Database Project Report

Eva Mukherjee and Maria Joseph

April 2025

1 Overview

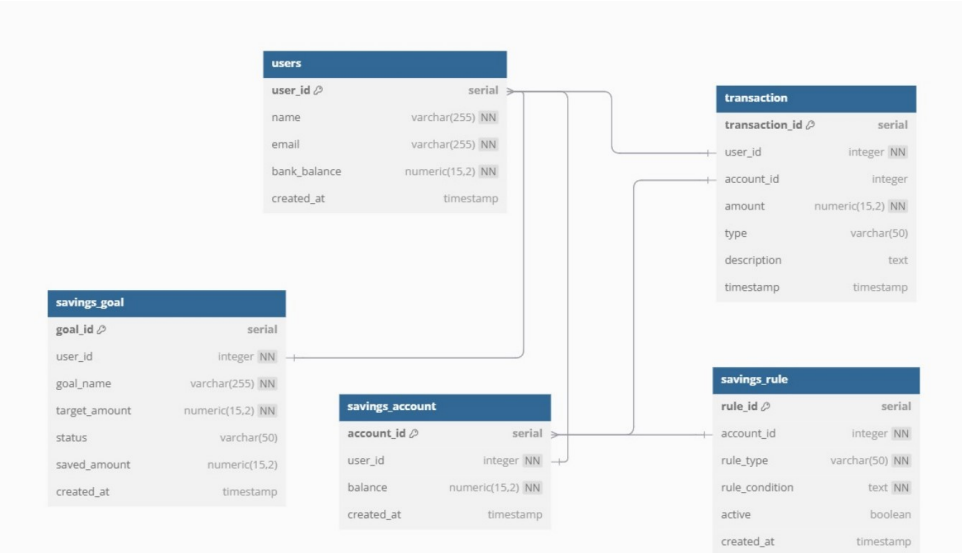
The Automated Savings System is an SQL-based financial web application designed to help users manage their personal finances more efficiently. The platform enables users to automate savings, track daily transactions, set personalized financial goals, and gain insights into their financial habits through analytics.

The platform uses PostgreSQL as the core database system, storing structured data such as user profiles, transaction logs, savings goals, and rule configurations. The backend is developed using Python (Flask), which processes user requests and handles all the logic, such as handling routes, processing input, and applying savings rules. The frontend is built with HTML and CSS, providing a user interface for entering user details, adding transactions and setting savings goals and rules.

2 Database Design

The database for the Automated Savings System is built upon a relational model, designed to effectively manage user information, financial transactions, savings accounts, automated saving rules, and personalized savings goals. The relational model is particularly well-suited for this application due to its ability to structure data into well-defined tables with clear relationships, thereby ensuring data integrity and facilitating efficient querying and data manipulation.

2.1 Entity-Relationship (ER) Diagram Description:



- The core entity is the User. A User can have one or more Savings Account (One to many relationship).A Savings_Account is associated with one User.
- A User can set multiple Savings_Goals (one to many relationship). Each Goal is linked back to the User.

- A User can have multiple Transactions (One to Many relationship). Each Transaction is associated with a specific User and a particular Savings Account (Many to one relationship).
- A Savings_Account can have multiple Transaction records associated with it (representing deposits/withdrawals from savings).
- Lastly, a Savings_Account can have several associated Savings_Rules (one to many relationship), allowing for multiple automated saving mechanisms for a single account.

2.2 Relational Model and Design Choices:

The database is structured using a relational model, where each entity is represented as a table. Relationships between these entities are established using primary and foreign keys, which are fundamental design choices for maintaining data integrity and linking related information across tables.

- **Primary Keys:** Each table has a primary key (e.g., User_ID, Transaction_ID, Account_ID, Rule_ID, Goal_ID) that uniquely identifies each row in the table. This is essential for referencing specific records.
- **Foreign Keys:** Foreign keys are used to create links between tables. For instance, the Transaction table includes User_ID and Account_ID as foreign keys, referencing the User and Savings_Account tables, respectively. This indicates which user initiated the transaction and which account it affected. Similarly, Savings_Account, Savings_Rule, and Savings_Goal tables use User_ID or Account_ID as foreign keys to link them back to the owning user or the relevant savings account.

