MoMo SMS Data Processing System

Database Design & Implementation Document

Week 2: Database Design Assignment

Team Name: Webcores

Team Members:

- Nirere Aliya
- Maellen MPINGANZIMA
- Benjamin NIYOMURINZI
- Noella UWERA

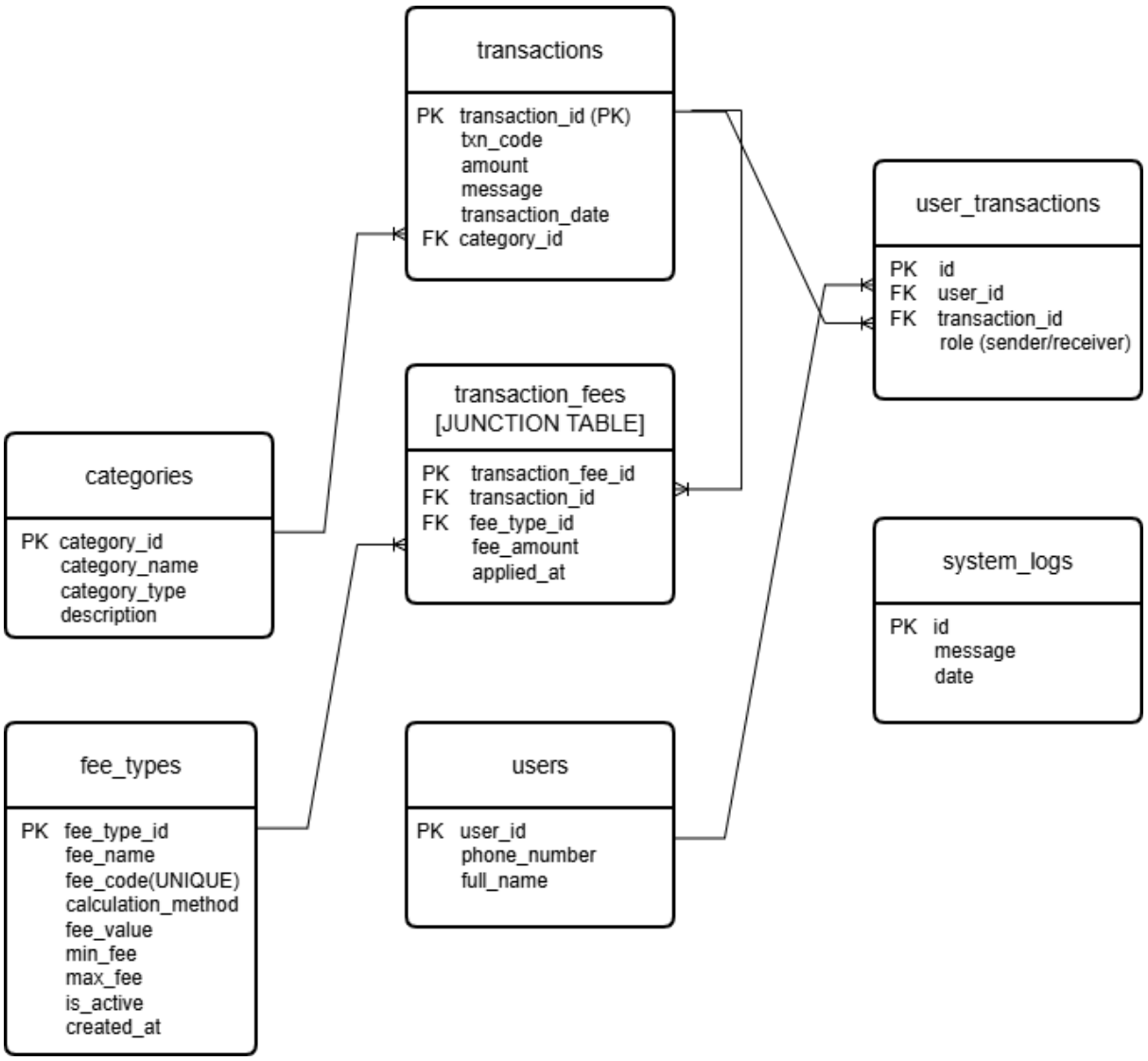
Course: Web infrastructure

Date: January 24, 2026

TABLE OF CONTENTS

1. ENTITY RELATIONSHIP DIAGRAM (ERD)

The following diagram represents the complete database schema for the MoMo SMS Data Processing System. It includes 7 tables with their relationships and cardinalities clearly marked.



Key Features:

- 7 entities (tables) representing different aspects of MoMo transactions
- 1 Many-to-Many relationship resolved with a junction table
- 6 One-to-Many relationships
- Primary Keys (PK) and Foreign Keys (FK) clearly marked
- Cardinality notation (1:M, M:N) on all relationships

2. DESIGN RATIONALE AND JUSTIFICATION

Our MoMo SMS data processing database is designed to efficiently handle mobile money transaction data while maintaining data integrity, supporting scalability, and enabling comprehensive analytics. The following rationale explains our key design decisions.

Entity Separation and Normalization

We separated users from transactions to eliminate data redundancy. A single user can participate in multiple transactions (as sender or receiver), so storing the user information once, and referencing it by ID prevents duplicate data and ensures consistency. This normalization approach reduces storage requirements and simplifies data updates.

Transaction Categorization Strategy
The categories table provides flexible transaction classification without modifying the main transaction records. This design supports easy addition of new transaction types (such as loan payments or subscription services) without database restructuring. Categories are defined once and reused across thousands of transactions, following the DRY (Don't Repeat Yourself) principle.

Many-to-Many Relationship Resolution
We identified a critical M:N relationship between transactions and fee types. A single transaction can incur multiple fees (transaction fee + tax + service charge), and a single fee type applies to many different transactions. We resolved this using the transaction_fees junction table, which stores the actual fee amount charged for each transaction-fee combination. This design provides maximum flexibility for complex fee structures while maintaining data integrity.

User Transaction Participation Model
The user_transactions table maps users to transactions with their specific roles (sender or receiver). This design allows one transaction to involve multiple participants and tracks each user's role explicitly. It also supports future expansion to multi-party transactions or group payments.

Performance Optimization
We added strategic indexes on frequently queried columns (phone_number, transaction_date, category_id) to optimize search and reporting performance. Composite indexes on (transaction_date, status) support common dashboard queries that filter by both date range and transaction status.

Data Integrity and Security
Foreign key constraints ensure referential integrity, preventing orphaned records. CHECK constraints enforce business rules such as positive amounts and valid balance calculations. UNIQUE constraints on phone_number prevent duplicate user accounts. These constraints act as database-level validation, ensuring data quality even if application-level validation fails.

Scalability Considerations
The design supports future growth through nullable optional fields (email, balance_before), timestamp tracking on all tables, and the standalone system_logs table for monitoring and debugging. The fee_types table allows new fee structures to be added without schema changes.

Audit and Monitoring
The system_logs table provides comprehensive ETL process tracking with structured error handling. This supports debugging, performance monitoring, and compliance auditing without impacting transactional tables.

3. DATA DICTIONARY

3.1 TABLE: users

Purpose: Stores customer and user account information for all MoMo system users.

