Sweeft-Project

Table Designs

Collections Table

id PK	collection	name	description	image_url	owner	twitter_username

Contracts Table

id PK	address	chain

collection_contracts table(since the collection can have mutliple contracts i implemented this table as a form of many-to-many relationship)

id PK	collectionid FK	contract_id FK

Data Extraction

```
headers = {
    "accept": "application/json",
    "x-api-key": "74236956836e4336b836845ae04c25f6"
}

def make_call_to_api(headers, url="https://api.opensea.io/api/v2/collections?chain=ethe
    response = requests.get(url, headers=headers)
    return response
```

Data Transformation

Cleaning the data (replacing null values with “Not Found” and dropping unnecessary columns)

```
def drop_unnecessary_columns(df, columns):
    # ["collection","name","twitter_username","contracts","description","owner","image_
    return df[columns]

def replace_all_whitespace(df):
    df.replace('', 'NOT FOUND', inplace=True)
```

Data Loading

Loading data into the ORM

```
def prepare_data_for_orm(df):
    df.to_dict(orient="records")


def insert_data(self, data):
        """
        handles both inserting a single value and multiple values
        :param data: list of dictionary (contracts field must be a dictionary itself)
        :return:
        """
        for entry in data:
            # auto generated contract id's
            ag_ids = []
```

```
            # list of contract dictionaries
            list_of_contract_dict = entry.pop('contracts', None)
            for dic in list_of_contract_dict:
                ag_ids.append(self.insert_into_contracts(dic["address"], dic["chain"]))
            collection_id = self.insert_into_collection(entry)
            for contract_id in ag_ids:
                self.insert_into_collection_contracts(collection_id, contract_id)
```

```
    def insert_into_contracts(self, address, chain):
        query = """
        INSERT INTO contracts (address, chain)
        VALUES (%s, %s);
        """
        self.cursor.execute(query, (address, chain))
        self.connection.commit()

        return self.cursor.lastrowid

    def insert_into_collection(self, data):
        query = """
        INSERT INTO collections (collection, name, description, image_url, owner, twitt
        VALUES (%s, %s, %s, %s, %s, %s);
        """
        params = tuple(data.values())
        self.cursor.execute(query, params)
        self.connection.commit()

        return self.cursor.lastrowid

    def insert_into_collection_contracts(self, collection_id, contract_id):
        query = """
        INSERT INTO collection_contracts (collection_id, contract_id) VALUES (%s, %s);
        """

        params = (collection_id, contract_id)
        self.cursor.execute(query, params)
        self.connection.commit()
```

Saving the data as json

```
def save_df_to_json(df):
    df.to_json("data.json", orient='records')
```

Database connection and management

- Connecting to database

```
    def __init__(self, host, user, password, database):
        self.connection = mysql.connector.connect(
            host=host,
            user=user,
            password=password,
            database=database
```

```
        )
        self.cursor = self.connection.cursor(dictionary=True)
```

- Creating tables

```
    def create_schema(self):
        self.create_collections_table()
        self.create_contracts_table()
        self.create_collection_contracts_relationship_table()


   def create_collections_table(self):
        query = """
        CREATE TABLE IF NOT EXISTS collections (
        id INT AUTO_INCREMENT PRIMARY KEY,
        collection VARCHAR(255),
        name VARCHAR(255),
        description TEXT,
        image_url VARCHAR(255),
        owner VARCHAR(255),
        twitter_username VARCHAR(255)
        );
        """
        self.cursor.execute(query)
        self.connection.commit()

    def create_contracts_table(self):
        query = """
        CREATE TABLE  IF NOT EXISTS contracts(
         id INT AUTO_INCREMENT PRIMARY KEY,
        address VARCHAR(255),
        chain VARCHAR(255)
        );
        """

        self.cursor.execute(query)
        self.connection.commit()

    def create_collection_contracts_relationship_table(self):
        query = """
        CREATE TABLE IF NOT EXISTS collection_contracts (
        id INT AUTO_INCREMENT PRIMARY KEY,
        collection_id INT,
        contract_id INT,
        FOREIGN KEY (collection_id) REFERENCES collections(id),
        FOREIGN KEY (contract_id) REFERENCES contracts(id)
        );
        """
        self.cursor.execute(query)
        self.connection.commit()
```

- Deleting tables

```
def delete_table(self, table):
        query = f"DROP TABLE IF EXISTS {table};"
        self.cursor.execute(query)
        self.connection.commit()
```

- Connecting to existing tables

here some of the subtasks are written as one function , those subtasks are

- Connect to existing tables.
- Perform SELECT statements, with LIMIT and ORDERBY
- Implement filtering operators like LIKE and ILIKE, IN in SELECT statements

```
    def retrieve_table_data(self, table, limit=None, order_by=None, LIKE=None, ILIKE=No
        """
        :param LIKE:
        :param ILIKE:
        :param order_by:
        :param limit:
        :param table:
        :return: dictionary of the table rows
        """
        query = f"SELECT * FROM {table}"
        if LIKE:
            query += f" WHERE {LIKE[0]} LIKE '{LIKE[1]}'"
        elif ILIKE:
            query += f" WHERE {LIKE[0]} ILIKE '{ILIKE[1]}'"
        if order_by:
            query += f" ORDER BY {order_by}"
        if limit:
            query += f" LIMIT {limit}"
        query += ";"
        self.cursor.execute(query)
        result = self.cursor.fetchall()
        return result

     sample usage => print(db.retrieve_table_data("collections",limit=5,LIKE=("name","%
```

Removing/adding columns to a table and changing its type

```
    def add_column_to_table(self, table, column, column_type):
        query = f"ALTER TABLE {table} ADD COLUMN {column} {column_type};"
        self.cursor.execute(query)
        self.connection.commit()

    def remove_column_from_table(self, table, column):
        query = f"ALTER TABLE {table} DROP COLUMN {column};"
        self.cursor.execute(query)
        self.connection.commit()

    def change_column_type(self, table, column_name, new_type):
        query = f"ALTER TABLE {table} MODIFY COLUMN {column_name} {new_type};"
```

```
        self.cursor.execute(query)
        self.connection.commit()
```

CRUD Operations

```
    def delete_row(self, table, row_id):
        """
        since all the tables in the db have id field no need for seperate functions for
        :param table:
        :param row_id:
        :return:
        """
        query = f"DELETE FROM {table} WHERE id = %s;"
        self.cursor.execute(query, (row_id,))
        self.connection.commit()

    def update_row(self, table, row_id, new_data):
        """
        :param table:
        :param row_id:
        :param new_data: dictionary containing column names and new values
        """
        set_values = ', '.join([f"{key} = %s" for key in new_data.keys()])
        query = f"UPDATE {table} SET {set_values} WHERE id = %s;"
        values = list(new_data.values())
        values.append(row_id)
        self.cursor.execute(query, tuple(values))
        self.connection.commit()

   Table creation and data retrieving methods are above
```

Some of the aggregation functions

```
def items_for_each_owner(df):
    return df.groupby('owner').size()


def average_length_of_description(df):
    df['description_length'] = df['description'].str.len()
    return df['description_length'].mean()


def filter_items_based_on_owner(df, owner):
    items = df[df['owner'] == owner]
    return items
```