2.3 Relational Database Schema:

- **Users:** This table represents individuals who own a savings account and initiate transactions. It includes attributes such as User_ID (an integer serving as the primary key, uniquely identifying each user), Name (a string to store the user's full name), Email (a string for the user's email address), Bank_Balance (a decimal value representing the user's current bank balance), and Created_At (a timestamp indicating when the user account was created). This table is fundamental for identifying and authenticating users within the system.

```
CREATE TABLE Users (
    User_ID SERIAL PRIMARY KEY,
    Name VARCHAR(255) NOT NULL,
    Email VARCHAR(255) UNIQUE NOT NULL,
    Bank_Balance DECIMAL(15,2) NOT NULL,
    Created_At TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

Column	Type	Description
User_ID	SERIAL PRIMARY KEY	Auto-incremented unique user ID
Name	VARCHAR(255)	Full name of the user (text input)
Email	VARCHAR(255) UNIQUE	Unique email address (email input)
Bank_Balance	DECIMAL(15,2)	Current bank balance (numeric input)
Created_At	TIMESTAMP	Automatically records creation time

- **Transactions:** This table records financial activities like deposits and withdrawals. Its attributes include Transaction_ID (an integer, the primary key for each transaction record), Amount (a decimal value indicating the transaction amount), Type (a string specifying the nature of the transaction, such as 'deposit' or 'withdrawal'), Description (a string providing a brief description of the transaction), Timestamp (a timestamp recording when the transaction occurred), User_ID (an integer serving as a foreign key referencing the User_ID in the Users table, linking the transaction to the user who initiated it), and Account_ID (an integer serving as a foreign key referencing the Account_ID in the Savings_Account table, linking the transaction to a specific savings account). This table is central to tracking users' spending and saving habits.

```sql
CREATE TABLE Transaction (
    Transaction_ID SERIAL PRIMARY KEY,
    User_ID INT NOT NULL,
    Account_ID INT,
    Amount DECIMAL(15,2) NOT NULL,
    Type VARCHAR(50) CHECK (Type IN ('Deposit', 'Withdrawal')),
    Description TEXT,
    Timestamp TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (User_ID) REFERENCES Users(User_ID) ON DELETE CASCADE,
    FOREIGN KEY (Account_ID) REFERENCES Savings_Account(Account_ID) ON DELETE CASCADE
);
```

Transaction_ID	SERIAL PRIMARY KEY	Unique transaction ID
User_ID	INT	References Users.User_ID
Account_ID	INT	References Savings_Account.Account_ID
Amount	DECIMAL(15,2)	Transaction amount (numeric)
Type	VARCHAR(50)	Either 'Deposit' or 'Withdrawal' (dropdown input)
Description	TEXT	Optional description (text)
Timestamp	TIMESTAMP	Time of transaction (auto)

- **Savings Account:** This table holds details about the separate savings accounts users can create. It contains attributes like Account_ID (an integer, the primary key for each savings account), User_ID (an integer, a foreign key referencing the User_ID in the Users table, indicating which user owns the account), Balance (a decimal value representing the current balance of the savings account), and Created_At (a timestamp indicating when the savings account was created). This separation of savings from the main bank balance allows for focused management of savings goals.

```sql
CREATE TABLE Savings_Account (
    Account_ID SERIAL PRIMARY KEY,
    User_ID INT NOT NULL,
    Balance NUMERIC(15,2) DEFAULT 0 NOT NULL,
    Created_At TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (User_ID) REFERENCES Users(User_ID) ON DELETE CASCADE
);
```

Account_ID	SERIAL PRIMARY KEY	Unique ID for savings account
User_ID	INT	References Users.User_ID
Balance	NUMERIC(15,2)	Savings account balance (numeric)
Created_At	TIMESTAMP	Auto-set account creation timestamp

- **Savings Rule:** This table defines the automated saving mechanisms set up by users. Its attributes include Rule_ID (an integer, the primary key for each rule), Account_ID (an integer, a foreign key referencing the Account_ID in the Savings_Account table, specifying which savings account the rule applies to), Rule_Type (a string indicating the type of saving rule, such as 'Round-Up', 'Fixed', or 'Percentage'), Rule_Condition (a string storing the specific condition for the rule, for example, the round-up value, fixed amount, or percentage), and Active (a boolean value indicating whether the rule is currently active). This table is crucial for the automation aspect of the system, enabling users to customize their saving strategies.