Columns:

Column Name	Data Type	Constraints	Description
user_id	INT	PK, AUTO_INCREMENT	Unique identifier for user
phone_number	varchar(15)	UNIQUE, NOT NULL	User phone number in format (+250XXXXXXXXX)
full_name	varchar	NOT NULL	Full name of the user

Relationships: One user → Many user_transactions (1:M via user_id)

3.2 TABLE: categories

Purpose: Defines transaction categories for classification and analysis.

Columns:

Column Name	Data Type	Constraints	Description
category_id	init	PK, AUTO_INCREAMENT	Unique identifier for category
category_name	VARCHAR(50)	UNIQUE, NOT NULL	Name of the transaction category
category_type	ENUM	NOT NULL	Type: Income or Expense
description	TEXT	NULL	Detailed category description

Relationships: One category → Many transactions (1:M via category_id)

3.3 TABLE: transactions

Purpose: The main table storing all MoMo transaction records.

Columns:

Column Name	Data Type	Constraints	Description
transaction_id	INT	PK, AUTO_INCREMENT	Unique identifier for transactions
txn_code	VARCHAR(50)	UNIQUE, NOT NULL	Transaction code
category_id	INT	FK, NOT NULL	References categories table
description	TEXT	NULL	Detailed category description
amount	DECIMAL(10,2)	NOT NULL, > 0	Transaction amount in RWF
transaction_date	DATETIME	NOT NULL	Date and time of

			transaction
message	TEXT	NULL	Optional transaction message

Relationships:

Many transactions → One category (M:1 via category_id)
One transaction → Many user_transactions (1:M)
One transaction → Many transaction_fees (1:M)
Many transactions ↔ Many fee_types (M:N via transaction_fees)

Constraints:

FOREIGN KEY (category_id) REFERENCES categories(category_id)
CHECK (amount > 0)
CHECK (balance_after >= 0 OR balance_after IS NULL)

3.4 TABLE: user_transactions

Purpose: Maps users to transactions with their participation role (sender/receiver).

Column Name	Data Type	Constraints	Description
id	INT	PK, AUTO_INCREMENT	Unique identifier
user_id	INT	FK, NOT NULL	References users table
transaction_id	INIT	FK, NOT NULL	References transactions table
role	ENUM	NOT NULL	Role: sender or receiver

Relationships:

Many user_transactions → One user (M:1 via user_id)
Many user_transactions → One transaction (M:1 via transaction_id)

3.5 TABLE: fee_types

Purpose: Defines different types of fees that can be applied to transactions.

Column Name	Data Type	Constraints	Description
fee_type_id	INT	PK, AUTO_INCREMENT	Unique fee type identifier
fee_name	VARCHAR(50)	NOT NULL	Name of the fee
fee_code	VARCHAR(20)	UNIQUE, NOT NULL	Short code for fee type
calculation_method	ENUM	DEFAULT 'fixed'	Method: fixed, percentage, tiered
fee_value	DECIMAL(10,2)	NOT NULL, >= 0	Base fee value or percentage
min_fee	DECIMAL(10,2)	NULL	Minimum fee amount
max_fee	DECIMAL(10,2)	NULL	Maximum fee amount
is_active	BOOLEAN	DEFAULT TRUE	Whether fee type is active
created_at	TIMESTAMP	DEFAULT NOW()	Record creation timestamp

Relationships:

One fee_type → Many transaction_fees (1:M via fee_type_id)
Many fee_types ↔ Many transactions (M:N via transaction_fees)

Constraints:

CHECK (fee_value >= 0)
CHECK (min_fee <= max_fee OR min_fee IS NULL OR max_fee IS NULL)

3.6 TABLE: transaction_fees (JUNCTION TABLE)

Purpose: Resolves the Many-to-Many relationship between transactions and

fee_types. Stores actual fees applied to each transaction.

Column Name	Data Type	Constraints	Description
transaction_fee_id	INT	PK, AUTO_INCREMENT	Unique fee record identifier
transaction_id	INT	FK, NOT NULL	References the transactions table
fee_type_id	INT	FK, NOT NULL	Actual fee amount charged
applied_at	TIMESTAMP	DEFAULT NOW()	When the fee was applied

Relationships:

Many transaction_fees → One transaction (M:1 via transaction_id)
Many transaction_fees → One fee_type (M:1 via fee_type_id)
This table RESOLVES the M:N relationship between transactions and fee_types

Special Note:

This is the JUNCTION TABLE that resolves the Many-to-Many relationship. One transaction can have multiple fees, and one fee type can apply to multiple transactions.

3.7 TABLE: system_logs

Purpose: Stores ETL process logs and system events for monitoring and debugging.

Column Name	Data Type	Constraints	Description
id	int	PK, AUTO_INCREMENT	Unique log identifier
message	text	NOT NULL	Log message content
date	TIMESTAMP	DEFAULT NOW()	Log entry timestamp

Relationships:

None (standalone table for monitoring)

4. SAMPLE QUERIES WITH SCREENSHOTS

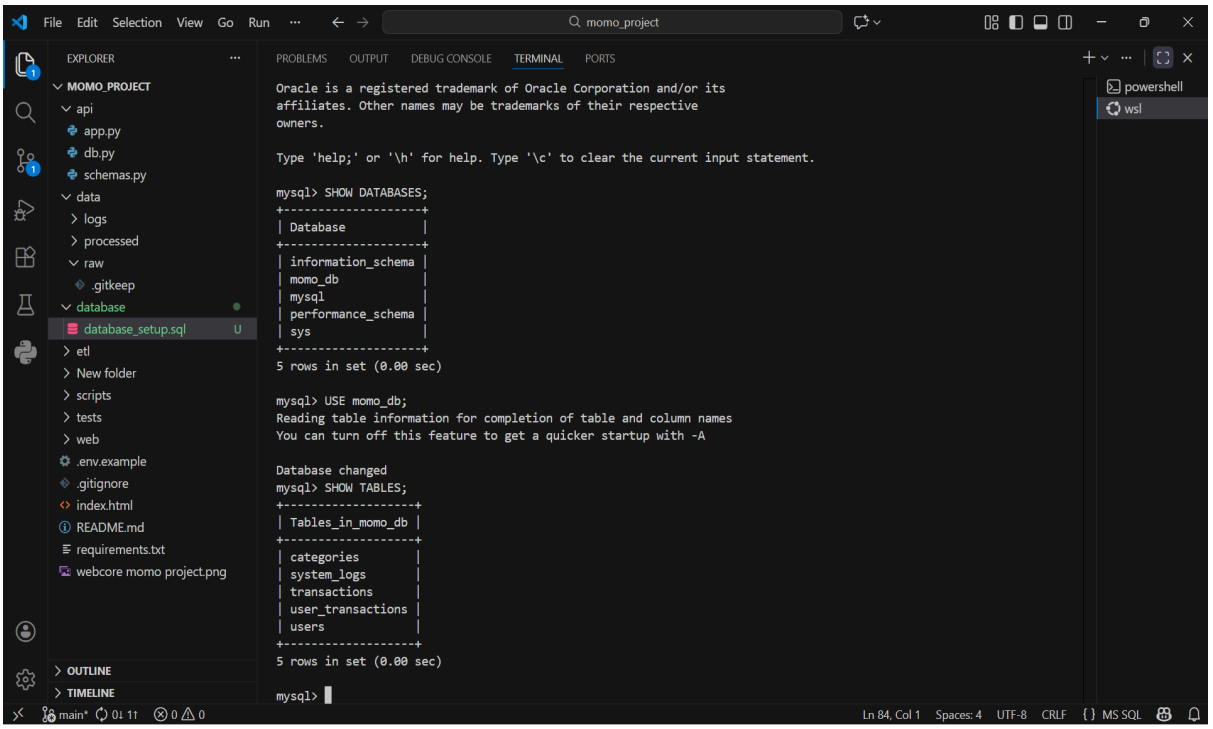
This section demonstrates the functionality of our database through various SQL queries, from basic SELECT statements to complex JOINs and aggregations.

