Final Project Report

STUDENT(S):		
	HAYDER ABBOODI	– HAAO22@STUDENT.BTH.SE
	ALI REZA SHARIFI	– ALSF22@STUDENT.BTH.SE

1. PROJECT IDEA

For this assignment, we have designed and implemented a budget tracking system to help users manage their personal finances. The system allows users to track income and expenses across different categories, view transaction histories, generate monthly and yearly summaries, and receive warnings for negative balances. The data is generated through manual inserts and a test data generator function in the Python implementation, simulating real-world financial transactions over several months. The main users are individuals who want to monitor their spending habits, such as students or young professionals. This idea fits well because it solves the problem of overspending by providing insights into financial patterns, helping users avoid debt. Key features include adding users, categories, and transactions; viewing detailed transaction lists; calculating category totals and income/expense summaries; and automatic warnings via triggers for negative balances.

2. SCHEMA DESIGN

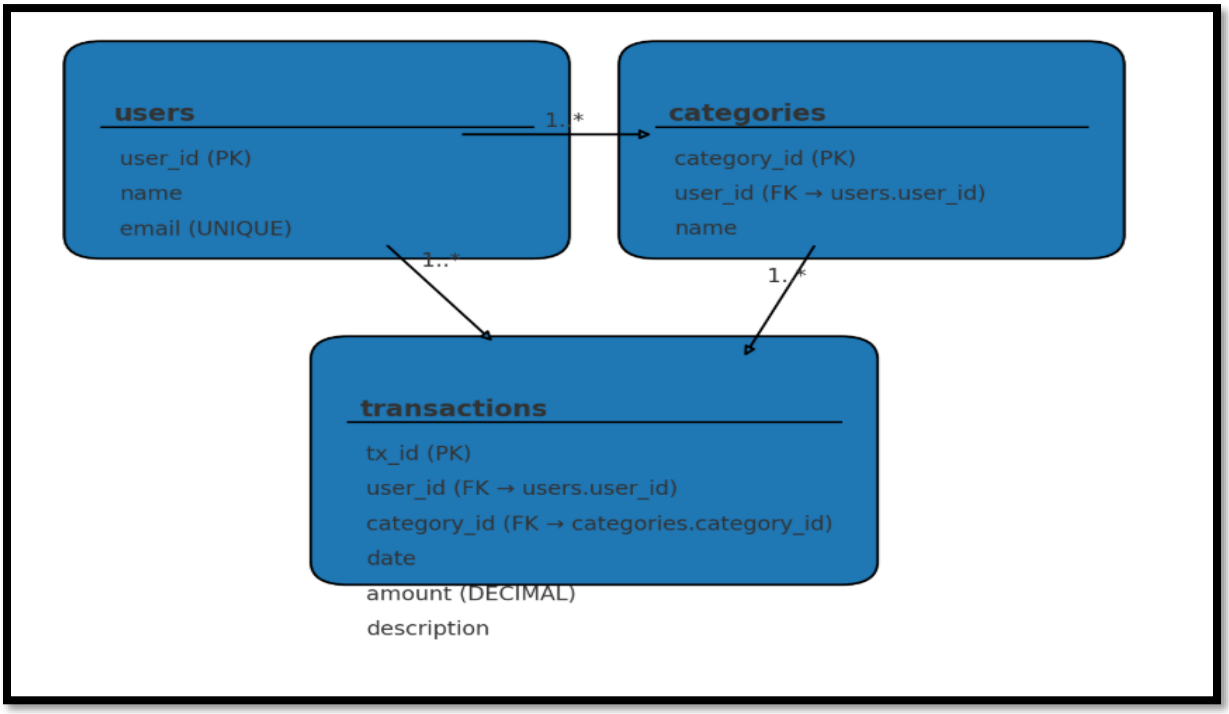
The logical model is designed as an Entity-Relationship (E/R) diagram to represent users, categories, and transactions. The main entities are:

- **Users**: Represents individuals using the system, with attributes like user_id (primary key), name, and email (unique).
- **Categories**: Represents financial categories (e.g., Hyra’, Mat’), with attributes category_id (primary key), user_id (foreign key to Users), and name. Each category belongs to a user.
- **Transactions**: Represents financial entries, with attributes tx_id (primary key), user_id (foreign key to Users), category_id (foreign key to Categories), date, amount (positive for income, negative for expenses), and description.
- **Warnings**: An additional table for storing alerts, with warning_id (primary key), tx_id (foreign key to Transactions), user_id (foreign key to Users), message, and created_at.

Relationships:

- A User can have many Categories (one-to-many).
- A User can have many Transactions (one-to-many).
- A Category can have many Transactions (one-to-many).
- Transactions can trigger Warnings (one-to-many, optional).

This design ensures data integrity with cascading deletes, preventing orphaned records. We motivated this by focusing on user-specific data isolation and efficient querying for summaries.



