Banking Core - Database Normalization Analysis

Schema Design Justification & Normal Form Compliance

Version: 1.0.0
Last Updated: January 2026
Target Audience: DBMS Undergraduate Students, Faculty
Database: MySQL 8.0 (InnoDB)

1. Introduction to Normalization in Banking Systems

1.1 Purpose of Normalization

Database normalization is the process of organizing a relational database to:

1. **Minimize redundancy** – Store each fact only once
2. **Prevent anomalies** – Avoid insert, update, and delete problems
3. **Ensure data integrity** – Maintain consistency across tables

1.2 Banking-Specific Considerations

Financial databases have unique requirements that influence normalization decisions:

Requirement	Impact on Design
Transaction Integrity	Strict normalization for financial records
Audit Compliance	Immutable audit trails (append-only)
Performance	Strategic denormalization for balance lookups
Regulatory Requirements	Complete history preservation

1.3 Normal Forms Overview

Normal Form	Rule
1NF	Atomic values, no repeating groups
2NF	1NF + No partial dependencies on composite keys
3NF	2NF + No transitive dependencies
BCNF	Every determinant is a candidate key

2. Functional Dependencies Analysis

A **functional dependency** X → Y means that for each value of X, there is exactly one value of Y.

2.1 Core Entity Tables

roles

```
id → {code, name, description, permissions, is_system, created_at, updated_at}
code → {id, name, description, permissions, is_system}
```

Candidate Keys: {id}, {code}

users

```
id → {email, password_hash, first_name, last_name, role_id, status, last_login_at, ...}
email → {id, password_hash, first_name, last_name, role_id, ...}
```

Candidate Keys: {id}, {email}

customers

```
id → {customer_number, email, first_name, last_name, phone, national_id, ...}
customer_number → {id, email, first_name, ...}
email → {id, customer_number, first_name, ...}
```

Candidate Keys: {id}, {customer_number}, {email}

2.2 Account Tables

account_types

```
id → {code, name, description, min_balance, is_active}
code → {id, name, description, min_balance, is_active}
```

Candidate Keys: {id}, {code}

accounts

```
id → {account_number, customer_id, account_type_id, status, opened_at, ...}
account_number → {id, customer_id, account_type_id, status, ...}
```

Candidate Keys: {id}, {account_number}

account_balances

```
account_id → {available_balance, pending_balance, hold_balance, currency, version, ...}
```

Candidate Keys: {account_id}

2.3 Transaction Tables

transaction_types

```
id → {code, name, description, requires_approval}
code → {id, name, description, requires_approval}
```

Candidate Keys: {id}, {code}

transactions

```
id → {transaction_reference, transaction_type_id, amount, currency, status,
      source_account_id, destination_account_id, processed_at, ...}
transaction_reference → {id, transaction_type_id, amount, ...}
```

Candidate Keys: {id}, {transaction_reference}

ledger_entries

```
id → {transaction_id, account_id, entry_type, amount, currency, balance_after,
      description, entry_date, created_at}
```

Candidate Keys: {id}

2.4 Audit Tables

transaction_audit

```
id → {ledger_entry_id, transaction_id, account_id, entry_type, amount,
      balance_after, audit_timestamp}
ledger_entry_id → {id, transaction_id, account_id, ...}
```

Candidate Keys: {id}, {ledger_entry_id}

audit_logs

```
id → {actor_id, actor_type, actor_role, action_type, entity_type, entity_id,
      before_state, after_state, metadata, created_at}
```

Candidate Keys: {id}

3. First Normal Form (1NF) Compliance

3.1 Requirements for 1NF

1. Each column contains only **atomic (indivisible) values**
2. Each column contains values of a **single type**
3. Each row is **uniquely identifiable** (has a primary key)
4. No **repeating groups** or arrays within a column