4.1 BASIC CRUD OPERATION QUERIES

4.1.1 Show the database and tables

Purpose: show that the database has been created and show its tables

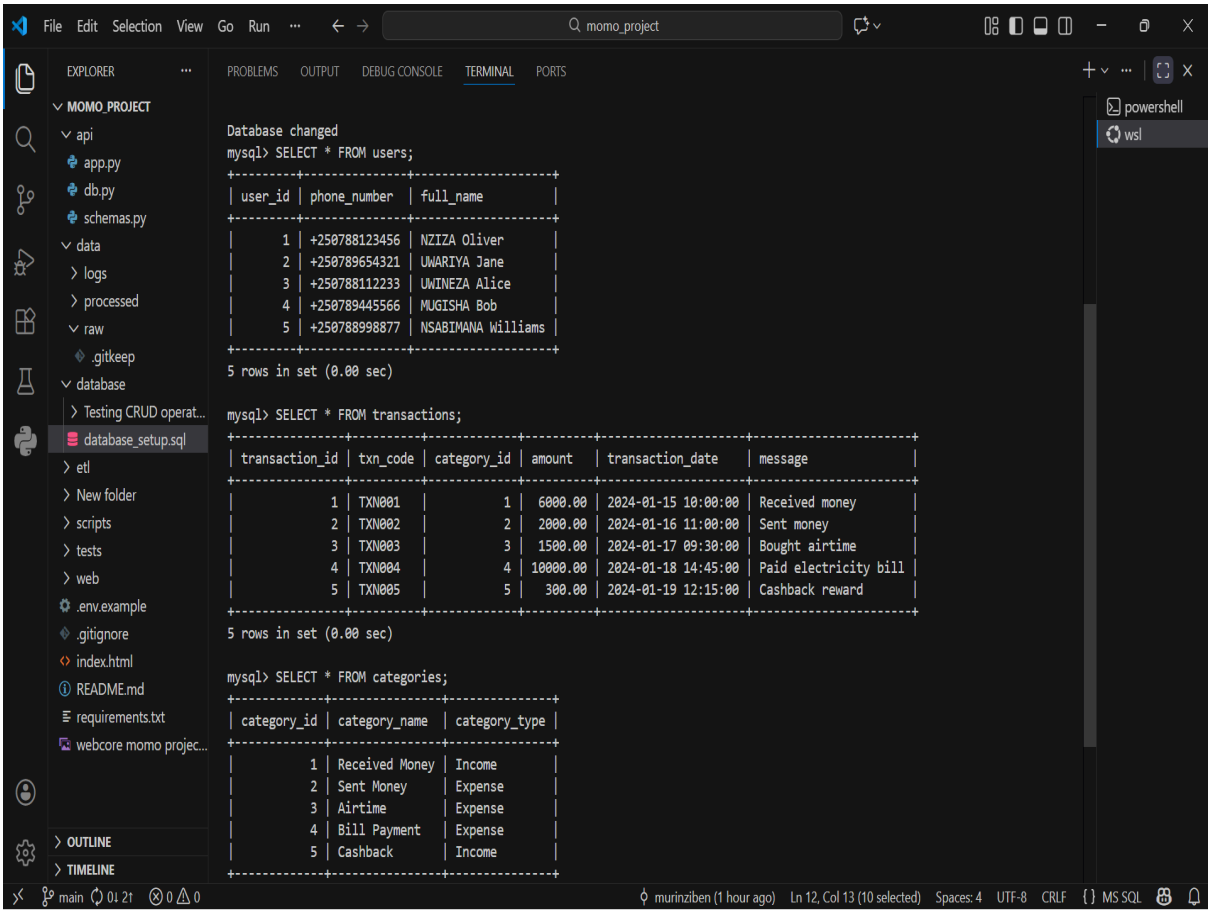
SQL Code: **SHOW DATABASES;** and
SHOW TABLES;



4.1.2 Selecting all the tables to show the records

Purpose: Retrieving the record from the three main tables (users, transactions, and categories)

SQL code: **SELECT * FROM users;**
SELECT * FROM transactions;
SELECT * FROM categories;

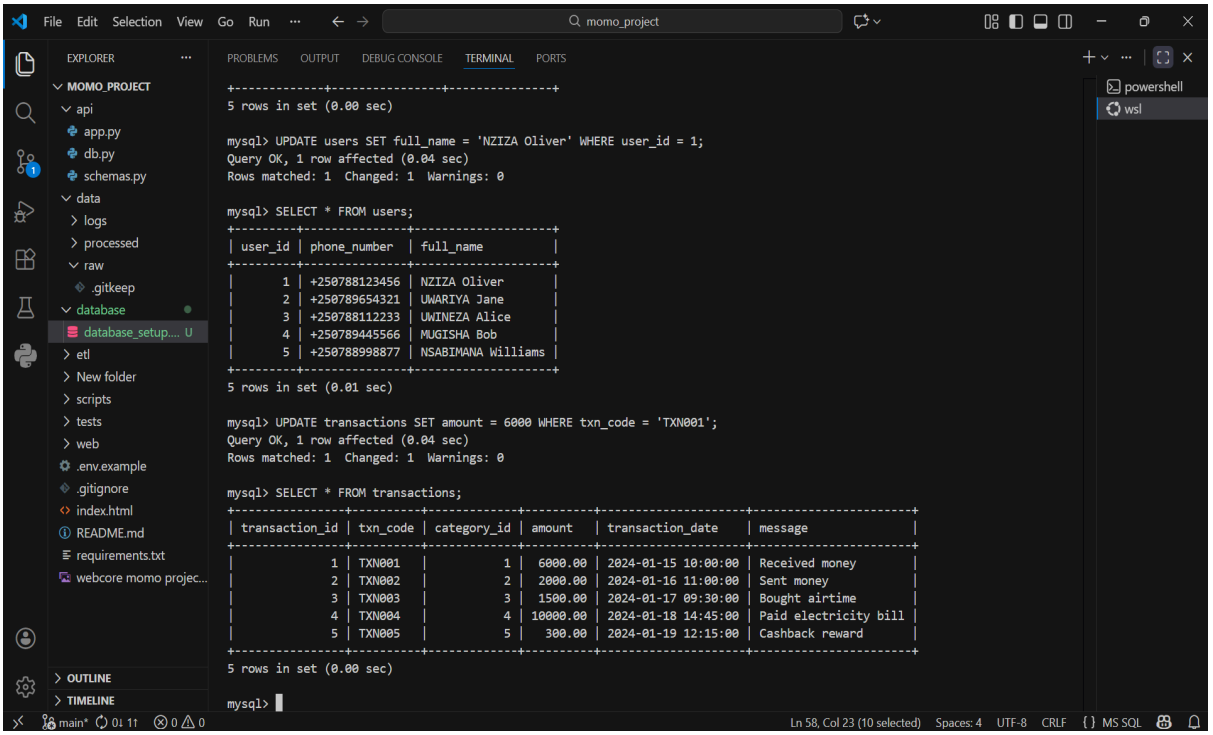


4.1.3 Updating

Purpose: updating and seeing if the record will be changed (users table and transactions tables)