```sql
CREATE TABLE Savings_Rule (
    Rule_ID SERIAL PRIMARY KEY,
    Account_ID INT NOT NULL,
    Rule_Type VARCHAR(50) CHECK (Rule_Type IN ('Round-Up', 'Fixed', 'Percentage')) NOT NULL,
    Rule_Condition TEXT NOT NULL,
    Active BOOLEAN DEFAULT TRUE,
    Created_At TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (Account_ID) REFERENCES Savings_Account(Account_ID) ON DELETE CASCADE
);
```

Rule_ID	SERIAL PRIMARY KEY	Unique rule ID
Account_ID	INT	References Savings_Account.Account_ID
Rule_Type	VARCHAR(50)	Type of rule: 'Round-Up', 'Fixed', or 'Percentage'
Rule_Condition	TEXT	Condition or configuration (input text)
Active	BOOLEAN	Whether the rule is enabled
Created_At	TIMESTAMP	Rule creation timestamp

- **Savings Goal:** This table represents the financial targets set by users. Its attributes include Goal_ID (an integer, the primary key for each goal), User_ID (an integer, a foreign key referencing the User_ID in the Users table, indicating which user created the goal), Goal_Name (a string providing a name for the goal, like "Vacation"), Target_Amount (a decimal value representing the desired amount to be saved), Saved_Amount (a decimal value indicating the current amount saved towards the goal), and Status (a string representing the current status of the goal, such as 'Active' or 'Completed'). This table allows users to define and monitor their progress towards their financial targets.

```
CREATE TABLE Savings_Goal (
    Goal_ID SERIAL PRIMARY KEY,
    User_ID INT NOT NULL,
    Goal_Name VARCHAR(255) NOT NULL,
    Target_Amount NUMERIC(15,2) NOT NULL,
    Saved_Amount NUMERIC(15,2) DEFAULT 0 NOT NULL,
    Status VARCHAR(50) CHECK (Status IN ('Active', 'Completed', 'Cancelled')),
    Created_At TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (User_ID) REFERENCES Users(User_ID) ON DELETE CASCADE
);
```

Goal_ID	SERIAL PRIMARY KEY	Unique goal ID
User_ID	INT	References Users.User_ID
Goal_Name	VARCHAR(255)	Goal name (text input)
Target_Amount	NUMERIC(15,2)	Target savings amount (numeric input)
Saved_Amount	NUMERIC(15,2)	Progress on saving goal
Status	VARCHAR(50)	Goal status: 'Active', 'Completed', or 'Cancelled'
Created_At	TIMESTAMP	Time goal was created

2.4 Functional Dependencies and Third Normal Form

1. Users Table

- **Functional Dependencies:**
  - User_ID → Name, Email, Bank_Balance, Created_At
  - Email → User_ID (since Email is UNIQUE)
- **3NF Check:**
  No transitive dependencies. All non-key attributes are fully dependent on candidate keys.
  Result: In 3NF

2. Transaction Table

- **Functional Dependencies:**
  - Transaction_ID → User_ID, Account_ID, Amount, Type, Description, Timestamp
- **3NF Check:**
  All attributes depend on the primary key. No partial or transitive dependencies.
  Result: In 3NF

3. Savings_Account Table

- **Functional Dependencies:**
  - Account_ID → User_ID, Balance, Created_At

- **3NF Check:**
  All non-key attributes depend directly on the primary key.
  Result: In 3NF

4. **Savings_Rule Table**
   - **Functional Dependencies:**
     - Rule_ID → Account_ID, Rule_Type, Rule_Condition, Active, Created_At
   - **3NF Check:**
     No transitive or partial dependencies.
     Result: In 3NF

5. **Savings_Goal Table**
   - **Functional Dependencies:**
     - Goal_ID → User_ID, Goal_Name, Target_Amount, Saved_Amount, Status, Created_At
   - **3NF Check:**
     All attributes depend only on the primary key. No transitive dependencies.
     Result: In 3NF

2.5 Triggers

2. `trigger_apply_savings_rules_after_transaction`

```sql
-- Trigger to apply rules after a transaction is created
CREATE OR REPLACE FUNCTION apply_savings_rules_after_transaction()
RETURNS TRIGGER AS $$
DECLARE
    rule RECORD;
    saved_amount DECIMAL(15,2);
BEGIN
    -- Check all active savings rules for the user of the transaction
    FOR rule IN
        SELECT Rule_ID, Rule_Type, Rule_Condition
        FROM Savings_Rule
        WHERE Account_ID = NEW.Account_ID AND Active = TRUE
    LOOP
        -- Apply the rule based on the rule type
        IF rule.Rule_Type = 'Round-Up' THEN
            -- Round-Up logic: round the transaction amount to the nearest whole number
            saved_amount := CEIL(NEW.Amount) - NEW.Amount;

        ELSIF rule.Rule_Type = 'Percentage' THEN
            -- Percentage logic: apply the percentage to the transaction amount
            saved_amount := NEW.Amount * (CAST(rule.Rule_Condition AS DECIMAL) / 100);

