01 Global Healthcare Data ETL & Analysis CLI

Project 1: Global Healthcare Data ETL & Analysis CLI 🌍💉

Overview

Build a robust command-line application that extracts healthcare data from a public API, transforms it into a clean and structured format, loads it into a MySQL database, and allows users to perform basic analytical queries directly from the CLI. This project emphasizes critical data engineering skills like data ingestion, cleaning, and storage for analytical purposes.

Difficulty Level

Advanced | Estimated Time: 20-25 hours

Problem Statement

Create a CLI-based ETL (Extract, Transform, Load) pipeline for healthcare data sourced from a public API. The application should handle data extraction, comprehensive data cleaning and transformation, efficient loading into MySQL, and offer CLI commands for querying the stored data. This mimics a real-world scenario of building a data pipeline for reporting and analysis.

Learning Objectives

- Master interacting with RESTful APIs for data extraction.
- Implement robust data cleaning and transformation techniques using Python.
- Design and manage relational database schemas in MySQL.
- Optimize data loading performance into MySQL.
- Build a user-friendly and functional command-line interface.
- Perform basic data analysis and reporting using SQL queries.
- Implement logging and error handling for data pipeline robustness.

Core Requirements

1. Data Extraction (E)

- Connect to a public healthcare API (e.g., WHO, CDC, or a reliable COVID-19 dataset API).
- Handle API authentication (if required) and rate limits.
- Fetch data based on user-defined parameters (e.g., country, date range, specific metrics).
- Gracefully handle API errors and network issues.

2. Data Transformation (T)

- Parse JSON/XML responses into a structured Python format (e.g., list of dictionaries, Pandas DataFrame).
- Perform data cleaning:
  - Handle missing values (e.g., imputation, removal).
  - Correct data types (e.g., strings to integers/floats, dates to datetime objects).
  - Standardize categorical data (e.g., country names, health indicators).
  - Remove duplicate records.
- Apply business logic transformations (e.g., calculating daily new cases from cumulative, calculating rates).

- Define and enforce a target data schema for MySQL.

3. Data Loading (L)

- Establish a connection to a MySQL database.
- Create necessary tables with appropriate data types and primary/foreign keys (DDL operations).
- Implement efficient batch insertion of transformed data into MySQL.
- Handle incremental loads (only insert new data or update existing records based on unique identifiers).
- Implement error handling for database operations.

4. CLI Interface & Data Analysis

- Develop a command-line interface using `argparse` or `Click`.
- Commands for:
    - `fetch_data [country] [start_date] [end_date]`: Fetches and processes data.
    - `list_tables`: Shows tables in the database.
    - `query_data <query_type> [parameters]`: Runs predefined analytical queries.
        - `query_type examples`: `total_cases <country>`, `daily_trends <country> <metric>`, `top_n_countries_by_metric <n> <metric>`
    - `drop_tables`: Cleans up the database.
- Display query results in a readable tabular format in the CLI.

Technical Specifications

Required Features

Core ETL Pipeline (60 points)

- [ ] **API Integration**: Connects to and fetches data from a chosen public healthcare API.
- [ ] **Data Parsing**: Successfully parses API response (JSON/XML) into structured Python objects.
- [ ] **Data Cleaning**: Implements at least 3 types of data cleaning (e.g., null handling, type conversion, standardization).
- [ ] **MySQL Schema Design**: Creates appropriate database tables with correct data types and constraints.
- [ ] **Data Loading**: Efficiently loads transformed data into MySQL (e.g., `executemany` or `to_sql`).
- [ ] **Incremental Loading**: Logic to avoid re-inserting existing data or update it.
- [ ] **Robust Error Handling**: Handles API errors, database connection errors, and data processing errors.
- [ ] **Logging**: Implements Python `logging` module to track ETL process.

CLI & Reporting (25 points)

- [ ] **CLI Framework**: Uses `argparse` or `Click` for a structured CLI.
- [ ] **Data Fetch Command**: `fetch_data` command with parameters (e.g., country, date range).
- [ ] **Query Commands**: At least 3 pre-defined analytical query commands (e.g., `total_cases`, `daily_trends`).
- [ ] **Tabular Output**: Displays query results clearly in the terminal.
- [ ] **Database Management Commands**: `list_tables`, `drop_tables`.

Bonus Features (Extra 25 points)

- [ ] **Configuration Management**: Uses a `.ini`, YAML, or JSON file for API keys, database credentials, etc.

