftime('%Y-%m-%d %H:%M:%S')
    timestamp_filter = f" AND IndexTime > '{since_str}'"
    print(f"  - Filtering data after: {since_str}")
```

This conditional block adds timestamp filtering when `self.filter_timestamp` is set (during incremental loading). The timestamp is formatted as a SQL-compatible date string, and an additional WHERE condition is added to filter records after this timestamp.

```
query_str = f'SELECT IndexTime, LoggerTagID, Value FROM {table_name} WHERE {tags}
{timestamp_filter} ORDER BY IndexTime DESC'
```

This assembles the complete SQL query, selecting the IndexTime, LoggerTagID, and Value columns, with the appropriate WHERE conditions, and ordering by IndexTime in descending order.

2. Executing the Query

```
query = pd.read_sql_query(query_str, self.engine)
```

Executes the SQL query using pandas' `read_sql_query` function, which returns the results as a pandas DataFrame. The query runs against the SQL Server database via `self.engine`.

3. Handling Empty Results

```
# If no data found, return empty dataframe
if query.empty:
    print(f"  - No data found in {table_name} for {feature} with the current
filter")
    return pd.DataFrame()
else:
    print(f"  - Found {len(query)} rows in {table_name} for {feature}")
```

This block checks if any data was returned. If the query result is empty, it logs this information and returns an empty DataFrame. Otherwise, it logs the number of rows found.

4. Data Transformation

```
# Process the data
query = query.drop_duplicates(subset='IndexTime', keep='last')
```

Removes duplicate timestamps from the data, keeping only the last occurrence of each timestamp. This is important because the same timestamp might have multiple readings.

```
df = query.pivot(index="IndexTime", columns="LoggerTagID", values="Value")
```

Pivots the data to transform it from a long format to a wide format:

- IndexTime becomes the DataFrame index
- LoggerTagID values become column names
- Value entries become the cell values

This transforms the data from:

```
IndexTime | LoggerTagID | Value
-------------------------
```

```
2025-06-19 | 485 | 42.3
2025-06-19 | 488 | 38.7
```

To:

```
IndexTime | 485 | 488
-------------------------
2025-06-19 | 42.3 | 38.7
```

```
df = df.ffill().bfill()  # Using newer pandas methods
```

Fills missing values in the DataFrame:

- `ffill()`: Forward fill - propagates the last valid observation forward
- `bfill()`: Backward fill - uses the next valid observation to fill gaps

This ensures there are no gaps in the data series.

```
df = df.resample("1min").mean()
```

Resamples the data to 1-minute intervals, calculating the mean for each interval. This ensures consistent time intervals across all data points.

5. Column Renaming and Ordering

```
# Rename columns to mill names
df.columns = [self.sql_tags[feature][str(k)] for k in df.columns if str(k) in
self.sql_tags[feature]]
```

This list comprehension transforms numeric LoggerTagID column names (e.g., '485') to their corresponding mill names (e.g., 'Mill01') using the mapping in `self.sql_tags`. It creates a new set of column names by:

1. Iterating through each column (k)
2. Converting k to a string (because dictionary keys are strings)
3. Looking up the corresponding mill name in the dictionary

```
df.index.names = ['TimeStamp']
```

Renames the index from 'IndexTime' to 'TimeStamp' for clarity and consistency.

```
df.sort_index(axis=1, inplace=True)
```

Sorts the columns alphabetically to ensure consistent column ordering.

Return Value

A pandas DataFrame with:

- Index: TimeStamp (renamed from IndexTime)
- Columns: Mill names for the specified feature (e.g., 'Mill01', 'Mill02')
- Values: Sensor readings for each mill

Technical Notes

- **Data Consistency**: The combination of deduplication, pivoting, and resampling ensures that the returned data has a consistent structure with uniform time intervals.
- **Memory Efficiency**: Ordering by `IndexTime DESC` in the SQL query is important for performance. Getting the most recent data first is more efficient when filtering by timestamp.
- **Dynamic SQL**: The query is constructed dynamically based on the feature being processed, allowing the same function to extract different types of data.
- **NULL Handling**: The `ffill().bfill()` sequence ensures that missing values are appropriately handled, maintaining data continuity.

Example

For the 'Ore' feature, this function might:

1. Query SQL Server for all Ore-related LoggerTagIDs (485, 488, 491, etc.)
2. Transform the raw data to have a timestamp index
3. Convert tag IDs to mill names (485 → Mill01, 488 → Mill02, etc.)
4. Return a DataFrame with columns like 'Mill01', 'Mill02', etc., each containing ore measurements

---

compose_feature

