Project 3: Cleaning and Enrichment Batch Job
CSC 6605 ML Production Systems

Overview

In this part of the project, you're going to implement a batch pipeline that consumes records from one set of tables, cleans and transforms the records, enriches the records with additional data, and inserts the resulting records in a second table. The cleaned and enriched data will be used for training a model.

Instructions

Part I: Create Tables for Storing Cleaned Events

You are going to use two tables for tracking the cleaned and enriched events. One table will contain the cleaned events. The ids should match the ids used in the raw events tables. You developed the basic schema in project 2. You will need to add columns for the home sale events from the population and public school tables. The second table will simply contain the ids of processed events (whether successfully cleaned or not). Unprocessed events will be identified by querying the raw events tables for ids that are not in the processed events table. Make sure that you create indices on the id columns.

Part II: Cleaning and Enrichment Batch Job Script

In your repository, create a directory named "cleaning-and-enrichment-job." In that directory, create a Python script cleaning_job.py that will:

1. Query unprocessed home sale events from the raw home sale events table by looking for ids not in the processed ids table
2. Query the population and public school tables for the home sales events' zip codes
3. For each home sale event:
    a. Enrich the event with the population and public school data
    b. Deserialize and validate the JSON object using marshmallow library
    c. Insert deserialized object into the cleaned home sale events tables (if there are no issues)
    d. Record id in the processed ids tables

The script should take PostgreSQL credentials and host information from environmental variables.

Define a data model (schema) using the marshmallow library. Your data model should include:

a. Fields with appropriate types and marks as required versus not
b. Pre-processing methods annotated with the @pre_load decorator that convert types, replace missing value placeholders with None values, and rename fields.
c. Validation methods annotated with the @validates decorators for checking ranges of values.

Robustness

It's important that your batch job be robust to failures. Cases you will need to handle include:

1. Ensure that the most recent records are processed first. There may be more records available to process than you can reasonably process. If you don't limit the number of records you process in each batch and ensure that you grab the most recent records first, your system may back up.

2. Some records will not pass the validation checks. These records should be marked as processed even though a corresponding entry in the cleaned events tables will not be created.

3. Do not let failures on individual records crash the entire batch job. Process each record individually, and perform a commit operation after each record is processed. Catch ValidationError exceptions and handle them appropriately.

4. Do not leave your system in an inconsistent state. The batch job writes to two separate tables. If the job is killed at precisely the right time, a record will be written to one table and not another. To prevent this, wrap both insertions in a SQL transaction that will create a single atomic operation. The psycopg library always starts transactions you automatically. No writes will be persisted to the database until conn.commit() is called.

Testing Hint
While developing the pipeline, you can comment out or exclude the conn.commit() line. (Or create a command-line argument for doing so.) This will allow you to run the pipeline without writing the results to the database. This is a good way to double check that all exceptions are caught.

Part III: Wrap the Batch Job with APScheduler

The batch job will need to be run periodically. The cron service included in many Linux distributions is one way to accomplish this. Cron does not prevent multiple instances of a job from running, however. If the batch job is still running when the next time period occurs, cron will start a second instance. It can be difficult to ensure that multiple instances are not processing the same records. Alternatives such as Apache Airflow are nice because they provide web interfaces for tracking all jobs in a single place. I happen to prefer the Python APScheduler library which enables control over the number of instances, saves state in an external database in case of failures, and has third-party web interfaces available.