        ELSIF rule.Rule_Type = 'Fixed' THEN
            -- Fixed savings: apply a fixed amount (as defined in the rule)
            saved_amount := CAST(rule.Rule_Condition AS DECIMAL);
        END IF;

        -- Add the saved amount to the user's Savings Account
        UPDATE Savings_Account
        SET Balance = Balance + saved_amount
        WHERE Account_ID = NEW.Account_ID;
    END LOOP;

    -- Return the NEW transaction data
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

-- Attach the trigger to the Transaction table
CREATE TRIGGER trigger_apply_savings_rules_after_transaction
AFTER INSERT ON Transaction
FOR EACH ROW
EXECUTE FUNCTION apply_savings_rules_after_transaction();
```

- Trigger activates **after an** `INSERT` into the `Transaction` table.
- Loops through all **active savings rules** associated with the account.
- For each rule:
  - `Round-Up`: Saves the difference between the transaction and the next whole number.
  - `Percentage`: Saves a percentage of the transaction amount.
  - `Fixed`: Saves a predefined fixed amount.
- Adds the calculated `saved_amount` to the `Savings_Account` balance.
- Returns the newly inserted transaction row.

2. `trigger_distribute_savings_to_goals`

```
CREATE OR REPLACE FUNCTION distribute_savings_to_goals()
RETURNS TRIGGER AS $$
DECLARE
    active_goal RECORD;
    active_goal_count INT;
    amount_per_goal DECIMAL(15,2);
    added_amount DECIMAL(15,2);
BEGIN
    -- Only proceed if balance increased
    IF NEW.Balance > OLD.Balance THEN
        added_amount := NEW.Balance - OLD.Balance;
        -- Count user's active goals
        SELECT COUNT(*) INTO active_goal_count
        FROM Savings_Goal
        WHERE User_ID = NEW.User_ID AND Status = 'Active';

        -- Only proceed if there are active goals
        IF active_goal_count > 0 THEN
            amount_per_goal := added_amount / active_goal_count;

            -- Loop through all active goals and update Saved_Amount
            FOR active_goal IN
                SELECT Goal_ID
                FROM Savings_Goal
                WHERE User_ID = NEW.User_ID AND Status = 'Active'
            LOOP
                UPDATE Savings_Goal
                SET Saved_Amount = Saved_Amount + amount_per_goal
                WHERE Goal_ID = active_goal.Goal_ID;

                -- Optionally: Immediately mark as completed if target met
                UPDATE Savings_Goal
                SET Status = 'Completed'
                WHERE Goal_ID = active_goal.Goal_ID
                  AND Saved_Amount >= Target_Amount
                  AND Status = 'Active';
            END LOOP;
        END IF;
    END IF;

    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER trigger_distribute_savings_to_goals
AFTER UPDATE ON Savings_Account
FOR EACH ROW
EXECUTE FUNCTION distribute_savings_to_goals();
```

- Trigger activates **after an** `UPDATE` on the `Savings_Account` table.
- Only runs if the account balance has **increased**.
- Calculates the added amount as `NEW.Balance - OLD.Balance`.
- Counts the number of **active savings goals** for the user.
- Divides the added amount **equally among all active goals**.
- Loops through each goal:
  - Adds the calculated portion to `Saved_Amount`.
  - Marks the goal as `Completed` if the target is reached or exceeded.
- Returns the updated `Savings_Account` row.

3. `trigger_update_user_bank_balance_after_transaction`

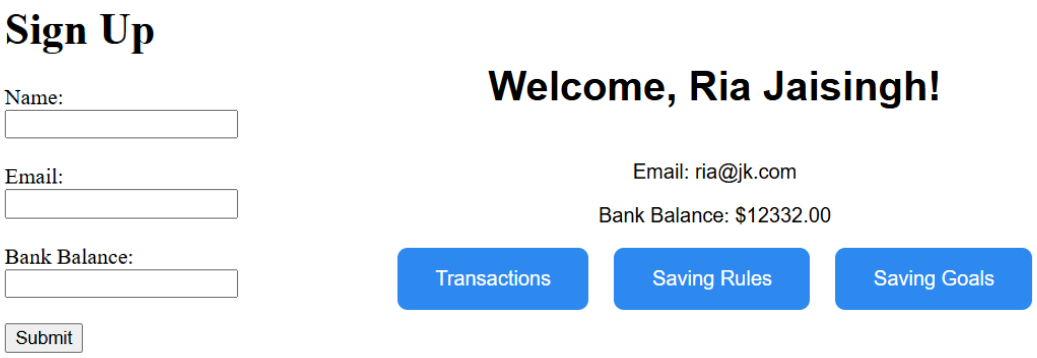
```
----Trigger for Updating User Bank Balance After Transaction
CREATE OR REPLACE FUNCTION update_user_bank_balance_after_transaction()
RETURNS TRIGGER AS $$
BEGIN
    IF NEW.Type = 'Deposit' THEN
        UPDATE Users
        SET Bank_Balance = Bank_Balance + NEW.Amount
        WHERE User_ID = NEW.User_ID;
    ELSIF NEW.Type = 'Withdrawal' THEN
        UPDATE Users
        SET Bank_Balance = Bank_Balance - NEW.Amount
        WHERE User_ID = NEW.User_ID;
    END IF;