3. SQL QUERIES

Here we present and discuss the most interesting queries. We have implemented more than five, but focus on key ones that meet the guidelines: at least two multi-relation queries with JOIN, aggregation/grouping, and use of triggers and functions.

Query 1: View Transactions (Basic) This multi-relation query joins Transactions and Categories to list a user's transactions with category names. It uses JOIN to combine data from two tables.

```sql
SELECT t.date, c.name AS category, t.amount, t.description
FROM transactions t
JOIN categories c ON t.category_id = c.category_id
WHERE t.user_id = user_id;
```

Motivation: Essential for users to review their transaction history, showing dates, categories, amounts, and descriptions.

Query 2: View Transactions with User Name This multi-relation query joins Transactions, Users, and Categories to include the user's name. It uses two JOINs for data from three tables.

```sql
SELECT t.date, u.name AS user_name, c.name AS category, t.amount,
t.description
FROM transactions t
JOIN users u       ON t.user_id = u.user_id
JOIN categories c  ON t.category_id = c.category_id
WHERE u.user_id = user_id;
```

Motivation: Useful for administrative views or reports where user identification is needed alongside transaction details.

Query 3: Monthly Category Totals This query uses aggregation (SUM) and grouping (GROUP BY) to sum amounts per category for a specific month, joining Transactions and Categories.

```sql
SELECT c.name AS category, SUM(t.amount) AS total
FROM transactions t
JOIN categories c ON t.category_id = c.category_id
WHERE t.user_id = user_id
  AND YEAR(t.date) = YEAR(CURRENT_DATE())
  AND MONTH(t.date) = month
GROUP BY c.name;
```

Motivation: Helps users analyze spending by category in a given month, identifying areas of high expense.

Query 4: Yearly Income/Expense Summary per Month This uses aggregation (SUM with CASE) and grouping (GROUP BY month) to separate income and expenses monthly.

```
SELECT MONTH(date) AS month,
       SUM(CASE WHEN amount >= 0 THEN amount ELSE 0 END) AS total_income,
       SUM(CASE WHEN amount <  0 THEN -amount ELSE 0 END) AS total_expense
FROM transactions
WHERE user_id = user_id
  AND YEAR(date) = YEAR(CURRENT_DATE())
GROUP BY MONTH(date);
```

Motivation: Provides a yearly overview of financial health, showing monthly inflows and outflows.

Query 5: Trigger for Negative Balance Check This trigger runs after inserts on Transactions, calculating the balance and inserting a warning if negative. It uses aggregation (SUM).

```
DELIMITER $$
CREATE TRIGGER check_negative_balance
AFTER INSERT ON transactions
FOR EACH ROW
BEGIN
  DECLARE current_balance DECIMAL(16,2);
  SELECT IFNULL(SUM(amount),0) INTO current_balance
    FROM transactions
    WHERE user_id = NEW.user_id;
  IF current_balance < 0 THEN
    INSERT INTO warnings(tx_id, user_id, message)
      VALUES (NEW.tx_id, NEW.user_id, CONCAT('Saldo negativt: ',
current_balance));
  END IF;
END $$
DELIMITER ;
```

Motivation: Automatically alerts users to potential overdrafts, enhancing proactive financial management.

Additional: Function for Monthly Balance This function calculates a user's balance for a specific month and year using aggregation.

```
DELIMITER $$
CREATE FUNCTION monthly_balance(u_id INT, m INT, y INT)
  RETURNS DECIMAL(10,2)
  DETERMINISTIC
BEGIN
  DECLARE bal DECIMAL(10,2);
  SELECT SUM(amount) INTO bal
    FROM transactions
    WHERE user_id = u_id
      AND MONTH(date) = m
      AND YEAR(date) = y;
  RETURN IFNULL(bal, 0);
END $$
DELIMITER ;
```

Motivation: Reusable for quick balance checks, integrated into the Python app for summaries.

4. DISCUSSION AND RESOURCES

The implementation uses a console-based Python interface with mysql.connector to interact with the database. We explicitly wrote all queries without ORM, as required. Challenges included handling positive/negative amounts for income/expenses and ensuring the trigger fires correctly for warnings. Test data was inserted manually and generated via a function to simulate realistic scenarios, including cases triggering negative balances.

The project uses the mysql-connector-python library; installation details are standard via pip.

Source code: https://github.com/01alireza/DV1663-Final-Project.git

Video demonstration: https://youtu.be/E8_KFMLA9Cg?si=8J0RXvQ3izW_cq6O

CHANGELOG

Person	Task	week
Hayder	Designed database schema and created tables	29-30
hayder	Implemented SQL queries, trigger, and function	31
Ali	Developed Python console application and functions	32
Hayder/ Ali	Inserted test data and tested warnings	33-34
Ali	Wrote project report	34