Payback:

Ansh Aggarwal, Dylan Mikulka, Shahid Shaikh, Malia Wanderer

Northeastern University, Boston, MA, USA

Abstract

In group settings, the difficulty of managing shared expenses and settling debts can often lead to hesitation overpaying upfront, or interpersonal tension. Tracking who paid for what, calculating who owes whom, and monitoring which debts are settled vs outstanding can become unnecessarily complicated. Payback aims to simplify the entire process, providing a clean, structured database approach to handling shared expenses across groups and optimizing settlement flows.

In this project, we will design a relational database to track users, groups, expenses, debts, and settlements. By establishing well-defined entities and their relationships, we will create a foundation that supports group creation and membership, expense logging, and payback calculations.

Introduction

The goal of this project is to build a database that can model shared expenses across groups. The database must handle:

- Tracking expenses logged by users
- Splitting costs among group members
- Calculating debts and credits
- Recording settlements
- And optimizing payments to minimize the number of required transactions

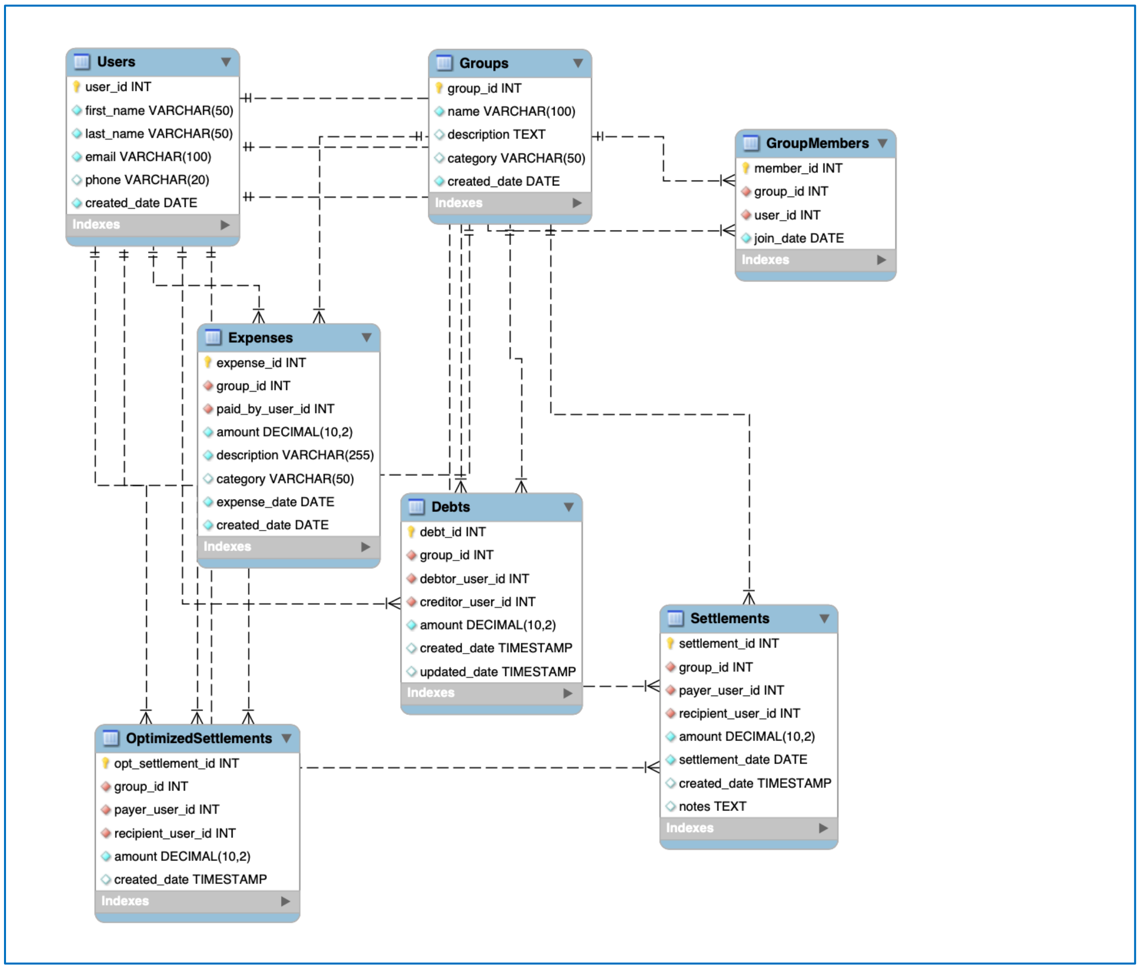
This application can be useful in many real-world scenarios, such as travel, dining, shared housing, and many other group activities. By tracking the flow of money in a structured way, Payback creates a hassle-free experience for financial coordination.