    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER trigger_update_user_bank_balance_after_transaction
AFTER INSERT ON Transaction
FOR EACH ROW
EXECUTE FUNCTION update_user_bank_balance_after_transaction();
```

- Trigger activates **after an** `INSERT` on the `Transaction` table.
- Checks the `Type` of the new transaction:
  - If it's a `Deposit`, it **adds** the `Amount` to the user's `Bank_Balance`.
  - If it's a `Withdrawal`, it **subtracts** the `Amount`.
- Identifies the correct user using `NEW.User_ID`.
- Performs the update on the `Users` table.
- Returns the inserted transaction row using `RETURN NEW`.

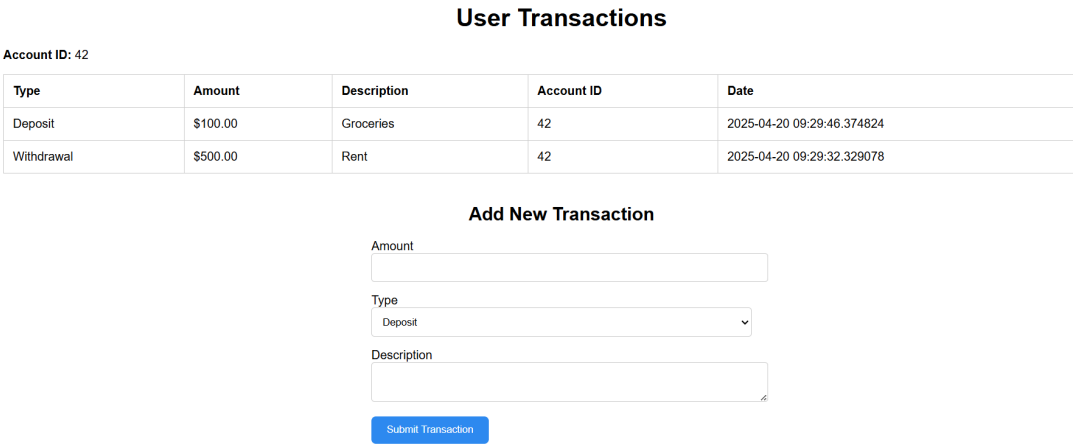
3 Use Case Scenarios:

The application is designed to handle several key use case scenarios which represent the primary ways users interact with the application to manage their finances.

- User Signup & Login: New users can register with the system by providing essential information such as their name, email address, and current bank balance. The email used must be unique to the user and of the correct format. This initial step allows them to create an account and begin using the platform. Once registered, users can log in to access their personalized financial management tools. This process ensures that only authorized individuals can access their financial data within the system.

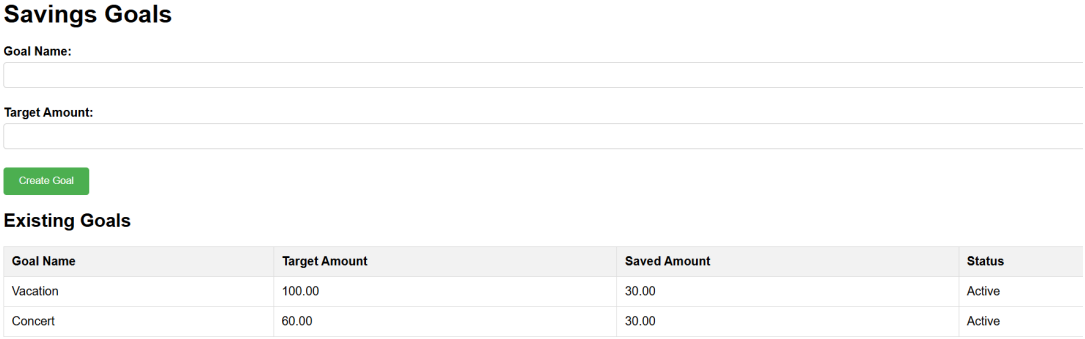


- Track Transactions: After logging in, users can record their daily financial transactions. This includes inputting details like the transaction amount, the type of transaction (e.g., deposit/withdrawal) and a brief description of the transaction. The system then stores this information, providing users with a comprehensive log of their financial activities. This feature enables users to monitor their spending habits and identify areas where they might be able to save more effectively. There is also a check within the system to ensure that a transaction does not exceed the user's bank balance, in this case this will prompt a warning.

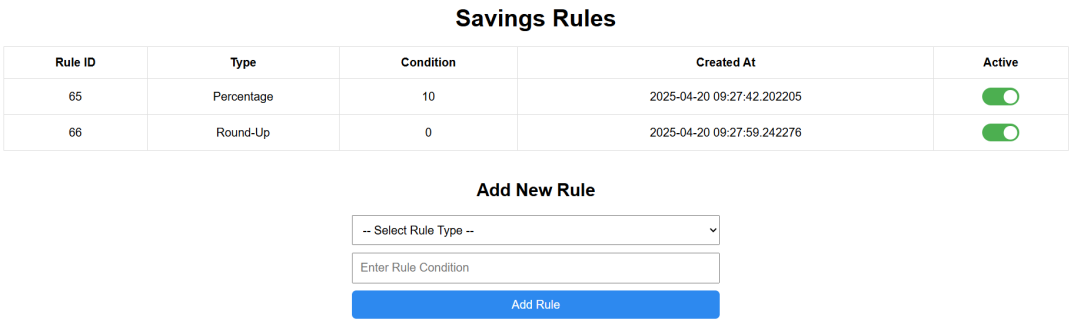


- Set Savings Goals: The application allows users to define specific financial goals they wish to achieve. This involves providing a name for the goal (e.g., "Buying an Apple Phone") and setting a target amount they aim to save. The system then tracks the user's progress towards these goals, providing a clear overview of how much has been saved and how much more is needed to reach the target.

- Track Progress: Users can visualize the progress of their savings goals through an intuitive interface. The system displays the amount saved to date alongside the target amount, allowing users to easily assess how close they are to achieving each goal. This feature provides continuous feedback which helps users stay engaged with their savings goals and provides a sense of accomplishment as they get closer to reaching their targets.



- Create Savings Rules: To facilitate automated savings, users can set up various rules that trigger automatic transfers to their savings accounts. These rules can be based on different criteria:
  - Round-Up: This rule rounds up each transaction to the nearest dollar, and the difference is automatically transferred to the user's savings account. For example, if a user spends $3.75 on coffee, the transaction will be rounded up to $4.00, and the $0.25 difference would be saved.
  - Fixed Amount: Users can set a recurring fixed amount to be automatically transferred to their savings account (e.g., $10 on every transaction). This ensures consistent saving without requiring manual intervention.
  - Percentage-Based: Users can choose to save a certain percentage of each transaction (e.g., save 5% of grocery purchase). This allows savings to grow proportionally with spending.