- [ ] **Data Validation Framework**: Implement a more formal data validation layer (e.g., `cerberus`, custom validators).
- [ ] **Data Lineage/Audit Trail**: Log when data was fetched, by whom, and its source.
- [ ] **Basic Data Visualization (CLI)**: Generate simple ASCII charts for trends within the CLI using libraries like `asciichartpy`.
- [ ] **ETL Scheduling (Mock)**: Simulate a scheduled job within the CLI (e.g., `run_scheduled_etl`).
- [ ] **More Complex SQL Queries**: Implement window functions, CTEs for advanced analytics.

Implementation Guidelines

Suggested Program Structure

```python
import requests
import mysql.connector # or sqlalchemy
import json # or xml.etree.ElementTree
from datetime import datetime, timedelta
import argparse # or click
import logging
import pandas as pd # Highly recommended for data transformation

# Configure logging
logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(levelname)s - %(message)s')

class APIClient:
    """Handles interaction with the external healthcare API."""
    def __init__(self, base_url, api_key=None):
        pass # Initialize client, set headers

    def fetch_data(self, endpoint, params=None):
        """Fetches data from a specific API endpoint."""
        pass # Make API call, handle errors

class DataTransformer:
    """Handles data cleaning and transformation."""
    def __init__(self):
        pass

    def clean_and_transform(self, raw_data, expected_schema):
        """Cleans and transforms raw data into a standardized format."""
        pass # Implement cleaning logic (nulls, types, duplicates)

class MySQLHandler:
    """Manages MySQL database connections and operations."""
    def __init__(self, host, user, password, database):
        pass # Establish connection

    def create_tables(self):
        """Creates necessary tables in the database."""
        pass # Execute DDL statements

    def insert_data(self, table_name, data_list):
        """Inserts a list of dictionaries/records into a specified table."""
        pass # Use executemany for efficiency

    def query_data(self, sql_query, params=None):
        """Executes a SELECT query and returns results."""
        pass # Fetch results

    def close(self):
        """Closes the database connection."""
        pass

class CLIManager:
    """Manages the command-line interface."""
    def __init__(self, api_client, db_handler):
        self.api_client = api_client
        self.db_handler = db_handler
        self.parser = argparse.ArgumentParser(description="Global Healthcare Data ETL &
Analysis CLI")
        self._setup_parser()

    def _setup_parser(self):
        subparsers = self.parser.add_subparsers(dest='command', help='Available commands')

        # ETL Command
        fetch_parser = subparsers.add_parser('fetch_data', help='Fetch, transform, and load
healthcare data.')
        fetch_parser.add_argument('--country', type=str, help='Country to fetch data for
(e.g., "India")', required=True)
        fetch_parser.add_argument('--start_date', type=str, help='Start date (YYYY-MM-DD)')
        fetch_parser.add_argument('--end_date', type=str, help='End date (YYYY-MM-DD)')

        # Query Commands
        query_parser = subparsers.add_parser('query_data', help='Query loaded data.')
        query_subparsers = query_parser.add_subparsers(dest='query_type', help='Types of
queries')

        total_cases_parser = query_subparsers.add_parser('total_cases', help='Get total
cases for a country.')
        total_cases_parser.add_argument('country', type=str, help='Country name.')

        daily_trends_parser = query_subparsers.add_parser('daily_trends', help='Get daily
trends for a metric.')
        daily_trends_parser.add_argument('country', type=str, help='Country name.')
        daily_trends_parser.add_argument('metric', type=str, help='Metric (e.g.,
"new_cases", "deaths").')

        top_n_parser = query_subparsers.add_parser('top_n_countries_by_metric', help='Get
top N countries by a metric.')
        top_n_parser.add_argument('n', type=int, help='Number of top countries.')
        top_n_parser.add_argument('metric', type=str, help='Metric (e.g., "total_cases",
"vaccinations").')

        # DB Management Commands
        subparsers.add_parser('list_tables', help='List tables in the database.')
        subparsers.add_parser('drop_tables', help='Drop all created tables (USE WITH
CAUTION).')

    def run(self):
        args = self.parser.parse_args()
        if args.command == 'fetch_data':
            # Logic to call API, transform, and load
            pass
```

```python
        elif args.command == 'query_data':
            # Logic to execute SQL queries based on query_type
            pass
        elif args.command == 'list_tables':
            # Logic to list tables
            pass
        elif args.command == 'drop_tables':
            # Logic to drop tables
            pass
        else:
            self.parser.print_help()

def main():
    # Database configuration (should ideally be in a config file)
    db_config = {
        'host': 'localhost',
        'user': 'root',
        'password': 'your_password', # Use environment variable or config file!
        'database': 'healthcare_db'
    }

    # API configuration (replace with your chosen API details)
    api_base_url = "https://api.example.com/healthcare/" # Replace with actual API URL
    api_key = "your_api_key" # Replace with actual API key if needed

    db_handler = MySQLHandler(**db_config)
    api_client = APIClient(api_base_url, api_key)

    # Ensure tables exist (students should implement robust check and creation)
    db_handler.create_tables()

    cli_manager = CLIManager(api_client, db_handler)
    cli_manager.run()

    db_handler.close()

if __name__ == "__main__":
    main()
```