Data Sources

Since Payback is an application prototype, we generated mock data to simulate realistic expense-sharing scenarios. We created 12 fictional users, including our team members and 8 additional users. We then designed 8 diverse groups representing common use cases including a spring break trip, apartment expenses, and a road trip. From those groups, we generated 49 total expenses with realistic amounts and assigned users to multiple groups to simulate real-world overlap.

This data we created was then stored into 4 CSV files (users.csv, groups.csv, group_members.csv, and expenses.csv) and imported into MySQL using the Table Data Import Wizard. The Debts table was subsequently populated through SQL queries that calculated splits based on group membership.

Database Design



Our database consists of seven interconnected tables designed to track users, groups, expenses, and who owes who.

- The Users table stores basic account info like names, emails, and phone numbers.
- The Groups table holds information about each expense-sharing group, including its name, category (e.g., housing, travel) and the creation date.
- The Group Members table links Users to Groups in their many-to-many relationship. This allows for our assumption that a group can have multiple users and a user can be in multiple groups.

- The Expenses table stores all transactions. Each expense links to a specific group and is associated with a specific user who paid for it. This creates specific relationships that allow one user to pay multiple expenses and for one group to have tons of expenses over time.
- The Debts table calculates who owes money to whom. This table connects to Users twice, once for the debtor and once for the creditor.
- The Settlements table records payments made to settle outstanding debts. Similarly to Debts, it links to Users for both the debtor and the creditor.
- The Optimized Settlements table stores results from the optimization algorithm that figures out the minimum number of payments needed to settle up a group.

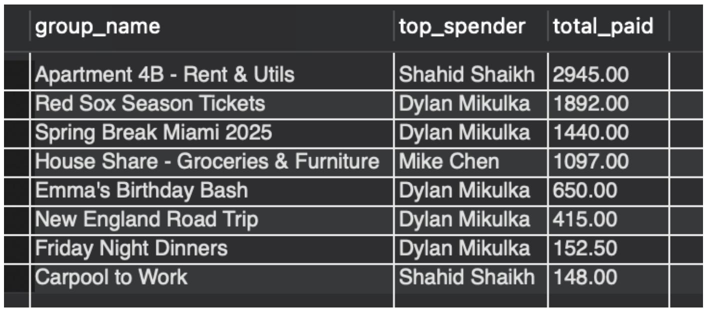
This relational structure allows for group membership, hassle-free expense logging, and precise debt tracking.

User Cases (Application Prototype) or Analysis (Data Science)

1. Who paid the most in the group? This query uses and joins the tables Expenses, Groups, and Users to output the user who has covered the most expenses in each group.

```
SELECT
    g.name AS group_name,
    CONCAT(u.first_name, ' ', u.last_name) AS top_spender,
    SUM(e.amount) AS total_paid
FROM Expenses e
JOIN `Groups` g ON e.group_id = g.group_id
JOIN Users u ON e.paid_by_user_id = u.user_id
GROUP BY g.group_id, g.name, u.user_id, u.first_name, u.last_name
HAVING SUM(e.amount) = (
    SELECT MAX(group_total)
    FROM (
        SELECT e2.group_id, e2.paid_by_user_id, SUM(e2.amount) AS group_total
        FROM Expenses e2
        WHERE e2.group_id = g.group_id
        GROUP BY e2.group_id, e2.paid_by_user_id
    ) AS subquery
)
ORDER BY total_paid DESC;
```