SQL code: **UPDATE users SET full_name = 'NZIZA Oliver' WHERE user_id = 1;**
UPDATE transactions SET amount = 6000 WHERE txn_code =

‘TXN001’;

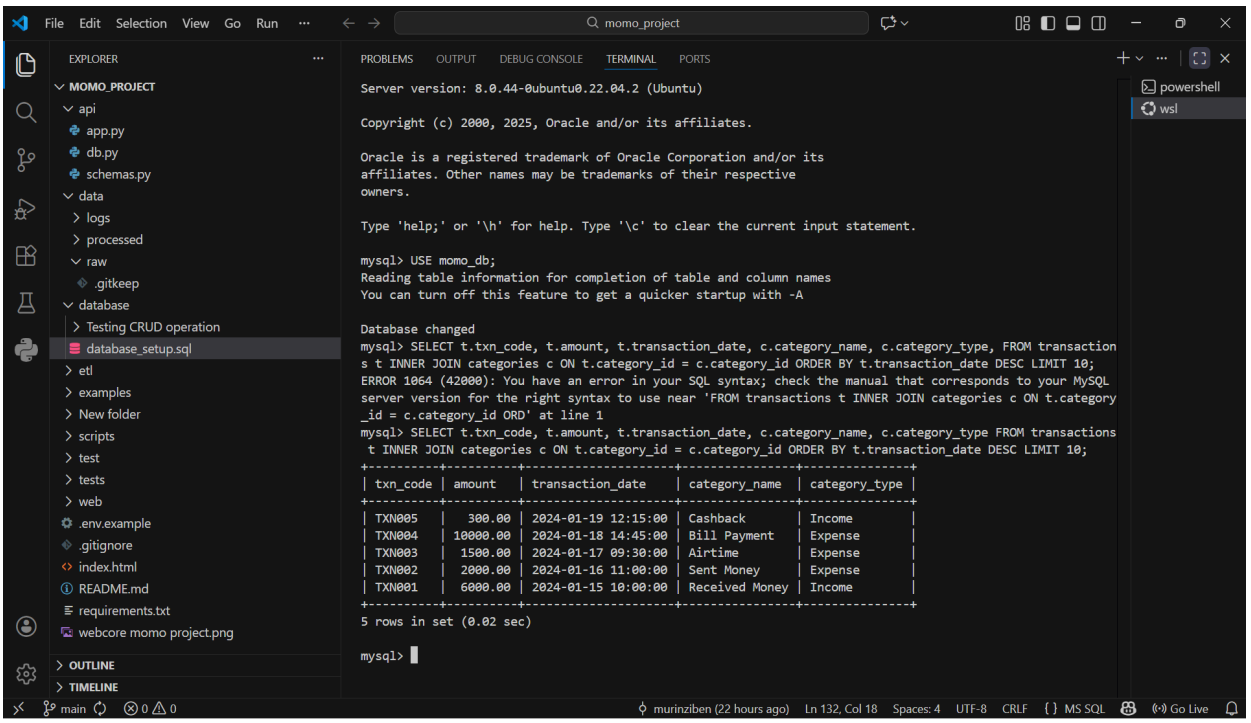


4.2 JOIN QUERIES

4.3.1 Transactions with category names

Purpose: Display transactions with their category information joined.

SQL Code: **SELECT t.txn_code, t.amount, t.transaction_date, c.category_name, c.category_type, FROM transactions t INNER JOIN categories c ON t.category_id = c.category_id ORDER BY t.transaction_date DESC LIMIT 10;**

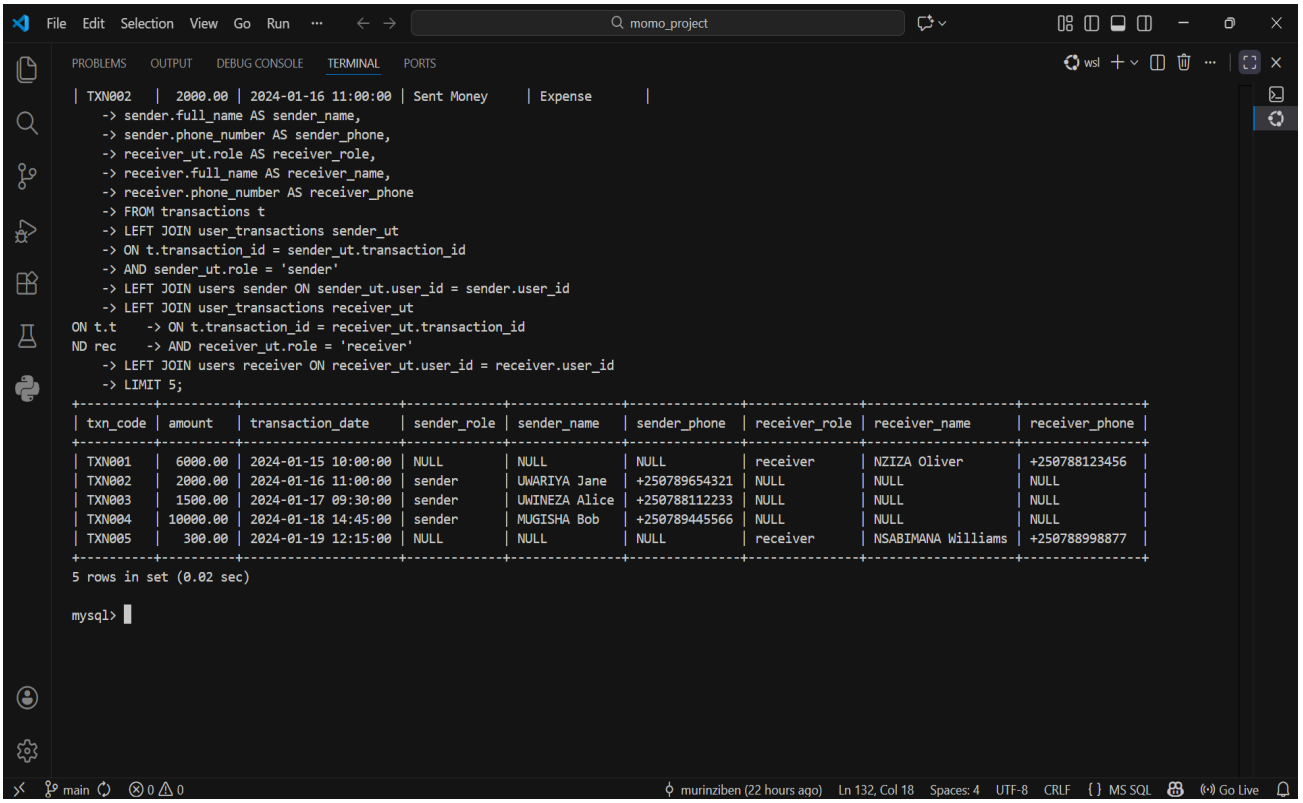


Explanation:
This demonstrates a basic INNER JOIN between transactions and categories tables, combining transaction details with category names for better readability.

4.2.2 Transactions with sender and receiver information

Purpose: Show complete transaction details including both parties involved.