Required Files Structure

```
healthcare_etl_cli/
├── main.py                 # Main application entry point and CLI setup
├── api_client.py           # Module for API interaction
├── data_transformer.py     # Module for data cleaning and transformation logic
├── mysql_handler.py        # Module for all MySQL operations
├── config.ini              # Configuration file for DB credentials, API keys (optional
but recommended)
├── requirements.txt        # Required Python packages
├── README.md               # Project documentation
├── sql/                    # Directory for SQL schema scripts
│   └── create_tables.sql
└── docs/                   # Optional: project design documents, API schema, etc.
```

Sample Application Flow

```
# First time setup (create tables)
python main.py list_tables
# Output: No tables found. Creating tables...
# Tables 'cases', 'vaccinations', etc. created successfully.

# Fetching data for India
python main.py fetch_data --country "India" --start_date "2023-01-01" --end_date "2023-01-
31"
# Output: Fetching data for India from 2023-01-01 to 2023-01-31...
# Successfully fetched 31 records from API.
# Cleaned and transformed data.
# Loaded 31 records into 'daily_cases' table.

# Querying total cases for India
python main.py query_data total_cases India
# Output:
# Total COVID-19 Cases in India: 44,997,000

# Querying daily trends for new cases in USA
python main.py fetch_data --country "USA" --start_date "2023-02-01" --end_date "2023-02-07"
# Output: ... (data loaded)
python main.py query_data daily_trends USA new_cases
# Output:
# Date       | New Cases
# -----------|-----------
# 2023-02-01 | 15000
# 2023-02-02 | 14500
# 2023-02-03 | 16200
# ...

# Getting top 3 countries by total vaccinations
python main.py query_data top_n_countries_by_metric 3 total_vaccinations
# Output:
# Rank | Country         | Total Vaccinations
# -----|-----------------|--------------------
# 1    | China           | 3,450,000,000
# 2    | India           | 2,210,000,000
# 3    | United States   | 675,000,000
```

Data Models

MySQL Table Examples (Conceptual)

`daily_cases` **table**

Column Name	Data Type	Constraints	Description
`id`	`INT`	`PRIMARY KEY`, `AUTO_INCREMENT`	Unique record ID
`report_date`	`DATE`	`NOT NULL`, `UNIQUE(country_name, report_date)`	Date of the report
`country_name`	`VARCHAR(255)`	`NOT NULL`	Name of the country
`total_cases`	`BIGINT`	`NULLABLE`	Cumulative reported cases
`new_cases`	`INT`	`NULLABLE`	New cases on that date

Column Name	Data Type	Constraints	Description
`total_deaths`	`BIGINT`	`NULLABLE`	Cumulative reported deaths
`new_deaths`	`INT`	`NULLABLE`	New deaths on that date
`etl_timestamp`	`TIMESTAMP`	`DEFAULT CURRENT_TIMESTAMP`	When record was loaded

`vaccination_data` table

Column Name	Data Type	Constraints	Description
`id`	`INT`	`PRIMARY KEY`, `AUTO_INCREMENT`	Unique record ID
`report_date`	`DATE`	`NOT NULL`, `UNIQUE(country_name, report_date)`	Date of the report
`country_name`	`VARCHAR(255)`	`NOT NULL`	Name of the country
`total_vaccinations`	`BIGINT`	`NULLABLE`	Total doses administered
`people_vaccinated`	`BIGINT`	`NULLABLE`	People with at least one dose
`people_fully_vaccinated`	`BIGINT`	`NULLABLE`	People fully vaccinated
`etl_timestamp`	`TIMESTAMP`	`DEFAULT CURRENT_TIMESTAMP`	When record was loaded

Required Features Detail

1. API Interaction & Error Handling

- Use `requests.get()` with proper URL construction.
- Implement `try-except` blocks for `requests.exceptions.RequestException` and other potential network/API issues.
- Check HTTP status codes (`response.status_code`).

