Database Design Document – MoMo SMS Data Processor

1. Introduction

The MoMo SMS Data Processor database is designed to store, clean, and analyze mobile money SMS data.

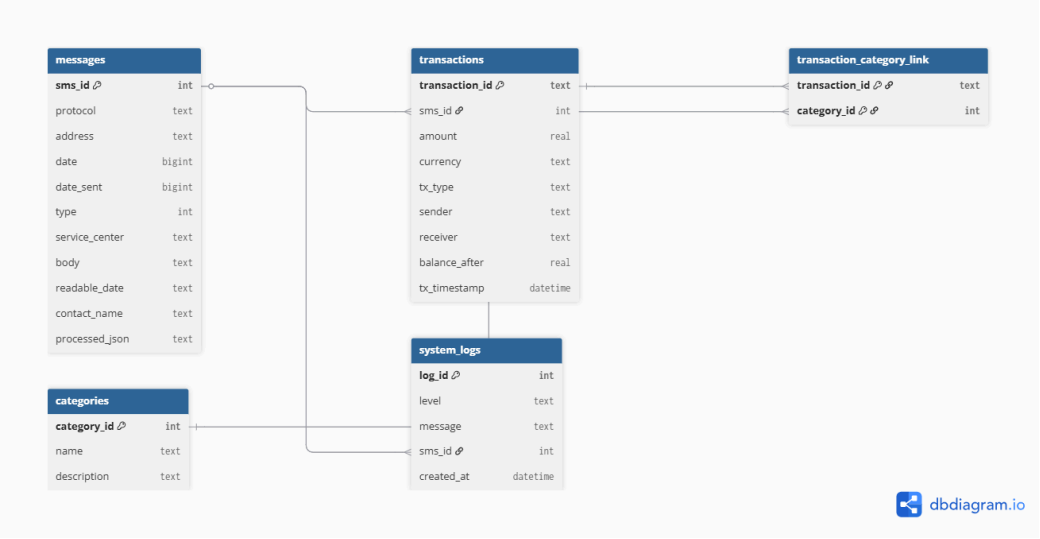
Incoming SMS messages contain unstructured financial information such as deposits, transfers, and payments.

This schema organizes the raw data into normalized tables so that transactions can be tracked, categorized, and audited while maintaining full access to the original SMS for reprocessing or verification.

2. Entity-Relationship Diagram (ERD)

The ERD shows five main entities:

- **messages** – raw SMS records from the mobile network.
- **transactions** – parsed and cleaned transaction details derived from the messages.
- **categories** – predefined transaction types (deposit, payment, transfer, etc.).
- **transaction_category_link** – junction table to support many-to-many relationships between transactions and categories.
- **system_logs** – ETL process logs and error tracking.



3. Design Rationale & Justification

The ETL workflow specified in our system architecture and the MoMo SMS XML structure are both reflected in our database schema.

All transactional information is contained in the raw messages that are provided by the XML input.

We divide raw storage, parsed transactions, and processing logs into distinct entities to maintain data integrity and facilitate analytics.

Every original SMS, complete with metadata like protocol, address, date, and message body, is kept in the messages table exactly as it was received.

Every record is guaranteed to be auditable and reprocessable in the event that the parsing logic changes thanks to this raw layer.

From these messages, our ETL pipeline parses business details (amount, currency, sender, receiver, and timestamps) into the transactions table, creating a clean and query-friendly dataset.

We introduce a categories lookup table and a transaction_category_link junction table to resolve this many-to-many relationship because a single transaction can belong to multiple categories (for instance, a deposit that is also a promotional transfer).

Without duplicating data, this design allows for flexible analytics, such as filtering by promotional activity or payment type.

Errors and dead-letter events from the ETL process are recorded in the system_logs table to facilitate debugging and ensure traceability.

Effective joins for dashboards and reports are made possible by foreign keys and indexes, which also maintain referential integrity.

Overall, this schema offers a strong basis for storing, querying, and analyzing MoMo transaction data by striking a balance between normalization, scalability, and operational transparency.

---

4. Data Dictionary

Table	Field	Type	Key	Description
messages	sms_id	INT	PK	Internal key for each SMS
	address	VARCHAR(64)		Sender/service name
	date	BIGINT		Epoch time received
	body	TEXT		Full SMS text
	...	...	...	Other metadata (protocol, service_center, etc.)
transactions	transaction_id	VARCHAR(64)	PK	Unique transaction ID
	sms_id	INT	FK → messages.sms_id	Links to source SMS
	amount	DECIMAL(18,2)		Transaction amount
	tx_type	VARCHAR(32)		deposit, payment, transfer...
	tx_timestamp	DATETIME		Parsed transaction time
categories	category_id	INT	PK	Category identifier
	name	VARCHAR(64)	UNIQUE	Category name
	description	TEXT		Details of category
transaction_category_link	transaction_id	VARCHAR(64)	PK/FK	Junction to transactions
	category_id	INT	PK/FK	Junction to categories
system_logs	log_id	INT	PK	Log entry
	level	VARCHAR(16)		INFO/WARN/ERROR

Table	Field	Type	Key	Description
	message	TEXT		Description of event
	sms_id	INT	FK → messages.sms_id	Related message (optional)

5. SQL Implementation

The database is implemented in MySQL 8.0 using `InnoDB`.