Output Example for Top Spender:

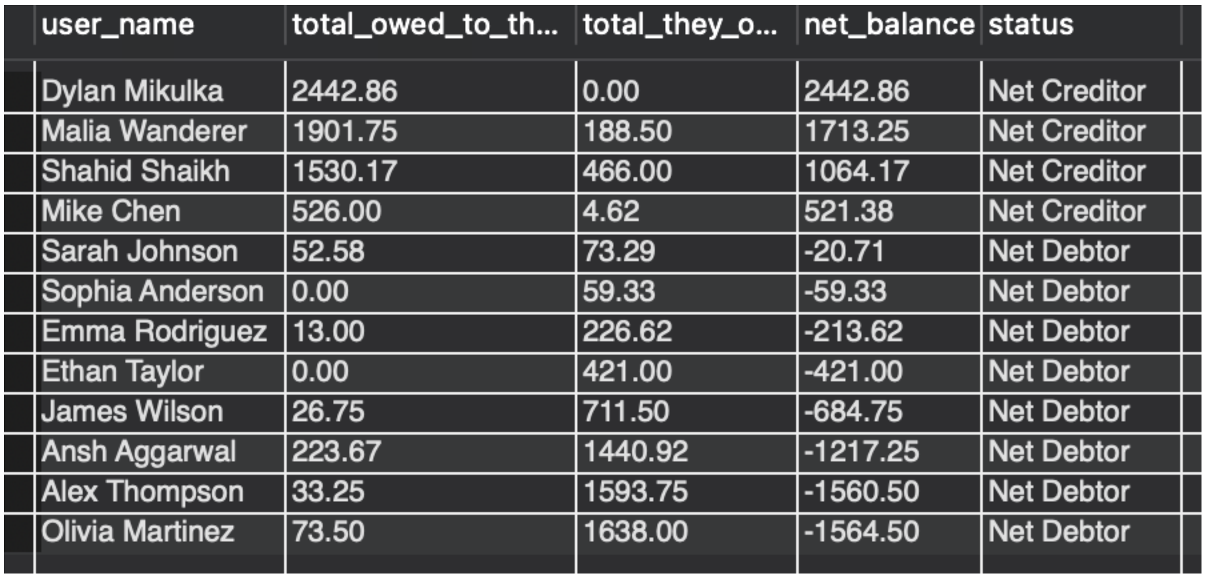


group_name	top_spender	total_paid
Apartment 4B - Rent & Utils	Shahid Shaikh	2945.00
Red Sox Season Tickets	Dylan Mikulka	1892.00
Spring Break Miami 2025	Dylan Mikulka	1440.00
House Share - Groceries & Furniture	Mike Chen	1097.00
Emma's Birthday Bash	Dylan Mikulka	650.00
New England Road Trip	Dylan Mikulka	415.00
Friday Night Dinners	Dylan Mikulka	152.50
Carpool to Work	Shahid Shaikh	148.00

2. What is the net balance of each user?
   This query uses two separate subqueries to calculate the credits and debts of each user by using left join to find the amount owed to and owed by each user in debts.

```
SELECT
    CONCAT(u.first_name, ' ', u.last_name) AS user_name,
    COALESCE(owed_to.total, 0) AS total_owed_to_them,
    COALESCE(they_owe.total, 0) AS total_they_owe,
    COALESCE(owed_to.total, 0) - COALESCE(they_owe.total, 0) AS net_balance,
    CASE
        WHEN COALESCE(owed_to.total, 0) - COALESCE(they_owe.total, 0) > 0
        THEN 'Net Creditor'
        ELSE 'Net Debtor'
    END AS status
FROM Users u
LEFT JOIN (
    SELECT creditor_user_id, SUM(amount) AS total
    FROM Debts
    GROUP BY creditor_user_id
) owed_to ON u.user_id = owed_to.creditor_user_id
LEFT JOIN (
    SELECT debtor_user_id, SUM(amount) AS total
    FROM Debts
    GROUP BY debtor_user_id
) they_owe ON u.user_id = they_owe.debtor_user_id
WHERE COALESCE(owed_to.total, 0) > 0 OR COALESCE(they_owe.total, 0) > 0
ORDER BY net_balance DESC;
```