2. Data Transformation Logic

```python
import pandas as pd

def transform_api_data(raw_json_data):
    """
    Transforms raw API JSON data into a clean, standardized Pandas DataFrame
    ready for loading into MySQL.
    """
    df = pd.DataFrame(raw_json_data)

    # Example cleaning:
    # 1. Convert 'date' column to datetime objects
    if 'date' in df.columns:
        df['date'] = pd.to_datetime(df['date'], errors='coerce')
        df = df.dropna(subset=['date']) # Drop rows where date conversion failed

    # 2. Convert numerical columns to appropriate types, handle non-numeric values
    numerical_cols = ['total_cases', 'new_cases', 'total_deaths', 'new_deaths']
    for col in numerical_cols:
        if col in df.columns:
            df[col] = pd.to_numeric(df[col], errors='coerce').fillna(0).astype(int)

    # 3. Standardize country names (if necessary, e.g., 'United States' to 'USA')
    if 'country' in df.columns:
        country_mapping = {'United States': 'USA', 'United Kingdom': 'UK'} # Example
        df['country'] = df['country'].replace(country_mapping).fillna('Unknown')

    # 4. Handle duplicates based on (country, date)
    if 'country' in df.columns and 'date' in df.columns:
        df = df.drop_duplicates(subset=['country', 'date'])

    # Rename columns to match MySQL table
    df = df.rename(columns={'date': 'report_date', 'country': 'country_name'})

    return df
```

3. MySQL Batch Insertion & Incremental Loads

- Use `mysql.connector.cursor.executemany()` for efficient inserts.
- For incremental loads, before inserting, query the database to check if a record with the same `(country_name, report_date)` already exists. If it does, either skip or update it.

```python
# Example for incremental load logic within MySQLHandler.insert_data
def insert_data(self, table_name, data_list):
    cnx = self.conn
    cursor = cnx.cursor()

    # Assuming 'report_date' and 'country_name' are part of the unique key
    # This query checks for existing records
    select_sql = f"SELECT report_date, country_name FROM {table_name} WHERE report_date =
%s AND country_name = %s"

    insert_sql = f"INSERT INTO {table_name} ({', '.join(data_list[0].keys())}) VALUES ({',
'.join(['%s'] * len(data_list[0]))})"

    new_records = []
    for record in data_list:
        # Check if record already exists
        cursor.execute(select_sql, (record['report_date'], record['country_name']))
        if not cursor.fetchone():
            new_records.append(tuple(record.values())) # Prepare tuple for executemany
        else:
            logging.info(f"Skipping existing record: {record['country_name']} on
{record['report_date']}")

    if new_records:
        try:
            cursor.executemany(insert_sql, new_records)
            cnx.commit()
            logging.info(f"Successfully inserted {len(new_records)} new records into
{table_name}.")
        except mysql.connector.Error as err:
            logging.error(f"Error inserting data: {err}")
            cnx.rollback()
    else:
        logging.info("No new records to insert.")

    cursor.close()
```

Testing Checklist

ETL Process

- [ ] API client successfully fetches data.
- [ ] Data transformer handles missing values and incorrect types.
- [ ] Duplicate records are identified and handled.
- [ ] MySQL tables are created correctly on first run.
- [ ] Data is loaded efficiently into MySQL.
- [ ] Incremental loading logic prevents duplicate inserts.
- [ ] Error messages are clear for API, transformation, and DB issues.

CLI Functionality

- [ ] `fetch_data` command works with various parameters.
- [ ] `query_data` commands return accurate results for all query types.
- [ ] `list_tables` correctly displays database tables.
- [ ] `drop_tables` clears the database (test on a non-production DB!).
- [ ] CLI arguments are validated (e.g., date formats, integer `n`).
- [ ] Output formatting is clear and readable.

Code Quality

- [ ] Proper use of classes and modular design.
- [ ] Docstrings for all classes and functions.
- [ ] Meaningful variable and function names.
- [ ] Consistent coding style (PEP 8).
- [ ] Robust error handling for all critical paths.
- [ ] Effective use of logging.