Hints
1. You will create a separate job runner Python script.
2. Restructure your batch job script to export two functions: check_environment() to check that the necessary environmental variables are defined and run_job() that runs the actual batch job. The if __name__ == "__main__" syntax can be used to run code if the batch job script is executed directly. The job runner script should import the functions from the batch script, call check_environment() before creating a scheduler, and then add run_job() as a job.
3. Use the BlockingScheduler since it will be the scheduler will be the only thing running in the job runner script.
4. Pass the max_jobs=1 and coalesce=True parameters when calling the add_job() method to ensure that only one instance of the batch job runs at a time.
5. Set the job to run every 15 minutes.
6. Use the following code snippet after adding the job to involve it immediately, so you don't have to wait 15 minutes when developing:

```
# start all jobs immediately instead of waiting for end of first interval
    for job in scheduler.get_jobs():
        job.modify(next_run_time=dt.datetime.now())
```

7. Call scheduler.start() at the end of your script

Part IV: Run the Batch Job in a Docker Container

From your VM, you can create a Docker image, test it, and deploy it. Note that since the server uses the arm64 architecture, Docker images must be built on an arm64 system. If you try to build the Docker image on your laptop (which uses the x86_64 architecture unless you have an Apple Silicon Mac), it won't run on the VM.

1. Create a personal access token for your GitHub repository and use it to check out your repository on your VM.

2. In the cleaning-and-enrichment-job directory, create a Dockerfile:

```
FROM python:3.12-slim

WORKDIR /app
COPY . /app/

RUN pip3 install -U pip wheel
RUN pip3 install "psycopg[binary]" apscheduler marshmallow

CMD ["python3", "job_runner.py"]
```

Make sure to commit your Dockerfile to your repository.

3. From the cleaning-and-enrichment-job directory, build your Docker image:

```
$ docker build --no-cache -t cleaning-and-enrichment-job .
```

4. Run the Docker container.

5. Check the output using the Docker logs command and checking the tables in PostgreSQL for processed events.

Part V: Write Documentation

In your GitHub repository, make sure that you have a README.md file in your cleaning-and-enrichment-job directory that contains:

1. Instructions on running the cleaning and enrichment script (standalone) and building and running the Docker image.
2. Descriptions of the database schemas.

Demonstration and Submission

Submit screenshots of the following to Canvas:

- `docker ps` output showing the batch job container running for at least 10 minutes
- PostgreSQL `\d tablename` output for your tables
- Output from selecting 10 records from your tables

Your instructor will review your code in your GitHub repository.

Rubric

Description	Percentage
Development Process • Commits are small and self-contained with reasonable messages. • Commits are contributed through feature branches and pull requests rather than direct commits to the main branch. • Commits were peer-reviewed and approved by other group members. • All source code is committed to the repository.	10%
Documentation • The cleaning and enrichment job has a separate README.md file • The README.md file describes how to build and run the batch job script (standalone) • The README.md file describes how to build and run the Docker image • The README.md describes the schemas for the tables used by the batch job.	10%
PostgreSQL • Tables for cleaned events are present. • Tables for processed ids are present. • Tables follow correct schemas including definition of appropriate fields, types, and nullability. • Id fields are indexed.	10%
Schema • Schema is defined using Marshmallow. • Schema has appropriate fields (including fields from enrichment data sets). • Appropriate fields are marked required versus not. • Schema has appropriate pre-load functions. • Schema has appropriate validation functions.	10%
Data Enrichment Batch Job Script • Queries PostgreSQL for unprocessed records. • Queries PostgreSQL for enrichment records. Handles cases where enrichment records are missing by filtering out record. • Enriches home sale events with population, school, and interest rate data. • Handles data cleaning issues by filtering out or fixing records. • Inserts processed record ids into appropriate table (even if a record was filtered out). • Takes credentials as environmental parameters. • Can be run standalone on the command-line. • Job is robust to failures that occur for individual records.	25%
Job Runner • APScheduler library is employed. • Creates a BlockingScheduler. • Adds batch job to scheduler. • Sets the job to run every 15 minutes, once maximum instance, and to coalesce jobs that overrun their scheduled times. • Schedules batch job to run immediately. • Starts scheduler.	25%

Batch Job Deployment • Dockerfile is present in the repository. • Can successfully build a Docker image. • Docker container is running on the VM. • Cleaned and enriched records are written to output tables in PostgreSQL.	10%