Output Example for Net Balance:

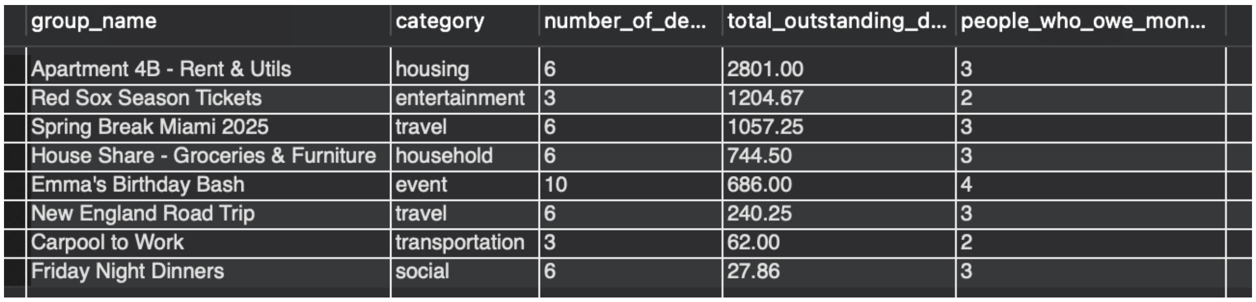


user_name	total_owed_to_th...	total_they_o...	net_balance	status
Dylan Mikulka	2442.86	0.00	2442.86	Net Creditor
Malia Wanderer	1901.75	188.50	1713.25	Net Creditor
Shahid Shaikh	1530.17	466.00	1064.17	Net Creditor
Mike Chen	526.00	4.62	521.38	Net Creditor
Sarah Johnson	52.58	73.29	-20.71	Net Debtor
Sophia Anderson	0.00	59.33	-59.33	Net Debtor
Emma Rodriguez	13.00	226.62	-213.62	Net Debtor
Ethan Taylor	0.00	421.00	-421.00	Net Debtor
James Wilson	26.75	711.50	-684.75	Net Debtor
Ansh Aggarwal	223.67	1440.92	-1217.25	Net Debtor
Alex Thompson	33.25	1593.75	-1560.50	Net Debtor
Olivia Martinez	73.50	1638.00	-1564.50	Net Debtor