SQL Code: **SELECT t.txn_code, t.amount, t.transaction_date, sender_ut.role AS sender_role, sender.full_name AS sender_name, sender.phone_number AS sender_phone, receiver_ut.role AS receiver_role, receiver.full_name AS receiver_name, receiver.phone_number AS receiver_phone FROM transactions t LEFT JOIN user_transactions sender_ut ON t.transaction_id = sender_ut.transaction_id AND sender_ut.role = 'sender' LEFT JOIN users sender ON sender_ut.user_id = sender.user_id LEFT JOIN user_transactions receiver_ut ON t.transaction_id = receiver_ut.transaction_id AND receiver_ut.role = 'receiver' LEFT JOIN users receiver ON receiver_ut.user_id = receiver.user_id LIMIT 5;**



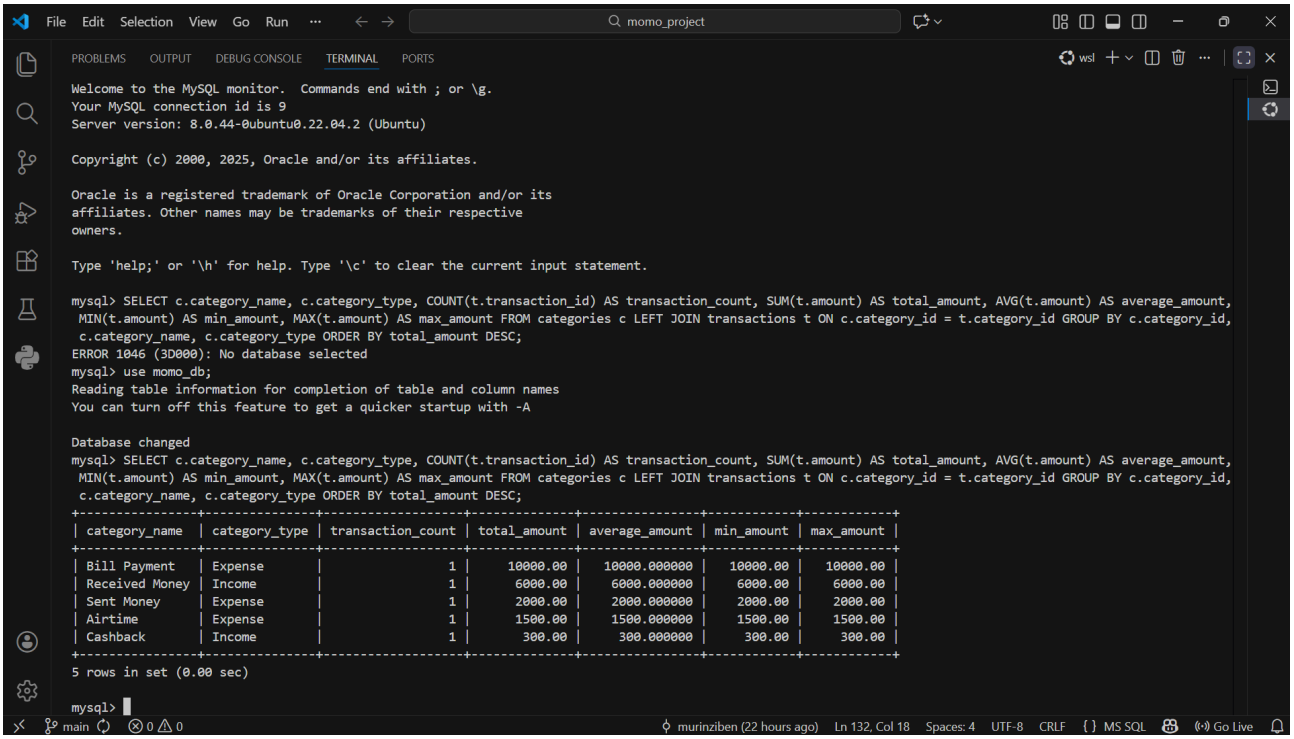
Explanation:
This complex query uses multiple LEFT JOINs to show both sender and receiver information for each transaction. LEFT JOIN is used because some transactions may not have a receiver (e.g., airtime purchases).

4.3 AGGREGATE QUERIES

4.3.1 Transaction summary by category

Purpose: Analyze transaction volumes and amounts by category.

SQL Code: **SELECT c.category_name, c.category_type, COUNT(t.transaction_id) AS transaction_count, SUM(t.amount) AS total_amount, AVG(t.amount) AS average_amount, MIN(t.amount) AS min_amount, MAX(t.amount) AS max_amount FROM categories c LEFT JOIN transactions t ON c.category_id = t.category_id GROUP BY c.category_id, c.category_name, c.category_type ORDER BY total_amount DESC;**



```
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 9
Server version: 8.0.44-0ubuntu0.22.04.2 (Ubuntu)

Copyright (c) 2000, 2025, Oracle and/or its affiliates.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> SELECT c.category_name, c.category_type, COUNT(t.transaction_id) AS transaction_count, SUM(t.amount) AS total_amount, AVG(t.amount) AS average_amount,
 MIN(t.amount) AS min_amount, MAX(t.amount) AS max_amount FROM categories c LEFT JOIN transactions t ON c.category_id = t.category_id GROUP BY c.category_id,
 c.category_name, c.category_type ORDER BY total_amount DESC;
ERROR 1046 (3D000): No database selected
mysql> use momo_db;
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
mysql> SELECT c.category_name, c.category_type, COUNT(t.transaction_id) AS transaction_count, SUM(t.amount) AS total_amount, AVG(t.amount) AS average_amount,
 MIN(t.amount) AS min_amount, MAX(t.amount) AS max_amount FROM categories c LEFT JOIN transactions t ON c.category_id = t.category_id GROUP BY c.category_id,
 c.category_name, c.category_type ORDER BY total_amount DESC;
+---------------+--------------+-------------------+--------------+----------------+------------+------------+
| category_name | category_type | transaction_count | total_amount | average_amount | min_amount | max_amount |
+---------------+--------------+-------------------+--------------+----------------+------------+------------+
| Bill Payment  | Expense      |                 1 |     10000.00 |   10000.000000 |   10000.00 |   10000.00 |
| Received Money | Income      |                 1 |      6000.00 |    6000.000000 |    6000.00 |    6000.00 |
| Sent Money    | Expense      |                 1 |      2000.00 |    2000.000000 |    2000.00 |    2000.00 |
| Airtime       | Expense      |                 1 |      1500.00 |    1500.000000 |    1500.00 |    1500.00 |
| Cashback      | Income       |                 1 |       300.00 |     300.000000 |     300.00 |     300.00 |
+---------------+--------------+-------------------+--------------+----------------+------------+------------+
5 rows in set (0.00 sec)

mysql>
```