Required Python Packages

Create `requirements.txt`:

```
requests
mysql-connector-python
pandas
# click>=8.0.0 # If using Click instead of argparse
```

Install with: `pip install -r requirements.txt`

Evaluation Criteria

Criteria	Excellent (A)	Good (B)	Satisfactory (C)	Needs Work (D/F)
Data Extraction	Robust API handling, comprehensive data fetch	Good API interaction, minor error handling gaps	Basic API fetch works	Fails to fetch data reliably
Data Transformation	Comprehensive cleaning, handles edge cases, uses Pandas effectively	Good cleaning, some minor issues	Basic transformations work	Data remains messy or incorrect
Data Loading	Efficient batch inserts, robust incremental loads, proper schema	Good data loading, some efficiency gaps	Basic inserts work	Data integrity issues, slow loading
Database Design	Well-normalized, indexed, appropriate data types	Good schema, minor optimizations needed	Basic table structure	Poorly designed, inefficient
CLI & Analysis	Intuitive CLI, diverse & accurate analytical queries	Functional CLI, good query results	Basic CLI, few queries	Unusable CLI, incorrect queries
Error Handling & Logging	Comprehensive error handling, detailed logging	Good error messages, useful logs	Basic try-except blocks	Errors crash application, no logs
Code Quality	Excellent modularity, clean code, well-documented	Good structure, readable, documented	Adequate organization, some docs	Poor structure, hard to maintain

Submission Requirements

What to Submit

1. **Source Code**: All Python files, organized into modules.
2. **SQL Scripts**: `create_tables.sql` file containing the DDL for the database schema.
3. **Configuration File**: `config.ini` (or equivalent) for database and API credentials.
4. **Requirements File**: `requirements.txt` with all dependencies.
5. **Documentation**: `README.md` with:
    - Installation and setup instructions (including MySQL setup).
    - Details of the chosen public API and its data.
    - CLI command usage guide with examples.
    - Database schema description.
    - ETL process description.
    - Known limitations and future improvements.
6. **Demo Screenshots**: Screenshots of CLI usage showing data fetching and query results.

Testing Data

- Document the specific API endpoint(s) used.
- Describe the expected data format from the API.

- Provide examples of transformed data structures.

Code Quality Requirements

- **Modularity**: Separate concerns into different Python modules (e.g., `api_client.py`, `mysql_handler.py`).
- **Error Handling**: Implement `try-except` blocks for all I/O, network, and database operations.
- **Input Validation**: Validate CLI arguments and user inputs.
- **Documentation**: Use docstrings for all classes, methods, and complex functions.
- **Readability**: Follow PEP 8 guidelines for code style.

Advanced Challenges (Optional)

For students who finish early or want extra challenges:

1. **Data Quality Checks**: Implement a module to run data quality checks after loading (e.g., ensure no negative values for cases, check data freshness).
2. **Alerting**: Integrate with a simple notification system (e.g., print to console, or send a mock email) if data quality issues are found or ETL fails.
3. **Visualization Integration**: Generate a simple HTML/CSV report from the queried data that can be opened in a browser or spreadsheet software.
4. **ETL Orchestration (Conceptual)**: Discuss how a tool like Apache Airflow or Prefect would orchestrate this pipeline in a production environment.
5. **Multi-source Integration**: Integrate data from a second, different public healthcare API and join it in MySQL.

Common Pitfalls to Avoid

1. **Hardcoding Credentials**: Never hardcode API keys or database passwords directly in the code. Use environment variables or a config file.
2. **Inefficient DB Inserts**: Avoid row-by-row inserts in a loop; use `executemany` for performance.
3. **Ignoring API Rate Limits**: Respect the API's rate limits; implement delays if necessary.
4. **Ignoring Data Schema**: Load data into MySQL tables that do not match the expected schema from the API or transformed data, leading to errors.
5. **Lack of Error Handling**: Unhandled exceptions will crash the application.
6. **No Logging**: Without logging, debugging ETL failures becomes very difficult.

Resources

- Requests Library Documentation
- MySQL Connector/Python Documentation
- Pandas Documentation
- Python `argparse` Tutorial
- Python `logging` Tutorial
- Database Normalization Concepts
- Example Public APIs (for inspiration, verify terms of use): WHO, Our World in Data, Johns Hopkins CSSE COVID-19 Dataset (often available via GitHub/APIs)

Academic Integrity

This project requires significant original implementation. While discussion of general approaches with classmates is encouraged, direct sharing of code is strictly prohibited. Proper attribution for any external code snippets or ideas is mandatory.

Happy coding, and may your data pipelines run smoothly! 📊 💪