**SQL Implementation**

**Schema Design and DDL**

To support the fraud detection system, we implemented several key tables using CREATE TABLE statements, including:

* parties(party\_id, name, date\_joined, ...)
* accounts(account\_id, party\_id, open\_date, ...)
* logins(login\_id, party\_id, device\_id, location, login\_time, failed\_login)
* alerts(alert\_id, party\_id, alert\_type, alert\_time)
* devices(device\_id, device\_type, ...)

Each table includes appropriate **primary keys**, **data types**, and **NOT NULL constraints** to ensure data quality.

**Challenges & Lessons Learned**

While building the SQL schema and Streamlit dashboard, several **shortcomings and technical difficulties** emerged. These experiences helped refine my understanding of database design and real-world data management:

**Random BIGINT Values for Account Numbers**

One of the major issues was that the mock data generator used full-length **BIGINT account numbers**—often 10 to 16 digits long. This caused several problems:

* **Visualization clutter**: Displaying full account numbers in charts or tables made the dashboard unreadable.
* **Unrealistic output**: Users don't typically need to see full account numbers; the last 4 digits are sufficient and safer for privacy.

**Solution:** I applied a transformation in the application layer (or SQL) to extract only the last 4 digits using a modulus operation:

RIGHT(account\_id, 4) AS last\_four\_digits

-- or in Python:

str(account\_id)[-4:]

**Duplicate Tables in the Dataset**

The mock schema came with several **duplicated or overlapping tables**, such as multiple versions of login or account tables with slightly different names or columns. This created confusion:

* Which version was the "official" or canonical source?
* Foreign key relationships could easily break if I used the wrong table.

**Solution:** I cleaned the schema by dropping duplicates and standardizing naming conventions. I retained only the most complete or best-structured versions of each table. This also helped streamline joins and queries later in the application.

**Foreign Key Complexity and Data Integrity**

While setting up **foreign key constraints**, I encountered issues where mock data violated these relationships:

* Devices or parties referenced in logins didn’t exist in devices or parties
* Data load failed due to missing referenced rows

**Lesson learned:** With mock data, it’s essential to **validate the order of inserts** and ensure referenced rows exist **before** inserting dependent data. I adjusted the script execution order and added referential checks to avoid these errors.

**Sparse Data for Certain Queries**

Some queries, like:

* **Accounts with >5 failures (1 day)**
* **High Device Failures**
* **Multi-Location Access in 60 Minutes**

… returned **no rows**, which was initially confusing. It seemed like a failure.

**Resolution:** I realized this was an artifact of the limited mock dataset. In production systems, these patterns **would** exist. I added conditional logic in Streamlit to gracefully handle and explain empty results, keeping the UX informative even when datasets are sparse.

**Future Improvements**

* Use **smarter mock data generators** that ensure referential integrity and populate meaningful edge cases
* Add **ETL validations** to detect bad foreign keys or missing references
* Build **test cases** for query logic to ensure robustness across empty and full datasets

**Integrity Constraints**

We enforced **referential integrity** using **foreign key constraints**, ensuring consistency between related entities. For example:

INSERT INTO parties (party\_id, name) VALUES (123, 'John Doe');

INSERT INTO accounts (account\_id, party\_id) VALUES (456, 123);

INSERT INTO logins (login\_id, party\_id, failed\_login) VALUES (1, 123, TRUE);

This allowed for testing edge cases such as:

* Multiple failed logins
* Logins from different devices
* High alert counts

**Application Layer (Streamlit UI)**

**Basic Functionality**

A **Streamlit dashboard** was created to visualize fraud-related patterns with the following features:

* Dropdown to select query/analysis
* Tabular display of results (e.g., multiple alerts, high failed logins)
* Summary statistics and bar chart visualizations
* Conditional warnings if no results were found (e.g., for mock datasets)

**Query Logic Highlights**

See attached files (UI\_screenshots.zip, SQL\_Queries.zip)