**SQL Implementation and Data Pipeline Development**

**Overview of the Approach**

This project involved developing a full-stack fraud detection dashboard, backed by a MySQL database and built using Python and Streamlit. The two core scripts powering the database setup and data population were:

* Create\_Tables.py: For programmatically defining the database schema.
* bulk\_loader.py: For automating the ingestion of synthetic CSV data into MySQL.

I intentionally opted for code-driven SQL creation and loading to streamline the workflow, support quick iteration, and ensure full control over schema logic and data integrity enforcement.

**1. Table Creation via Create\_Tables.py**

Rather than writing static .sql files, I wrote a Python script (Create\_Tables.py) that connects to MySQL and executes a series of CREATE TABLE statements. This made it easy to:

* Apply consistent naming conventions.
* Add constraints like PRIMARY KEY, FOREIGN KEY, and NOT NULL.
* Automate the schema rebuild each time I updated the logic or schema relationships.

**Challenges Encountered:**

* **Foreign Key Dependencies:** Because many tables referenced each other (e.g., customer\_account\_data referencing both customer\_data and account\_data), the creation order was critical. I had to debug several constraint errors and reorder my statements to avoid breaking referential integrity.
* **BIGINT in card\_data:** Initially, the card\_data table stored full card numbers as BIGINT. This seemed reasonable for synthetic data, but it created several issues:
  + **Display Issues in the UI:** Full card numbers weren't ideal for visualization. For realism and security mimicry, I decided to show only the **last 4 digits**.
  + **Inserts Failing or Truncating:** Some randomly generated card numbers exceeded the safe integer range for MySQL BIGINT, leading to silent truncation or out-of-bounds errors.
  + **Solution:** I modified both the schema and my data generation logic to store only the last four digits (VARCHAR(4)) and ensure formatting matched real-world constraints.

**2. Bulk Loading with bulk\_loader.py**

Once the schema was created, I needed to populate the tables with mock transactional data. The bulk\_loader.py script was responsible for this step. It:

* Scanned for each .csv file (e.g., customer\_data.csv, fraud\_alert\_data.csv).
* Read each file into a Pandas DataFrame.
* Executed efficient INSERT operations using parameterized SQL with MySQL’s executemany().

**Challenges Encountered:**

* **Partial Table Success:** Only about 60% of the CSVs loaded successfully on the first attempt. Common reasons included:
  + **Column Count Mismatches:** In some CSVs, the header did not match the number of columns defined in the SQL schema.
  + **Foreign Key Violations:** Some records referred to parent keys that hadn’t been inserted yet.
  + **Empty or Null Values:** Some tables had empty strings that violated NOT NULL constraints in the schema.
* **Debugging Approach:**
  + I printed the first few rows of each CSV before loading.
  + I implemented error catching per-table, so a failure in one table wouldn’t prevent others from loading.
  + I used dummy values or reordered inserts to satisfy foreign key dependencies.

**Key Learnings and Takeaways**

* **Programmatic Table Creation is Powerful but Risky:** While it accelerated schema evolution, small bugs (like typos or mismatched types) had cascading effects across many tables.
* **Using Python for SQL Has Tradeoffs:** It’s flexible, but debugging SQL errors from Python adds layers of complexity compared to raw SQL.
* **Good Data Design Is Critical:** Thinking about realistic and useful data representations (e.g., card numbers, device identifiers) early could have saved time.