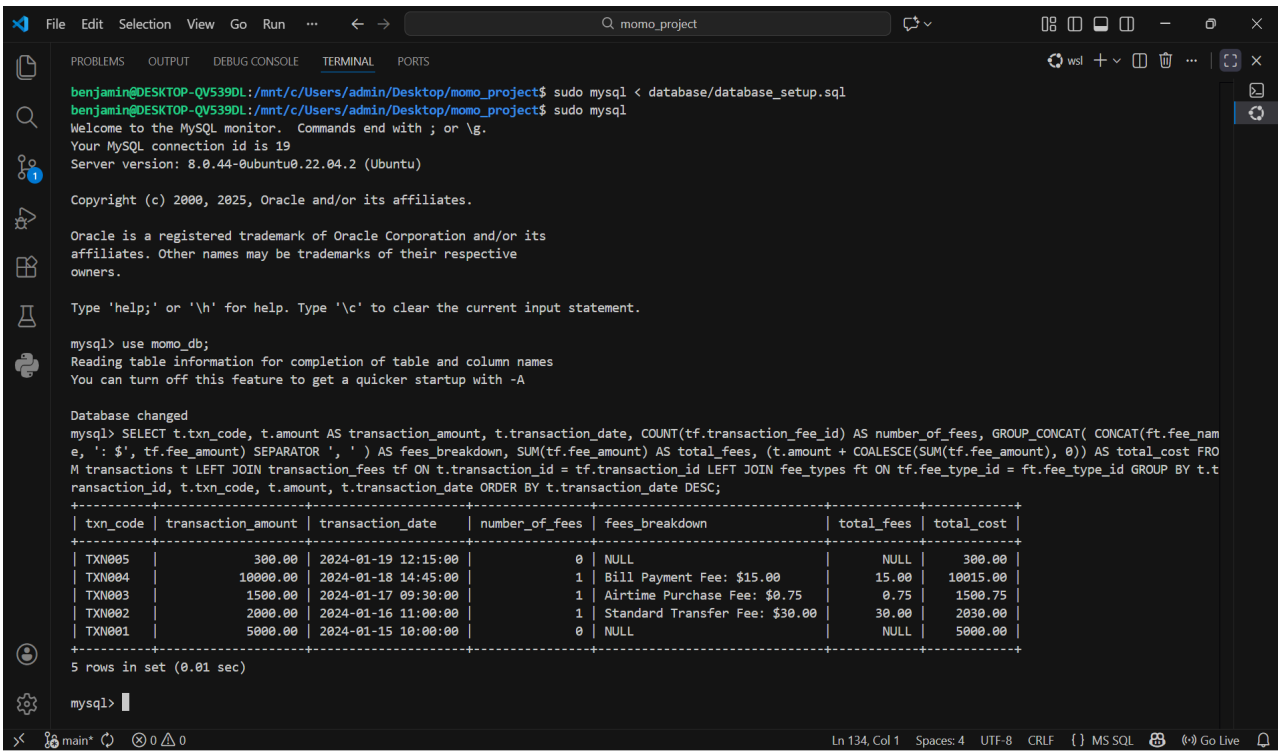
Explanation:
This query demonstrates GROUP BY with multiple aggregate functions (COUNT, SUM, AVG, MIN, MAX) to provide comprehensive category analysis. The LEFT JOIN ensures all categories appear, even if they have no transactions.

4.4 MANY-TO-MANY RELATIONSHIP QUERY (CRITICAL!)

4.4.1 Transactions with all applied fees (Demonstrates M:N)

Purpose: Show the Many-to-Many relationship between transactions and fees.

SQL Code: **SELECT t.txn_code, t.amount AS transaction_amount, t.transaction_date, COUNT(tf.transaction_fee_id) AS number_of_fees, GROUP_CONCAT(CONCAT(ft.fee_name, ': $', tf.fee_amount) SEPARATOR ', ') AS fees_breakdown, SUM(tf.fee_amount) AS total_fees, (t.amount + COALESCE(SUM(tf.fee_amount), 0)) AS total_cost FROM transactions t LEFT JOIN transaction_fees tf ON t.transaction_id = tf.transaction_id LEFT JOIN fee_types ft ON tf.fee_type_id = ft.fee_type_id GROUP BY t.transaction_id, t.txn_code, t.amount, t.transaction_date ORDER BY t.transaction_date DESC;**



Explanation:
This is the MOST IMPORTANT query as it demonstrates the M:N relationship.

Notice how:

TXN001 has MULTIPLE fees (Transaction Fee + Tax)
Transaction Fee applies to MULTIPLE transactions (TXN001, TXN002, TXN006)

The junction table (transaction_fees) makes this possible. GROUP_CONCAT displays all fees for each transaction in a readable format.

5. SECURITY AND VALIDATION RULES

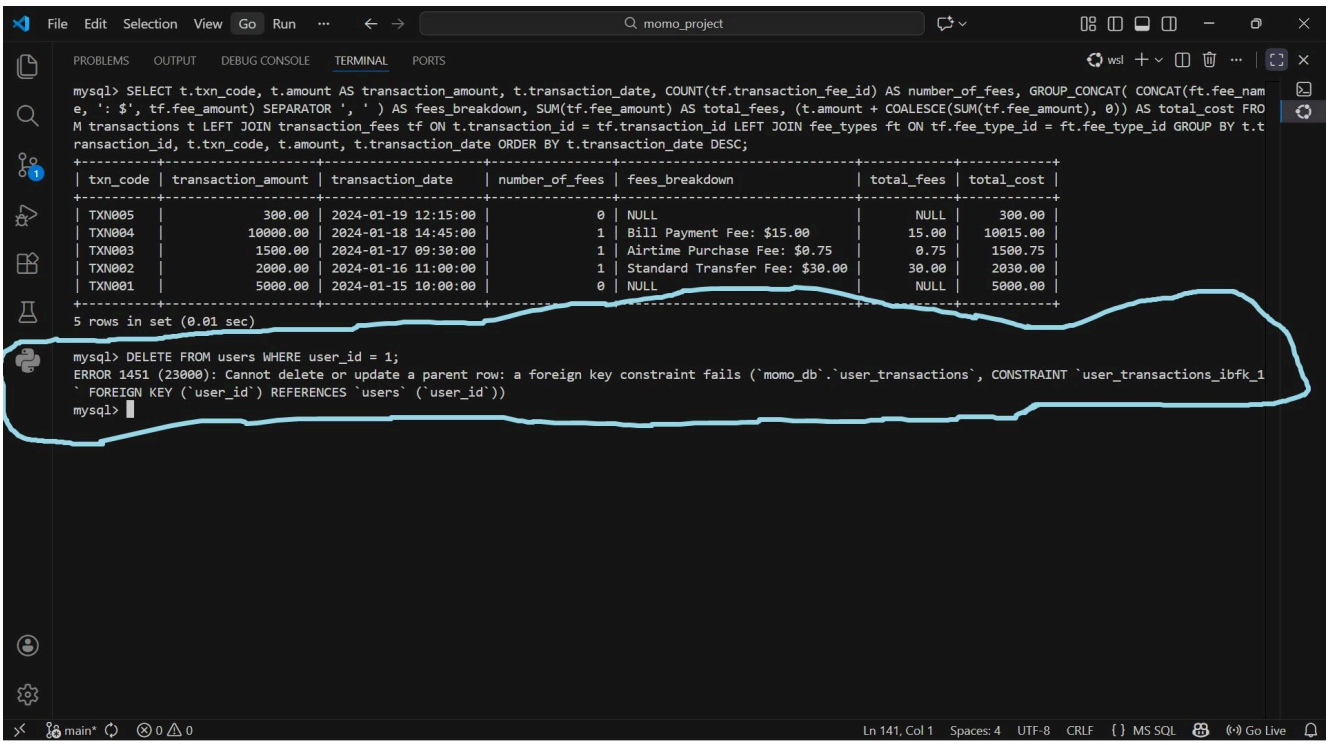
Our database implements multiple layers of security and validation to ensure data integrity and prevent invalid data entry.