```python
def compose_feature(self, feature):
    """Combine data from all tables for a specific feature"""
    frames = []
    for tbl in self.table_names:
        print(f"Processing {tbl} for {feature}")
        frames.append(self.read_sql_table(tbl, feature))

    df = pd.concat(frames)
    df = df.sort_index()
    df = df[~df.index.duplicated(keep='first')]  # Remove duplicate timestamps
    df = df.shift(2, freq='h')
    df = df.ffill().bfill()  # Using newer pandas methods
    return df
```

Purpose

The `compose_feature` function aggregates data for a specific feature (e.g., 'Ore', 'WaterMill') across multiple source SQL Server tables. It manages the combination of data from current and possibly archived tables, then applies essential transformations like time-shifting and handling duplicate timestamps.

Parameters

- `feature`: Feature name to compose (e.g., 'Ore', 'WaterMill')

Process Breakdown

1. Data Collection from Multiple Tables

```
frames = []
for tbl in self.table_names:
    print(f"Processing {tbl} for {feature}")
    frames.append(self.read_sql_table(tbl, feature))
```

This loop iterates through each table name in `self.table_names` (which could include current and archive tables) and:

1. Prints a status message indicating which table is being processed
2. Calls `read_sql_table` to extract and process data for the specified feature from that table
3. Appends the resulting DataFrame to the `frames` list

This design allows the code to process data from multiple tables, which is essential for historical data that might be stored in archive tables.

2. Data Consolidation

```
df = pd.concat(frames)
```

Combines all the individual DataFrames into a single DataFrame. The `pd.concat` function vertically stacks the DataFrames, preserving the index (timestamp) information. This creates a unified dataset containing all records from all tables for the specified feature.

3. Time-based Sorting

```
df = df.sort_index()
```

Sorts the consolidated DataFrame by its index (TimeStamp), ensuring chronological order. This is crucial for time series data to maintain proper sequence for subsequent operations and analysis.

4. Removing Duplicate Timestamps

```
df = df[~df.index.duplicated(keep='first')]  # Remove duplicate timestamps
```

This removes any duplicate timestamps that might have resulted from combining data from multiple tables. The `keep='first'` parameter means that if multiple rows share the same timestamp, only the first occurrence is kept. This boolean indexing with the `~` operator (NOT) selects only rows with timestamps that are not duplicated.

5. Time Shifting

```
df = df.shift(2, freq='h')
```

Shifts all timestamps forward by 2 hours. This critical operation adjusts for a business requirement, likely dealing with time zone differences or aligning data with operational shifts. The `freq='h'` parameter specifies that the shift is in hours.

6. Handling Missing Values

```
df = df.ffill().bfill()  # Using newer pandas methods
```

Fills any remaining missing values in the DataFrame:

- `ffill()`: Forward fill - propagates the last valid observation forward
- `bfill()`: Backward fill - uses the next valid observation to fill gaps backward

This ensures data continuity by eliminating gaps in the time series.

Return Value

A pandas DataFrame containing the combined and processed data from all tables for the specified feature, with:

- Index: TimeStamp (shifted forward by 2 hours) A combined pandas DataFrame with:
- Index: TimeStamp (shifted by 2 hours)
- Columns: Mill names for the specified feature
- Values: Sensor readings from all tables

Key Operations

- Concatenation of data from multiple tables
- Time shifting by 2 hours
- Removal of duplicate timestamps
- Filling of missing values

---

create_mill_dataframe

```python
def create_mill_dataframe(self, mill):
    """Create a dataframe for a specific mill with all features"""
    all_data = []
    common_index = None

    for feature in self.sql_tags.keys():
        print(f"Processing {feature} for {mill}")
        feature_df = self.compose_feature(feature)
        if mill in feature_df.columns:
            feature_series = feature_df[mill]
            if common_index is None:
                common_index = feature_series.index
            else:
                common_index = common_index.intersection(feature_series.index)
            all_data.append((feature, feature_series))