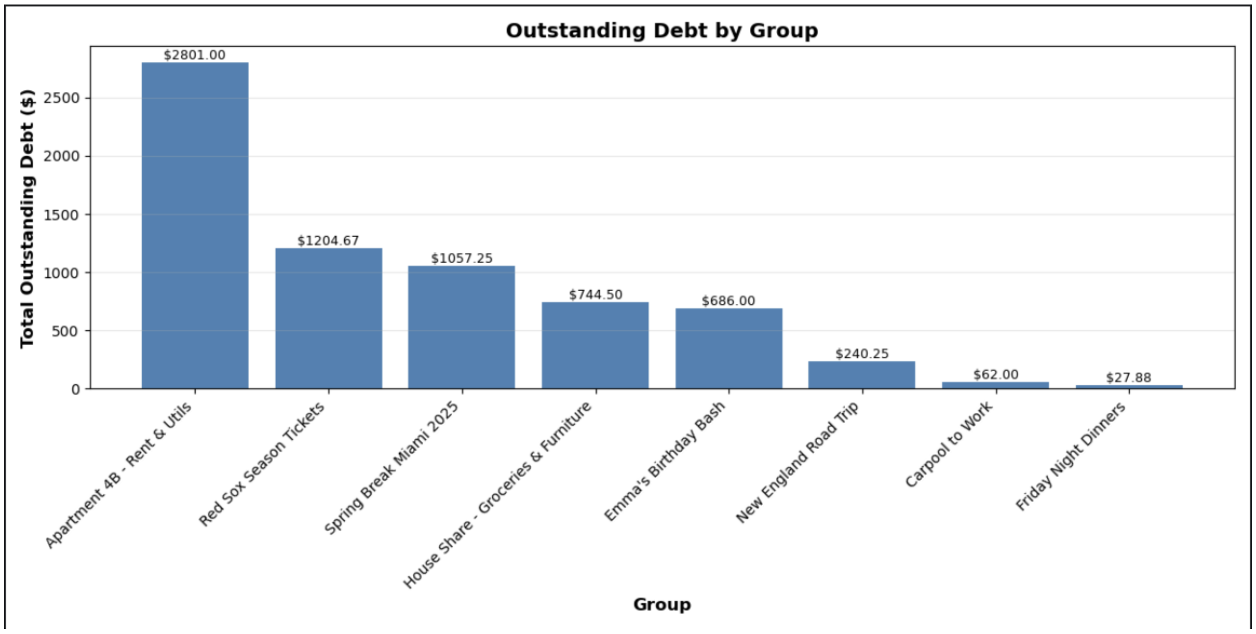
3. Which groups have the most outstanding debt? This query joins debts and groups and counts the number of debts. Using this total number of debts, the results are ordered in descending ord

```sql
SELECT
    g.name AS group_name,
    g.category,
    COUNT(d.debt_id) AS number_of_debts,
    SUM(d.amount) AS total_outstanding_debt,
    COUNT(DISTINCT d.debtor_user_id) AS people_who_owe_money
FROM `Groups` g
JOIN Debts d ON g.group_id = d.group_id
GROUP BY g.group_id, g.name, g.category
ORDER BY total_outstanding_debt DESC;
```

Output Example for Outstanding Debt:



group_name	category	number_of_de...	total_outstanding_d...	people_who_owe_mon...
Apartment 4B - Rent & Utils	housing	6	2801.00	3
Red Sox Season Tickets	entertainment	3	1204.67	2
Spring Break Miami 2025	travel	6	1057.25	3
House Share - Groceries & Furniture	household	6	744.50	3
Emma's Birthday Bash	event	10	686.00	4
New England Road Trip	travel	6	240.25	3
Carpool to Work	transportation	3	62.00	2
Friday Night Dinners	social	6	27.86	3



4. What is the settlement progress per group? This query compares the number of settlements made with the number of outstanding debts

```
SELECT
    g.name AS group_name,
    COALESCE(debt_totals.total_debt, 0) AS total_debt,
    COALESCE(settlement_totals.total_settled, 0) AS total_settled,
    ROUND(
        COALESCE(settlement_totals.total_settled, 0) /
        NULLIF(COALESCE(debt_totals.total_debt, 0), 0) * 100,
        1
    ) AS percent_settled
FROM `Groups` g
LEFT JOIN (
    SELECT group_id, SUM(amount) AS total_debt
    FROM Debts
    GROUP BY group_id
) debt_totals ON g.group_id = debt_totals.group_id
LEFT JOIN (
    SELECT group_id, SUM(amount) AS total_settled
    FROM Settlements
    GROUP BY group_id
) settlement_totals ON g.group_id = settlement_totals.group_id
ORDER BY percent_settled DESC;
```

5. Roommate debt breakdown—in a housing group who owes who? This query uses the specific example of a housing group to return a tabular representation of settlements needing to be made between members.

```
SELECT
    CONCAT(debtor.first_name, ' ', debtor.last_name) AS roommate_who_owes,
    CONCAT(creditor.first_name, ' ', creditor.last_name) AS roommate_owed_money,
    d.amount AS amount_owed
FROM Debts d
JOIN Users debtor ON d.debtor_user_id = debtor.user_id
JOIN Users creditor ON d.creditor_user_id = creditor.user_id
JOIN `Groups` g ON d.group_id = g.group_id
WHERE g.category = 'housing'
ORDER BY d.amount DESC;
```

