FRAUD DETECTION AND RISK ANALYTICS USING SQL

BY ADITI MEENA

INTRODUCTION

This project demonstrates the use of SQL for fraud detection and financial data analysis by applying advanced querying techniques such as Recursive Common Table Expressions (CTEs), window functions, and multi-step joins. The dataset used is *PaySim*, a simulated dataset that mimics mobile money transactions, originally based on real financial transaction data but anonymized and scaled for research purposes.

The project addresses multiple aspects of fraud analytics: detecting recursive chains of money laundering, calculating rolling sums of fraudulent activity over time, identifying suspicious account behaviors using multiple CTEs, validating data consistency between computed and actual balances, and detecting zero-balance anomalies before or after transactions. These insights illustrate how SQL can be effectively applied in real-world risk management, compliance, and financial crime detection.

1. Detecting Recursive Fraudulent Transactions

Question:

Use a recursive CTE to identify potential money laundering chains where money is transferred from one account to another across multiple steps, with all transactions flagged as fraudulent.

Solution:

1. A recursive Common Table Expression (CTE) named fraud_chain is used to trace chains of money laundering.
2. The anchor member of the CTE selects the initial fraudulent transactions, where isFraud is 1 and type is 'TRANSFER'.
3. The recursive member then joins the fraud_chain with the transactions table to find subsequent transfers from the destination of the previous transaction, effectively building a chain.

```sql
--Recursive CTE for Fraud Detection
WITH RECURSIVE fraud_chain (initial_account, next_account, step) AS(
    --Anchor member
        SELECT
            nameorig AS initial_account,
            namedest AS next_account,
            step
        FROM transactions
        WHERE isfraud = 1 AND type = 'TRANSFER'

        UNION ALL
        --Recursive member
        SELECT
            fc.initial_account,
            t.namedest,
            t.step
        FROM fraud_chain AS fc
        INNER JOIN transactions AS t
        ON fc.next_account = t.nameorig
        WHERE isfraud= 1 AND fc.step < t.step
        AND t.type= 'TRANSFER'
)
SELECT * FROM fraud_chain
--Aditi Meena
```

2. **Analyzing Fraudulent Activity over Time**

Question:

Use a CTE to calculate the rolling sum of fraudulent transactions for each account over the last 5 steps.

Solution:

- A Common Table Expression (CTE) named rolling_fraud is used to calculate the rolling sum.
- The SUM function with PARTITION BY nameOrig and ORDER BY step groups the data by account and orders it by time.
- ROWS BETWEEN 4 PRECEDING AND CURRENT ROW specifies the window of the last five steps (the current row and the four preceding rows) to be included in the sum.

```sql
-- Analyzing Fraudulent Activity over Time
WITH rolling_fraud AS (
    SELECT
        nameorig,
        step,
        SUM(isFraud) OVER (
        PARTITION BY nameorig
        ORDER BY step
        ROWS BETWEEN 4 PRECEDING AND CURRENT ROW ) AS onlyfrauds
    FROM transactions
)
SELECT *
FROM rolling_fraud
Where onlyfrauds >0
-- Aditi Meena
```

3. **Complex Fraud Detection Using Multiple CTEs**

Question:

Use multiple CTEs to identify accounts with suspicious activity, including large transfers, consecutive transactions without balance change, and flagged transactions

Solution:

The query uses multiple CTEs to isolate different suspicious patterns:

1. lt finds large transfer transactions greater than the average legitimate amount.

2. nbc finds transactions where the sender's balance didn't change, and ft finds transactions flagged as fraud.

3. The final JOIN returns accounts that appear in all three sets, i.e., highly suspicious fraud cases.

```sql
-- Complex Fraud detection using multiple CTE
-- lt = large transactions, nbc= no balance change,ft= fraud transactions
WITH lt AS (
    SELECT
    nameorig,
    step,
    amount
    FROM transactions
    WHERE type = 'TRANSFER'
    AND amount >(SELECT AVG(amount) FROM transactions WHERE isfraud=0)),
nbc AS (
    SELECT
    nameorig,
    step,
    oldbalanceorig,
    newbalanceorig
    FROM transactions
    WHERE oldbalanceorig=newbalanceorig),
ft AS (
    SELECT
    nameorig,
    step
    FROM transactions
    WHERE isflaggedfraud = 1)
SELECT lt.nameorig, nbc.oldbalanceorig, nbc.newbalanceorig FROM lt
JOIN nbc ON lt.nameorig = nbc.nameorig AND lt.step = nbc.step
JOIN ft ON lt.nameorig = ft.nameorig AND lt.step = ft.step
-- Aditi Meena
```

4. Checks whether the **computed new_updated_Balance** is the **same** as the **actual newbalancedest** in the table. If they are equal, it returns those rows.

Solution:

- The SQL query checks for data consistency by comparing a computed value with a stored value.
- It calculates the new balance by adding old balance at destination account and amount transferred into destination account , and then uses the **WHERE** clause to find rows where this calculated value exactly matches the newbalancedest column.
- This effectively validates the integrity of the data in the transactions table.

```sql
-- Checking if computational values of new balance at destination = provided newbalancedest

SELECT *, (oldbalancedest + amount) AS new_updated_balance FROM transactions
WHERE (oldbalancedest + amount) = newbalancedest
-- Aditi Meena
```

5. **Detect Transactions with Zero Balance Before or After**

Question: Find transactions where the destination account had a zero balance before or after the transaction.

Solution:

- This SQL query is designed to find all transactions where the destination account had a zero balance at some point, either before or after the transaction occurred.
- It selects the originator's name *nameorig* , the old balance at the destination account *oldbalancedest*, and the new balance at the destination account *newbalancedest* from the transactions table.
- The WHERE clause uses an OR condition to check two possibilities: *oldbalancedest = 0 or newbalancedest = 0.*
- This logic effectively filters the dataset to return only the rows that satisfy either of those conditions, directly answering the question of finding transactions with a zero balance before or after.

```
-- Checking Transactions with Zero Balance Before or After at destination account

SELECT nameorig, oldbalancedest, newbalancedest FROM transactions
    WHERE oldbalancedest=0
    OR newbalancedest=0;
-- Aditi Meena
```

THANK YOU