All tables include primary keys, foreign keys for referential integrity, and indexes on frequently queried fields (e.g., transaction date, type).

A CHECK constraint restricts `tx_type` to known values such as `deposit`, `payment`, `transfer`, `withdrawal`, `airtime`, and `other`.

Sample data were inserted for each main table to support testing and dashboard development.

(Include key excerpts from `database_setup.sql` with proper citations:
"Initial table structure drafted with AI assistance (ChatGPT, 2025) and reviewed by the DEVSQUAD team.")

6. Sample CRUD Queries and Results

1️⃣ User Transaction Summary

(Shows totals for one sender/receiver across categories)

```
-- Get transaction summary for a specific sender
SELECT
   t.sender_name,
   t.sender_msisdn,
   c.name AS category_name,
   COUNT(*) AS transaction_count,
   SUM(CASE WHEN t.tx_type IN ('payment','transfer') THEN t.amount ELSE 0 END) AS total_sent,
   SUM(CASE WHEN t.tx_type = 'deposit' THEN t.amount ELSE 0 END) AS total_received,
   SUM(t.fee) AS total_fees
FROM transactions t
JOIN transaction_category_link l ON t.transaction_id = l.transaction_id
JOIN categories c ON l.category_id = c.category_id
WHERE t.sender_name = 'Samuel Carter'
GROUP BY t.sender_name, t.sender_msisdn, c.name
ORDER BY total_sent DESC;
```

Expected Results

sender_name	sender_msisdn	category_name	transaction_count	total_sent	total_received	total_fees
Samuel Carter	250*******013	deposit	1	0.00	25000.00	0.00
Samuel Carter	250*******013	transfer	1	10000.00	0.00	100.00

2️⃣ Daily Transaction Volume

(Aggregate by day and category)

```
SELECT
    DATE(tx_timestamp) AS transaction_day,
    c.name AS category_name,
    COUNT(*) AS transaction_count,
    SUM(amount) AS total_amount,
    SUM(fee) AS total_fees,
    ROUND(AVG(amount),2) AS avg_transaction_amount
FROM transactions t
JOIN transaction_category_link l ON t.transaction_id = l.transaction_id
JOIN categories c ON l.category_id = c.category_id
GROUP BY DATE(tx_timestamp), c.name
ORDER BY transaction_day DESC, total_amount DESC;
```

Expected Result

transaction_day	category_name	transaction_count	total_amount	total_fees	avg_transaction_amount
2024-05-14	deposit	1	25000.00	0.00	25000.00
2024-05-12	payment	1	10900.00	0.00	10900.00
2024-05-11	transfer	1	10000.00	100.00	10000.00
2024-05-11	payment	1	2000.00	0.00	2000.00
2024-05-10	deposit	1	2000.00	0.00	2000.00

3️⃣ **Failed Transaction Analysis**
(Simulated error log query using `system_logs` *as an example)*

```
-- Identify common ETL errors (simulating failed transactions)
SELECT
    JSON_EXTRACT(context, '$.file') AS error_source,
    COUNT(*) AS failure_count,
    COUNT(CASE WHEN level = 'ERROR' THEN 1 END) AS error_events,
    ROUND(COUNT(CASE WHEN level = 'ERROR' THEN 1 END)*100.0/COUNT(*),2) AS error_rate
FROM system_logs
GROUP BY JSON_EXTRACT(context, '$.file')
ORDER BY failure_count DESC;
```

Expected Result

error_source	failure_count	error_events	error_rate
"momo.xml"	1	1	100.00

💣

---

7. Unique Rules and Integrity Constraints

- **Transaction Type Constraint**: `tx_type` is limited to defined values to prevent invalid categories.
- **Foreign Key Enforcement**: Deleting a message automatically sets `sms_id` in related transactions to `NULL`, preserving historical transaction records.

- **Unique Category Names**: Prevents duplicate category entries.

Unique Rules for Enhanced Security and Accuracy

To ensure data accuracy and prevent duplicates, the following unique rules were applied to the database:

1. **Unique Email for Users**
   - Each user must register with a unique email address.
   - Prevents multiple accounts with the same email, improving security and login accuracy.
2. **Unique Phone Number for Contacts**
   - Phone numbers are unique in the `contacts` table.
   - Avoids duplication of contacts and ensures correct communication.
3. **Unique Vehicle Number (for motor-taxi app example)**
   - Each vehicle number is stored only once.
   - Prevents duplication in the fleet database, ensuring correct vehicle tracking.
4. **Primary Key Constraints**
   - Every table has a primary key that uniquely identifies each record.
   - Guarantees that every record is distinguishable and ensures data integrity.

AI Assistance Disclosure

During the preparation of this report,

ChatGPT (OpenAI, 2025)

was used

only for language editing and formatting support

.

No ERD design, SQL schema generation, or database logic was produced by AI.

All database structures, relationships, and business rules were independently created and verified by the DEVSQUAD team.