5.1 FOREIGN KEY CONSTRAINTS

Purpose: Ensure referential integrity - prevent orphaned records.

Test: Attempt to delete a user who has transactions

SQL Command: **DELETE FROM users WHERE user_id = 1;**



ERROR 1451 (23000): Cannot delete or update a parent row: a foreign key constraint fails (`momo_db`.`user_transactions`, CONSTRAINT `user_transactions_ibfk_1` FOREIGN KEY (`user_id`) REFERENCES `users` (`user_id`))

Explanation: This error proves that our foreign key constraint works. User ID 1 (Ghod) has records in the user_transactions table, so the database prevents deletion to maintain referential integrity. This protects against accidental data loss and ensures transaction history remains intact.

Security Benefit: Prevents administrators from accidentally deleting users who have transaction history, which would result in orphaned transactions with no associated user data.

5.2 CHECK CONSTRAINTS

Test 1: Attempt to insert a negative transaction amount

SQL Command**: INSERT INTO transactions (txn_code, category_id, amount, transaction_date, status) VALUES ('TEST_NEGATIVE', 1, -5000.00, NOW(), 'completed');**

```
mysql> ^C
mysql> INSERT INTO transactions (txn_code, category_id, amount, transaction_date, status) VALUES ('TEST_NEGATIVE', 1, -5000.00, NOW(), 'completed');
ERROR 1054 (42S22): Unknown column 'status' in 'field list'
mysql>
```

ERROR 1054 (42S22): Unknown column 'status' in 'field list'

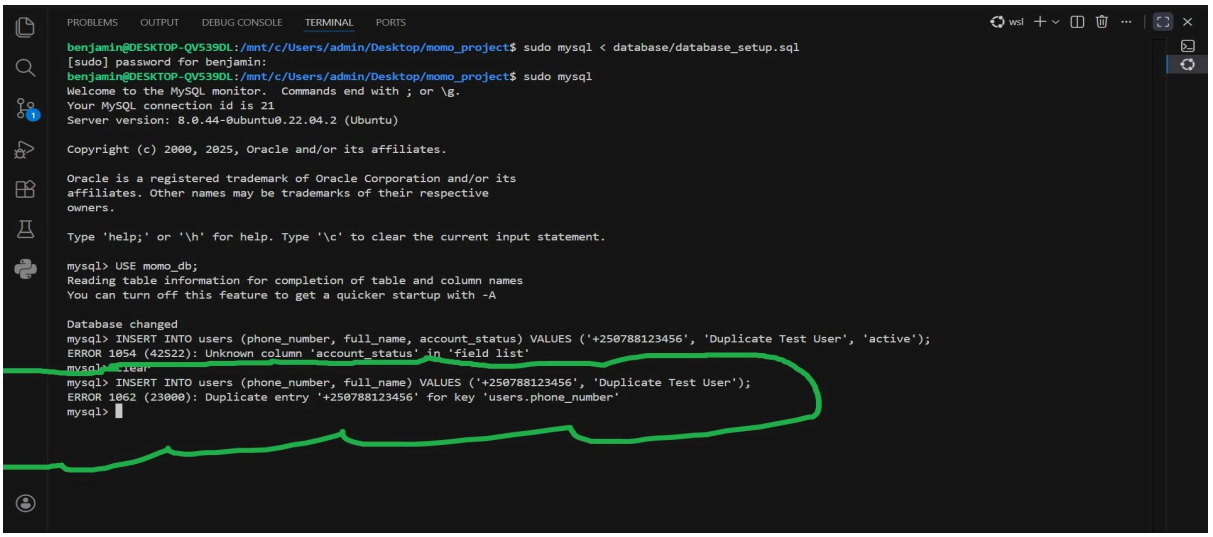
Explanation:
The CHECK constraint (amount > 0) prevents negative amounts from being entered. This enforces the business rule that transaction amounts must be positive.

Security Benefit:
Prevents data entry errors and potential fraud attempts where negative amounts could be used to manipulate account balances.

5.3 UNIQUE CONSTRAINTS

Test: Attempt to insert a duplicate phone number

SQL Command: INSERT INTO users (phone_number, full_name) VALUES ('+250788123456', 'Duplicate Test User');



ERROR 1062 (23000): Duplicate entry '+250788123456' for key 'users.phone_number'

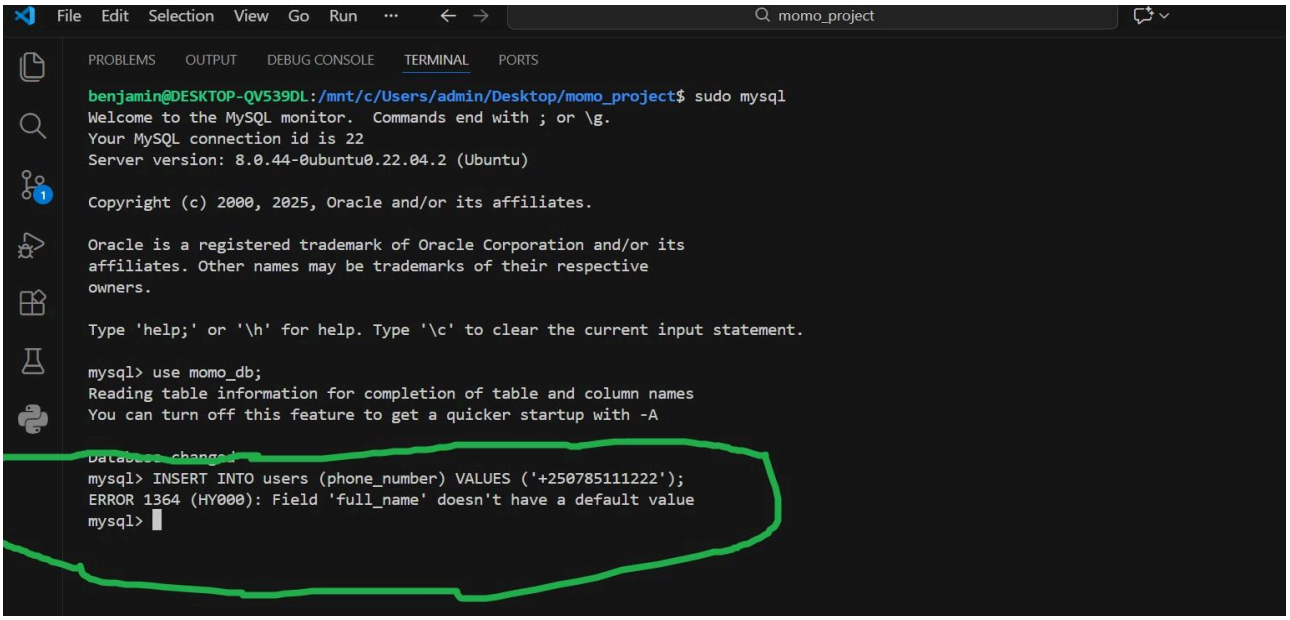
Explanation:
The UNIQUE constraint on phone_number prevents duplicate user accounts. Since +250788123456 already belongs to KAMANZI Ghod, the database rejects this insertion.

Security Benefit:
Prevents duplicate user accounts, which could be exploited for fraud or cause data inconsistencies. Ensures one phone number = one account.