    # Create dataframe with aligned index
    mill_df = pd.DataFrame(index=common_index)
    for feature, series in all_data:
        mill_df[feature] = series.reindex(common_index)

    return mill_df
```

Purpose

The `create_mill_dataframe` function generates a comprehensive dataset for a single mill, combining data from all available features (like Ore, Water, Power) into a unified DataFrame. This pivots the data perspective from feature-centric to mill-centric, creating a complete profile of the mill's operational metrics over time.

Parameters

- `mill`: Mill name (e.g., 'Mill01', 'Mill02') to generate the DataFrame for

Process Breakdown

1. Initialization

```
all_data = []
common_index = None
```

Prepares two key data structures:

- `all_data`: A list that will store tuples of (feature_name, feature_series)
- `common_index`: Will hold the intersection of timestamps across all features for consistent time alignment

2. Feature Collection and Index Alignment

```
for feature in self.sql_tags.keys():
    print(f"Processing {feature} for {mill}")
    feature_df = self.compose_feature(feature)
    if mill in feature_df.columns:
        feature_series = feature_df[mill]
        if common_index is None:
            common_index = feature_series.index
        else:
            common_index = common_index.intersection(feature_series.index)
        all_data.append((feature, feature_series))
```

This loop iterates through each feature (like 'Ore', 'WaterMill', etc.) and:

1. Prints a status message for tracking progress
2. Calls `compose_feature` to get all mills' data for that feature
3. Checks if the specified mill exists in the feature's DataFrame (since not all mills may have data for all features)
4. If present, extracts the series for just this mill
5. Updates the common_index to be the intersection of all timestamps seen so far
    - This ensures that the final DataFrame will only contain timestamps where data exists for ALL features
6. Stores the feature name and its corresponding series in the `all_data` list

The index intersection operation is particularly important, as it ensures time-alignment across all features, preventing situations where some features have data at timestamps where others don't.

3. DataFrame Construction

```
# Create dataframe with aligned index
mill_df = pd.DataFrame(index=common_index)
for feature, series in all_data:
    mill_df[feature] = series.reindex(common_index)
```

This final block:

1. Creates a new empty DataFrame with the common timestamp index
2. Iterates through each feature series collected in `all_data`
3. Reindexes each series to match the common index (which should be unnecessary given the prior intersection, but ensures consistency)
4. Adds each feature as a column in the DataFrame, named after the feature

The result is a DataFrame where:

- Each row represents a specific timestamp
- Each column represents a different feature (e.g., 'Ore', 'WaterMill')
- Each cell contains the measurement value for that feature at that timestamp

Return Value

A pandas DataFrame with:

- Index: TimeStamp - representing the timeline of measurements
- Columns: Feature names (e.g., 'Ore', 'WaterMill', 'Power')
- Values: Measurements for the specified mill across all features

Technical Notes

- **DataFrame Pivoting**: This function performs a conceptual pivot of the data model. While previous steps organized data by feature, this function reorganizes it by mill, creating a complete operational profile.

- **Missing Feature Handling**: If a mill doesn't have data for a particular feature, that feature is simply not included in the final DataFrame.

- **Time Alignment**: The intersection of indexes ensures that the resulting DataFrame only contains timestamps where data exists for all features, providing a clean, aligned dataset for analysis.

- **Data Integrity**: By collecting all features for a single mill into one DataFrame, this function enables comprehensive analysis of the mill's operations over time, making it possible to study relationships between different operational metrics.

Example

For 'Mill01', the function might process:

1. 'Ore' data → produces a Series with Ore measurements
2. 'WaterMill' data → produces a Series with Water measurements
3. 'Power' data → produces a Series with Power measurements

The resulting DataFrame would have columns ['Ore', 'WaterMill', 'Power'] with aligned timestamps, ready for analysis or storage.

Purpose

Creates a comprehensive DataFrame for a specific mill by combining data for all features associated with that mill.

Parameters

- `mill`: Mill name to process (e.g., "Mill01")

Process

1. Creates dictionaries to store feature DataFrames and common time indices

2. For each feature in `self.sql_tags`:

   - Calls `compose_feature` to get data for that feature
   - If the mill exists in the feature's columns:
     - Extracts the mill's data as a Series

        - Renames the Series to the feature name
        - Stores this Series in the `feature_frames` dictionary
        - Stores the Series index in `feature_indices`

3. If no features have data for the mill, returns an empty DataFrame

4. Finds the common time range across all features:

    - Takes the intersection of all feature indices
    - Ensures all features will align on the same timestamps

5. Creates a DataFrame with:

    - Common time index
    - Each feature as a column
    - Data values from each feature for the mill

Return Value

A pandas DataFrame with:

- Index: TimeStamp (common across all features)
- Columns: Feature names (Ore, WaterMill, etc.)
- Values: Sensor readings for the specific mill

Example

For Mill01, the result might have columns like:

- Ore: Values from ore sensors
- WaterMill: Values from water mill sensors
- WaterFinePump: Values from water fine pump sensors

---

save_to_postgresql