6. Where is the most money being spent? This query looks at the category of each expense to return a breakdown of the total spent in each category as well as averages and smallest and largest expenses.

```sql
SELECT
    category,
    COUNT(*) AS number_of_transactions,
    SUM(amount) AS total_spent,
    ROUND(AVG(amount), 2) AS average_expense,
    MIN(amount) AS smallest_expense,
    MAX(amount) AS largest_expense
FROM Expenses
GROUP BY category
ORDER BY total_spent DESC;
```

7. Which users are in multiple groups together? This query self joins on group members to find user pairs belonging to more than one group together.

```sql
SELECT
    CONCAT(u1.first_name, ' ', u1.last_name) AS user_1,
    CONCAT(u2.first_name, ' ', u2.last_name) AS user_2,
    COUNT(DISTINCT gm1.group_id) AS shared_groups,
    GROUP_CONCAT(DISTINCT g.name SEPARATOR ', ') AS group_names
FROM GroupMembers gm1
JOIN GroupMembers gm2 ON gm1.group_id = gm2.group_id
    AND gm1.user_id < gm2.user_id
JOIN Users u1 ON gm1.user_id = u1.user_id
JOIN Users u2 ON gm2.user_id = u2.user_id
JOIN `Groups` g ON gm1.group_id = g.group_id
GROUP BY u1.user_id, u2.user_id, u1.first_name, u1.last_name,
         u2.first_name, u2.last_name
HAVING COUNT(DISTINCT gm1.group_id) > 1
ORDER BY shared_groups DESC;
```

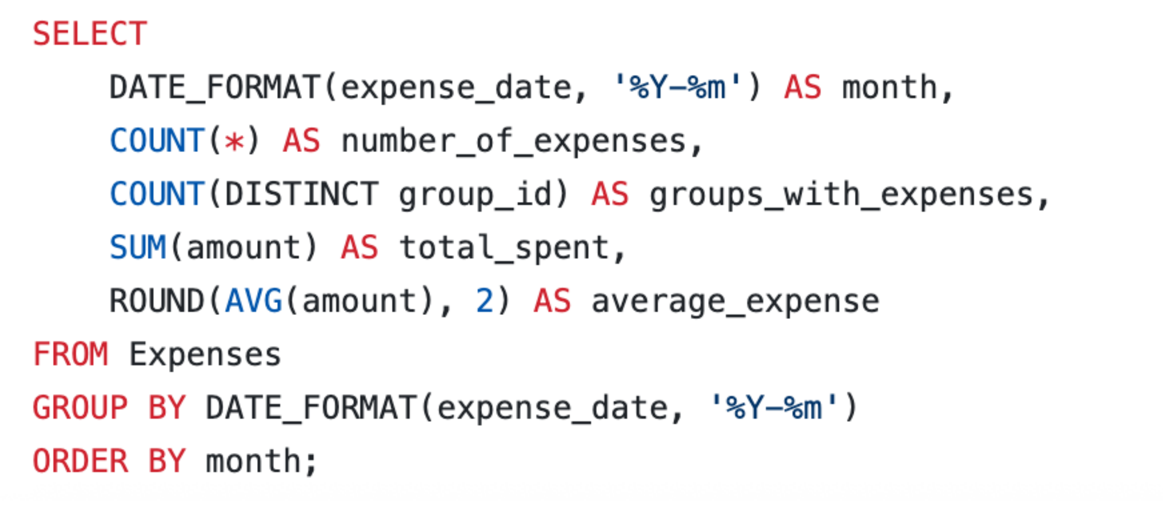
8. What does Ansh owe? This query presents an example of a personal dashboard by filtering by user id. This is the amount that Ansh would see when he logs into the app.

```sql
SELECT
    g.name AS group_name,
    CONCAT(creditor.first_name, ' ', creditor.last_name) AS you_owe,
    d.amount AS amount
FROM Debts d
JOIN `Groups` g ON d.group_id = g.group_id
JOIN Users creditor ON d.creditor_user_id = creditor.user_id
WHERE d.debtor_user_id = 4  -- Ansh
ORDER BY d.amount DESC;
```