- Analytics: The application includes an internal analytics section that utilizes the collected transaction data to extract insights into user financial behavior. Although not visible to end-users, this feature allows the company to analyze spending patterns by category (e.g., groceries, entertainment, utilities), monitor savings growth trends, and potentially predict future financial behaviors based on historical data. These insights can guide future development, improve user experience, and support data-driven decision-making for feature enhancements.

4 System Architecture Overview

This application is built using a three-tier architecture, integrating a Flask backend, HTML frontend, and a PostgreSQL database.

Backend (Flask)

Flask serves as the core of the application. It handles:

- HTTP requests and routing (e.g., `/signup`, `/login`, `/transactions`)
- Business logic, such as validating withdrawals against user balances

- Communication with the PostgreSQL database using SQL queries
- Passing data to the frontend templates using Jinja2

Frontend (HTML + Jinja2)

The frontend consists of HTML templates rendered by Flask. These pages:

- Provide user interfaces for signing up, logging in, adding transactions, and more
- Display data dynamically using Jinja2 templating syntax (e.g., {{ `user.name` }})
- Handle form submissions that send data to the backend via POST requests

Database (PostgreSQL)

PostgreSQL is used to persist all data, including:

- User information (`Users` table)
- Savings account details (`Savings_Account`)
- Transactions (`Transaction`)
- Savings rules and goals

Flask connects to PostgreSQL using `psycopg2`, executing SQL queries to insert, retrieve, and update data.

Workflow Example: Adding a Transaction

1. A user fills out a form on the `/transactions` page.
2. The form data is sent to the Flask backend.
3. Flask validates the transaction (e.g., ensures sufficient balance for withdrawals).
4. If valid, the transaction is inserted into the `Transaction` table, and the user's bank balance is updated.
5. The updated list of transactions is fetched from the database and rendered back in the HTML page.

5 Application Features

5.1 Functional Features:

- User Signup & Login: The signup process requires users to provide their name, email, and initial bank balance. This information is stored securely in the database. The login functionality allows registered users to access their accounts using their name.
- Track Transactions: Users can input details of their financial transactions through a user-friendly interface. This typically involves fields for the amount, type (deposit/withdrawal) and description. The entered data is then stored in the Transaction table in the database, linked to the user's account.
- Set Savings Goals: The application provides an interface for users to create new savings goals. This would involve specifying a goal name and a target amount. This information is stored in the Savings_Goal table, associated with the user's User_ID.
- Set Savings Rule: Users can configure automated savings rules through a dedicated section of the application. For the "Round-Up" rule, users might specify if they want transactions rounded to the nearest dollar. For the "Fixed Amount" rule, they would define the amount of transfer. For the "Percentage-Based" rule, they would specify the percentage to be saved. These rules are stored in the Savings_Rule table, linked to a specific Account_ID.

- Track Progress: The system displays the progress towards each savings goal. This shows the Saved_Amount compared to the Target_Amount from the Savings_Goal table.
- Analytics: The application features a dashboard that provides key analytics such as the total number of users, the top 5 savers by savings account balance, and inactive accounts (those with no transactions in the past 3 months). It also visualizes transaction activity over the last 30 days using graphs that highlight daily deposits and withdrawals, offering insights into user behavior and system engagement.

5.2 Non-Functional Features:

