Pulse to PostgreSQL - Code Explanation

This document provides a detailed explanation of the `pulse_to_postgresql.py` script, which extracts data from a Microsoft SQL Server database and loads it into a PostgreSQL database.

Table of Contents

Imports

```
import pandas as pd
from datetime import datetime, timedelta
from sqlalchemy import create_engine, text
import pyodbc
import psycopg2
import sys
import os
from sqlalchemy.exc import SQLAlchemyError
```

- **pandas (pd)**: Used for data manipulation and analysis
- **datetime, timedelta**: For handling date and time operations
- **sqlalchemy**: SQL toolkit and ORM library used to communicate with databases
- **pyodbc**: Library for connecting to ODBC databases (MS SQL Server in this case)
- **psycopg2**: PostgreSQL database adapter
- **sys, os**: Standard Python modules for system interactions
- **SQLAlchemyError**: For handling sqlalchemy-specific exceptions

PulseDBTransformer Class

Constructor

The constructor initializes the following:

```
def __init__(self, pg_host='localhost', pg_port=5432, pg_dbname='em_pulse_data',
pg_user='postgres', pg_password='postgres'):
```

SQL Server Connection

```
# 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)
```

This section sets up the connection to the source SQL Server database:

- Defines the server address, database name, username, and password
- Creates a connection string compatible with the ODBC driver
- Initializes a SQLAlchemy engine for database interactions

PostgreSQL Connection

```
# 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}")
```

This section sets up the connection to the target PostgreSQL database:

- Takes connection parameters as constructor arguments with default values
- Creates a SQLAlchemy engine for PostgreSQL database interactions

Mill Names and 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.

SQL Tags Dictionary

```
# SQL tags dictionary mapping tag IDs to mill names for each feature
self.sql_tags = {
    'Ore': {"485" : "Mill01", "488" : "Mill02", ...},
    'WaterMill': {"560" : "Mill01", "580" : "Mill02", ...},
```

```
    ...
}
```

This dictionary maps numeric tag IDs in the SQL Server to the corresponding mill and feature. The structure is:

- Top level: Feature names (Ore, WaterMill, etc.)
- Second level: Tag ID (as string) → Mill name

Table Names

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

# Uncomment for historical data
# self.table_names = ['LoggerValues',
#         'LoggerValues_Archive_Jan2025', 'LoggerValues_Archive_Dec2024', ...
```

This defines which tables to query in the SQL Server database:

- Default is just the current data in 'LoggerValues'
- Commented section allows processing historical archived data

Filter Timestamp

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

This parameter is used for incremental loading in the append operation:

- When `None`, all available data is processed
- When set to a timestamp, only data newer than that timestamp is processed

Data Processing Methods

read_sql_table

```
def read_sql_table(self, table_name, feature):
```

```
    """Read data from SQL Server for a specific feature"""
```

This method:

1. Builds a SQL query for a specific feature from a specific table
2. Adds timestamp filtering if applicable
3. Executes the query and retrieves the data
4. Processes the data:
    - Removes duplicate timestamps
    - Pivots the data (IndexTime as index, LoggerTagID as columns)
    - Fills missing values
    - Resamples to 1-minute intervals
    - Renames columns to mill names based on the sql_tags dictionary
5. Returns a DataFrame with mills as columns and timestamps as index

compose_feature

```
def compose_feature(self, feature):
    """Combine data from all tables for a specific feature"""
```

This method:

1. Processes each table in `self.table_names` for the given feature
2. Combines the results into a single DataFrame
3. Sorts by timestamp and removes duplicate timestamps
4. Shifts the timestamps by 2 hours (to align data)
5. Fills missing values
6. Returns a combined DataFrame for the feature

create_mill_dataframe

```
def create_mill_dataframe(self, mill):
    """Create a dataframe for a specific mill with all features"""
```

This method:

1. Processes each feature for the specified mill
2. Finds the common time index across all features

3. Creates a DataFrame with features as columns for the mill
4. Returns a DataFrame with features as columns and timestamps as index

Database Operations

save_to_postgresql

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

This method:

1. Creates the specified schema if it doesn't exist
2. Processes each mill in `self.mills`
3. Creates a DataFrame for each mill with all features
4. Formats the table name (e.g., Mill01 → MILL_01)
5. Saves each mill's data to PostgreSQL, replacing any existing data
6. Reports progress and results

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"""
```

This method implements incremental loading:

1. Creates the specified schema if it doesn't exist
2. Stores the original table_names to restore later
3. For each mill:
    - Checks if the table exists in PostgreSQL
    - If it exists, finds the latest timestamp in the table
    - Sets a filter timestamp to only process newer data (with a 2.1-hour buffer for time shifts)
    - Creates a DataFrame for the mill with the filtered data
    - Filters out data with timestamps not newer than the latest timestamp
    - Appends the new data to the existing table

    - If the table doesn't exist, creates it with all available data
4. Restores the original state (clears filter_timestamp)
5. Reports progress and results

Main Function

```
def main():
    # PostgreSQL connection parameters
    pg_host = 'em-m-db4.ellatzite-med.com'
    pg_port = 5432
    pg_dbname = 'em_pulse_data'
    pg_user = 's.lyubenov'
    pg_password = 'tP9uB7sH7mK6zA7t'

    # Initialize the transformer with PostgreSQL connection parameters
    transformer = PulseDBTransformer(
        pg_host=pg_host,
        pg_port=pg_port,
        pg_dbname=pg_dbname,
        pg_user=pg_user,
        pg_password=pg_password
    )

    # Save or append based on command line argument
    if len(sys.argv) > 1 and sys.argv[1] == 'append':
        transformer.append_to_postgresql()
    else:
        transformer.save_to_postgresql()
```

The main function:

1. Defines PostgreSQL connection parameters
2. Initializes the PulseDBTransformer class with these parameters
3. Determines whether to perform a full save or incremental append operation:
    - If the script is run with the 'append' argument, it performs an append operation
    - Otherwise, it performs a full save/replace operation

Data Flow

1. **Extract**: Data is extracted from SQL Server by:

- Querying tables listed in `self.table_names`
- Filtering by tag IDs in the `self.sql_tags` dictionary
- Optional timestamp filtering for incremental loading

2. **Transform**: Data is transformed by:

   - Pivoting to organize by timestamp and tag ID
   - Renaming columns to meaningful mill names
   - Resampling for consistent time intervals
   - Filling missing values
   - Time shifting (2 hours)
   - Organizing into mill-specific DataFrames with features as columns

3. **Load**: Data is loaded into PostgreSQL by:

   - Creating schemas and tables as needed
   - Either replacing existing data or appending new data
   - Using SQLAlchemy's to_sql method for the actual loading