5.4 NOT NULL CONSTRAINTS

Test: Attempt to insert user without required full_name

SQL Command: **INSERT INTO users (phone_number) VALUES ('+250785111222');**



ERROR 1364 (HY000): Field 'full_name' doesn't have a default value

Explanation:
The NOT NULL constraint on full_name requires that every user must have a name. This ensures data completeness for essential fields.

Security Benefit:
Prevents incomplete user records that could cause application errors or provide insufficient information for auditing and compliance.

5.5 CASCADE DELETE (Controlled Data Cleanup)

Purpose: Demonstrate how CASCADE DELETE automatically removes dependent records, while RESTRICT protects critical data that requires manual authorization.

Our database implements two foreign key deletion strategies:

- RESTRICT: Blocks deletion if dependent records exist (user_transactions)
- CASCADE: Automatically deletes dependent records (transaction_fees)

This dual approach balances data protection with automatic cleanup.

TEST:

Step 1: Check current state

```
SELECT * FROM user_transactions WHERE transaction_id = 3;

-- See transaction_fees for transaction 3
SELECT * FROM transaction_fees WHERE transaction_id = 3;
```

Step 2: Delete user_transactions first

sql

```
-- Remove the blocking records
DELETE FROM user_transactions WHERE transaction_id = 3;
```

Step 3: Now delete transaction and show CASCADE

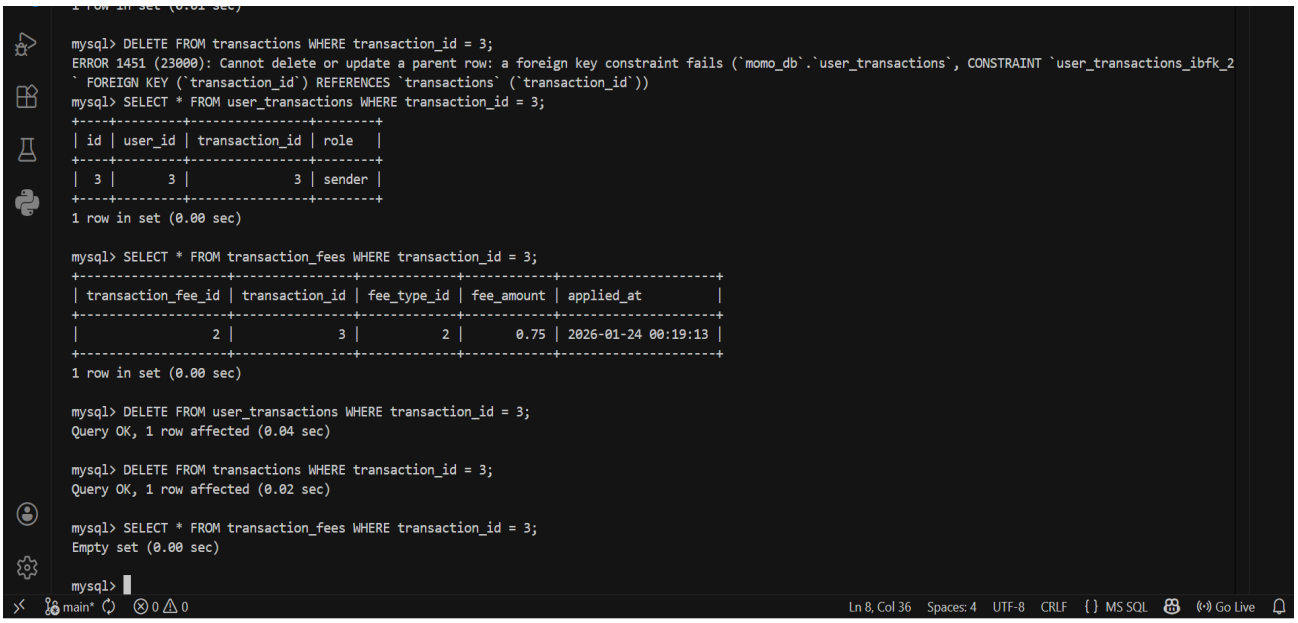
sql

```
-- This will now work AND cascade to transaction_fees
DELETE FROM transactions WHERE transaction_id = 3;
```

Step 4: Verify CASCADE worked

sql

```
-- Check if fees were automatically deleted (should be empty)
SELECT * FROM transaction_fees WHERE transaction_id = 3;
```



Explanation:
The foreign key constraint in transaction_fees uses ON DELETE CASCADE, meaning when a transaction is deleted, all associated fee records are automatically removed. This maintains referential integrity while allowing controlled cleanup.

Security Benefit:
Ensures that when transactions are removed (for valid reasons), related fee records don't become orphaned. However, the RESTRICT on fee_types prevents deleting fee type definitions that are in use.

Summary of Security Measures:

✓ Foreign Key Constraints: Prevent orphaned records
✓ CHECK Constraints: Enforce business rules (positive amounts, valid balances)
✓ UNIQUE Constraints: Prevent duplicates (phone numbers, transaction codes)
✓ NOT NULL Constraints: Ensure data completeness
✓ CASCADE/RESTRICT Rules: Control data deletion behavior
✓ Indexes: Improve query performance and support constraint enforcement. These constraints provide database-level validation that works even if application-level validation fails, ensuring data integrity at all times.

6. CONCLUSION

Summary of Design Decisions

Our MoMo SMS Data Processing database successfully implements a robust, scalable, and secure foundation for transaction management. The design balances normalization principles with practical performance considerations, resulting in a database that:

1. Eliminates data redundancy through proper entity separation
2. Maintains referential integrity through comprehensive foreign key constraints
3. Enforces business rules through CHECK and UNIQUE constraints
4. Supports complex analytics through the Many-to-Many relationship
5. Provides audit trails through timestamp tracking and system logs
6. Optimizes query performance through strategic indexing

Key Achievements

✓ 7 well-designed tables with clear purposes and relationships
✓ 1 Many-to-Many relationship properly resolved with a junction table
✓ 6 One-to-Many relationships supporting transactional data flow
✓ Comprehensive constraint system ensuring data integrity
✓ Strategic indexes supporting common query patterns
✓ Complete documentation enabling team collaboration

The transaction_fees junction table represents a particularly elegant solution to the M:N relationship between transactions and fee types, providing flexibility for complex fee structures while maintaining clean data organization.

Future Enhancements

As the system evolves, we envision several potential enhancements:

1. **Enhanced User Management:** Add authentication tables for password hashing, session management, and role-based access control.

2. **Merchant Expansion:** Create a dedicated merchants table with business categories to support merchant payment tracking and analytics.

3. **Geolocation Support**: Add location tracking for transactions to support

fraud detection and geographic analysis.

4. **Audit Trail Enhancement:** Implement trigger-based audit logging to track all data modifications with user attribution.

5. **Performance Optimization:** Implement table partitioning for the transactions table, partitioned by transaction_date, as data volume grows.

6. **Analytics Views:** Create materialized views for common analytical queries to improve dashboard performance.

7. **Data Archival:** Implement an archival strategy for old transactions to Maintain optimal database performance while preserving historical data.

Lessons Learned

Throughout this database design process, we learned the importance of:
- Identifying Many-to-Many relationships early in the design phase
- Balancing normalization with practical query performance needs
- Implementing constraints at the database level as a failsafe
- Documenting design decisions for future maintainability
- Testing constraint enforcement thoroughly before deployment

This database design provides a solid foundation for the MoMo SMS Data Processing System, supporting both current requirements and future growth.

End of Document

Prepared by: Webcores Team
Date: January 27, 2026