1. **Usability:** The application features a clean, intuitive user interface with simple forms and navigation, ensuring ease of use for both new and returning users.
2. **Performance:** Efficient use of SQL queries and a lightweight Flask backend enables fast page loads and responsive user interactions, even with multiple users accessing the system.
3. **Security:** Input validation and parameterized SQL queries help prevent SQL injection and unauthorized access. Unique constraints and relational integrity further enhance data protection.
4. **Reliability:** The application includes checks such as preventing overdrafts, ensuring consistent functionality, and minimizing system errors during financial operations.
5. **Maintainability:** The modular code structure, separation of logic (Flask), presentation (HTML), and data (PostgreSQL), makes the application easy to maintain and extend in the future.

6 Development Challenges:

- Implementing automated behavior using PostgreSQL triggers—especially for distributing savings among multiple goals and updating related tables—required careful planning to avoid circular dependencies and ensure data consistency. Debugging triggers was also more challenging compared to regular application logic.
- Designing a normalized database schema that avoided redundancy while still allowing for efficient queries and updates posed challenges. Ensuring referential integrity through primary and foreign keys across multiple related tables required rigorous validation.
- Making sure the application frontend reflects changes in real-time (e.g., balance updates, goal progress) while avoiding race conditions or stale data was a non-trivial task. Using transactions and carefully structuring the update flow helped mitigate issues.
- Building a clean and intuitive interface in Flask using HTML templates while integrating multiple views (sign-up, login, dashboard, transactions, rules, goals) needed iterative design decisions to ensure a smooth user experience.

7 Database Concepts Used in the Application

This application relies heavily on PostgreSQL and SQL for its data management. Below is a detailed explanation of the key database concepts implemented in the system:

1. Relational Database Model

The system is built using the relational database model where data is organized into structured tables. Relationships between entities such as users, accounts, transactions, and rules are managed using foreign keys.

2. Schema Design

A well-defined schema includes entities like `Users`, `Transaction`, `Savings_Account`, `Savings_Rule`, and `Savings_Goal`. Each table specifies appropriate data types, constraints, and keys to maintain data integrity and reduce redundancy.

3. Primary Keys

Each table has a primary key that uniquely identifies its rows:

- `User_ID` for the `Users` table
- `Account_ID` for the `Savings_Account` table
- `Transaction_ID` for the `Transaction` table

4. Foreign Keys

Foreign keys establish relationships between tables:

- `Transaction.User_ID` references `Users.User_ID`
- `Savings_Account.User_ID` references `Users.User_ID`
- `Transaction.Account_ID` references `Savings_Account.Account_ID`

5. Normalization (Up to 3NF)

The database design follows normalization principles up to Third Normal Form (3NF):

- **1NF:** All fields contain only atomic values.
- **2NF:** All non-key attributes are fully functionally dependent on the primary key.
- **3NF:** No transitive dependencies exist.

6. Functional Dependencies

Key functional dependencies in the schema include:

- `User_ID` $\rightarrow$ `Name, Email, Bank_Balance`
- `Account_ID` $\rightarrow$ `Balance, User_ID`
- `Transaction_ID` $\rightarrow$ `Amount, Type, Timestamp, User_ID, Account_ID`

7. Data Types

The application uses appropriate data types for each field:

- `INT` for IDs
- `VARCHAR` for names, emails, and descriptions
- `DECIMAL` for monetary values like balances and amounts
- `TIMESTAMP` for recording transaction time
- `BOOLEAN` for status flags