9. How many transactions can we save using our settlement optimization algorithm? This query is to show the value of our settlement optimization algorithm and how Payback can reduce personal calculations of sending transactions back and forth.

```sql
SELECT
    g.name AS group_name,
    (SELECT COUNT(*) FROM Debts WHERE group_id = g.group_id)
        AS current_transactions_needed,
    (SELECT COUNT(DISTINCT user_id)
     FROM (
         SELECT debtor_user_id AS user_id FROM Debts WHERE group_id = g.group_id
         UNION
         SELECT creditor_user_id AS user_id FROM Debts WHERE group_id = g.group_id
     ) AS all_users) - 1
        AS optimized_transactions_needed,
    (SELECT COUNT(*) FROM Debts WHERE group_id = g.group_id) -
    (SELECT COUNT(DISTINCT user_id)
     FROM (
         SELECT debtor_user_id AS user_id FROM Debts WHERE group_id = g.group_id
         UNION
         SELECT creditor_user_id AS user_id FROM Debts WHERE group_id = g.group_id
     ) AS all_users2) + 1
        AS transactions_saved
FROM `Groups` g
WHERE EXISTS (SELECT 1 FROM Debts WHERE group_id = g.group_id)
ORDER BY transactions_saved DESC;
```

10. Monthly spending trends, when do groups spend the most? This query groups expenses by months to show patterns of spending over time.



```
SELECT
    DATE_FORMAT(expense_date, '%Y-%m') AS month,
    COUNT(*) AS number_of_expenses,
    COUNT(DISTINCT group_id) AS groups_with_expenses,
    SUM(amount) AS total_spent,
    ROUND(AVG(amount), 2) AS average_expense
FROM Expenses
GROUP BY DATE_FORMAT(expense_date, '%Y-%m')
ORDER BY month;
```

Implementation Challenges

Building the Payback database came with some tricky design problems we had to work through. The biggest challenge was figuring out how to split expenses. We ended up going with equal splits to keep things simple, but we knew pretty early on that real life doesn't always work that way. For example, going out to dinner with your friends, one person might order an expensive meal and drink, while the others are budgeting. We decided to stick with equal splits for now to get a working system going, knowing we could add more complex splitting later if we needed to.

Another thing that took some thinking was how to connect Debts and Settlements. At first, we thought about just updating the Debts table directly when someone paid, like if you owed $50 and paid back $20, we'd just change it to $30. But we quickly realized that we'd lose all the history of who paid what and when. So instead, we kept Debts as the running total and made Settlements a separate record of payments.

We also had to be careful about data validation. We needed to make sure people couldn't accidentally get added to the same group twice, that expense amounts made sense, and that the debt calculations matched up with who was actually in the group when each expense happened. We used UNIQUE constraints to stop duplicate memberships and CHECK constraints to make sure numbers were valid.

Conclusions

Our project brought together the full design and analysis of the Payback database. We created tables for users, groups, group members, expenses, debts, and settlements, and showed how each one works together to model real shared expenses. We used our generated data to test the system and to answer the research questions from the slides and the report. These included finding who paid the most, calculating net balances, identifying outstanding debts, checking settlement progress, and reviewing group spending patterns.

The data sources we created allowed us to run meaningful SQL queries and confirm that the database supports all key tasks for an expense sharing app. The structure also allowed us to analyze different types of groups such as trips, food outings, and housing costs. This showed that the model is flexible and can be used across many situations.

There are limits. The data is synthetic, and the system currently assumes equal splitting of expenses. The settlement logic works but can be improved in the future. Even with these limits, the project successfully delivered a complete database foundation that supports accurate tracking, clear reporting, and future expansion of the Payback application.

Author Contributions

Dylan Mikulka: Generated mock data using Python, imported CSV files into MySQL, wrote and tested SQL queries, contributed to the report, and set up the GitHub repository.

Malia Wanderer: Led database schema design, created the ER diagram in MySQL Workbench, helped write the final report, and contributed to query development.

Ansh Aggarwal: Wrote and tested SQL queries, developed the settlement optimization of query logic, helped create the presentation slides, and documented query explanations.

Shahid Shaikh: Wrote and tested SQL queries, helped create the slide deck, contributed to the final report, and contributed to the database schema design.