```
def save_to_postgresql(self, schema='mills'):
    """Save all mill data to PostgreSQL database, replacing existing data"""
```

Purpose

Processes all mills and saves their data to PostgreSQL, replacing any existing data.

Parameters

- `schema`: PostgreSQL schema name (default: 'mills')

Process

1. Creates the specified schema if it doesn't exist:
    - Uses a SQL query with `CREATE SCHEMA IF NOT EXISTS`
2. For each mill in `self.mills`:
    - Calls `create_mill_dataframe` to get the mill's data
    - Converts mill name to table name (e.g., Mill01 → MILL_01)
    - If data exists:
        - Saves it to PostgreSQL using `to_sql` with `if_exists='replace'`
        - Uses TimeStamp as the index
    - Reports progress with print statements
3. Displays a completion message

Key Operations

- Schema creation
- Generation of mill data
- Complete replacement of existing tables
- Progress reporting

PostgreSQL Table Structure

For each mill:

- Table name: MILL_XX (e.g., MILL_01)
- Schema: as specified (default: mills)
- Index: TimeStamp
- Columns: Feature names (Ore, WaterMill, etc.)
- Values: Sensor readings

---

append_to_postgresql

```
def append_to_postgresql(self, schema='mills'):
    """Append new data to existing PostgreSQL database tables, only adding records
newer than the latest timestamp"""
```

Purpose

Incrementally loads data from SQL Server to PostgreSQL, only adding new records that don't already exist.

Parameters

- `schema`: PostgreSQL schema name (default: 'mills')

Process

1. Creates the specified schema if it doesn't exist

2. Saves the original table_names to restore later

3. For each mill in `self.mills`:

    - Converts mill name to table name (e.g., Mill01 → MILL_01)

    - Checks if the table exists in PostgreSQL:

      If table exists:

        - Queries for the latest timestamp in the table

        - Sets a filter timestamp 2.1 hours earlier than the latest timestamp (2.1 hours accounts for the 2-hour time shift plus a 0.1-hour buffer)
        - Creates a DataFrame for the mill with data newer than the filter
        - Filters out any data with timestamps not newer than the latest timestamp
        - Appends new data to the existing table using `to_sql` with `if_exists='append'`

    If table doesn't exist:

        - Creates a full DataFrame for the mill (all available data)
        - Creates a new table with this data

4. Resets the filter timestamp to None

5. Restores the original table_names

6. Displays a completion message

Key Operations

- Timestamped filtering for incremental loading
- Checking for existing tables
- Selective append of only new data
- Handling of edge cases (non-existent tables)

Technical Note

The 2.1-hour adjustment is critical because:

- Data timestamps are shifted forward by 2 hours in the `compose_feature` method
- An extra 0.1-hour buffer ensures no data is missed

---

main

```
def main():
    # Function body...
```

Purpose

Entry point for the script when run directly, handling command-line arguments and executing the appropriate operation.

Process

1. Defines PostgreSQL connection parameters:
    - Host: 'em-m-db4.ellatzite-med.com'
    - Port: 5432
    - Database: 'em_pulse_data'
    - User: 's.lyubenov'
    - Password: 'tP9uB7sH7mK6zA7t'
2. Initializes the PulseDBTransformer with these parameters
3. Determines which operation to perform:
    - If run with 'append' argument: Calls `append_to_postgresql()`
    - Otherwise: Calls `save_to_postgresql()`

Usage

- Full data load: `python pulse_to_postgresql.py`
- Incremental update: `python pulse_to_postgresql.py append`

Command-line Interface

The script accepts an optional 'append' argument that determines the operation mode:

- No arguments: Replaces all data
- 'append' argument: Adds only new data