8. Triggers and Procedural Logic

Several `PL/pgSQL` triggers automate logic:

- Applying savings rules after a transaction
- Distributing savings into goals
- Updating bank balance after transactions

9. Stored Procedures

Stored procedures encapsulate logic and are invoked by triggers to:

- Evaluate savings conditions
- Modify account and goal balances

10. Transaction Management

Transaction integrity is maintained using explicit `commit()` calls in the application to ensure atomicity and data consistency.

11. Data Integrity Constraints

Constraints like `NOT NULL`, `UNIQUE`, `CHECK`, and key constraints ensure that only valid data is stored.

12. Indexing

To improve performance, indexing frequently used columns such as `User_ID` and `Account_ID`.

13. CRUD Operations

The application implements:

- **Create:** New users, accounts, transactions, rules, and goals
- **Read:** Dashboards, history, top savers, inactive accounts
- **Update:** Balances, rule states, goal progress
- **Delete:** (Can be extended in future)

14. Integration with Flask

The Flask backend interacts with the database using safe, parameterized SQL queries and encapsulates the logic for:

- User registration and login
- Balance and goal tracking
- Transaction processing
- Dashboard reporting

8 Set up instructions for the website

This guide will help you set up and run the Automated Savings App using **Flask** and **PostgreSQL**.

1. Clone or Download the Project

Download the project folder `App.env` containing the following files:

- `app.py` (Flask application code)
- `connection.py` (PostgreSQL connection code)
- `info.env` (environment variables — you'll create this)
- `projectsqldump.backup` (SQL dump for PostgreSQL)
- `templates/` (folder for HTML templates)
- `static/` (folder for CSS, JavaScript, and images)

2. Restore the PostgreSQL Database in pgAdmin

2.1 Create a New Database

- Open **pgAdmin** and log in.
- In the **pgAdmin** dashboard, right-click on **Databases** → **Create** → **Database**.
- Name the new database: `Automated_savings_system`.
- Click **Save** to create the database.

2.2 Restore the `.backup` File

- Right-click on the newly created database (`Automated_savings_system`) → **Restore**.
- In the **Restore** dialog:
  - For **Format**, select `Custom or Tar`.
  - For **Filename**, click the **folder icon** and locate the `.backup` file.
- Ensure that the **Restore Options** include:
  - **Restore schema**: Checked
  - **Restore data**: Checked
- Click **Restore**. After it completes, you should see tables like `users`, `transactions`, etc., in the database.

3. Create an `info.env` File

In the project folder `App`, create a file named `info.env` and paste the following (replace the placeholder with your actual PostgreSQL password):

```
DB_NAME=Automated_savings_system
DB_USER=postgres
DB_PASSWORD=YourPasswordHere
DB_HOST=localhost
DB_PORT=5432
```

Make sure to use the correct password for your local PostgreSQL server in `DB_PASSWORD`.

4. Run the Flask App

4.1 Start the Flask App

In the terminal (with your virtual environment activated), run:

```
cd [your project directory]
```

Once you are in the correct directory within the activated environment, execute the following command:

```
python app.py
```

4.2 Visit the Flask App in Your Browser

Open a browser and visit:

- **Homepage**: `http://127.0.0.1:5000/` – Should show a welcome message.
- **Users Page**: `http://127.0.0.1:5000/users` – Should show data from the `users` table in your database.

Troubleshooting

- **PostgreSQL Connection**: If there are issues connecting to the database, check if the **username**, **password**, and **database name** in the `info.env` file are correct.
- **pgAdmin**: Make sure you've successfully restored the dump using the **Query Tool** in pgAdmin, and the tables appear.
- **Flask App Not Running**: Ensure Flask is installed and there are no errors when starting the app with `python app.py`.