3.2 Table-by-Table 1NF Analysis

Table	Atomic Values	Single Type	PK Exists	No Repeating Groups	1NF?
roles	✅	✅	✅ (id)	⚠️ See Note 1	✅
users	✅	✅	✅ (id)	✅	✅
customers	✅	✅	✅ (id)	✅	✅
account_types	✅	✅	✅ (id)	✅	✅
accounts	✅	✅	✅ (id)	✅	✅
account_balances	✅	✅	✅ (account_id)	✅	✅
transaction_types	✅	✅	✅ (id)	✅	✅
transactions	✅	✅	✅ (id)	✅	✅
ledger_entries	✅	✅	✅ (id)	✅	✅
transaction_audit	✅	✅	✅ (id)	✅	✅
audit_logs	✅	✅	✅ (id)	⚠️ See Note 2	✅
system_config	✅	✅	✅ (id)	✅	✅
idempotency_keys	✅	✅	✅ (key)	⚠️ See Note 3	✅
events	✅	✅	✅ (id)	⚠️ See Note 3	✅
outbox	✅	✅	✅ (id)	⚠️ See Note 3	✅

3.3 Notes on JSON Columns

Note 1 (roles.permissions): Contains JSON array like `["read:*", "write:*"]`

- **Justification:** While JSON could be considered non-atomic, MySQL 8 treats JSON as a first-class type. Permissions are read as a whole set, not queried individually.
- **Alternative:** A separate `role_permissions` junction table would add complexity for minimal benefit.
- **Conclusion:** Acceptable trade-off; still considered 1NF compliant.

Note 2 (audit_logs.before_state, after_state, metadata): Contains JSON objects

- **Justification:** Audit logs capture arbitrary entity states. A fully normalized design would require a separate audit table per entity type.
- **Conclusion:** JSON provides flexibility for audit trails; 1NF compliant.

Note 3 (events/outbox.payload): Contains JSON event data

- **Justification:** Event payloads vary by event type. Normalizing would require dozens of event-specific tables.
- **Conclusion:** Standard practice for event sourcing; 1NF compliant.

4. Second Normal Form (2NF) Compliance

4.1 Requirements for 2NF

1. Must be in **1NF**
2. **No partial dependencies** – All non-key attributes must depend on the entire primary key

4.2 Analysis of Composite Keys

2NF is primarily concerned with tables that have **composite primary keys**. In Banking Core:

Table	Primary Key	Type
All core tables	Single column (id)	Simple
account_balances	account_id	Simple
idempotency_keys	idempotency_key	Simple

Finding: All tables use **single-column primary keys**, so **partial dependencies are impossible**.

4.3 2NF Compliance Summary

Table	1NF	Single-Column PK	Partial Dependencies	2NF?
roles	✅	✅	N/A	✅
users	✅	✅	N/A	✅

Table	1NF	Single-Column PK	Partial Dependencies	2NF?
customers	✅	✅	N/A	✅
account_types	✅	✅	N/A	✅
accounts	✅	✅	N/A	✅
account_balances	✅	✅	N/A	✅
transaction_types	✅	✅	N/A	✅
transactions	✅	✅	N/A	✅
ledger_entries	✅	✅	N/A	✅
transaction_audit	✅	✅	N/A	✅
audit_logs	✅	✅	N/A	✅
system_config	✅	✅	N/A	✅
idempotency_keys	✅	✅	N/A	✅
events	✅	✅	N/A	✅
outbox	✅	✅	N/A	✅

Result: All tables satisfy **2NF** by design (single-column PKs).

5. Third Normal Form (3NF) Compliance

5.1 Requirements for 3NF

1. Must be in **2NF**
2. **No transitive dependencies** – Non-key attributes must not depend on other non-key attributes

5.2 Transitive Dependency Analysis

A transitive dependency exists when: `A → B → C` (A determines B, B determines C, but B is not a candidate key)

users table

```
id → role_id → role_name (via JOIN)
```

- **Analysis:** `role_name` is NOT stored in users table
- **Stored:** Only `role_id` (FK reference)
- **Result:** ✅ No redundancy, 3NF compliant

accounts table

```
id → customer_id → customer_name (via JOIN)
id → account_type_id → account_type_name (via JOIN)
```

- **Analysis:** Customer and type names are NOT stored in accounts
- **Stored:** Only foreign key IDs
- **Result:** ✅ 3NF compliant

ledger_entries table

```
id → transaction_id → transaction_amount (via JOIN)
id → account_id → account_number (via JOIN)
```

- **Analysis:** Redundant data NOT stored
- **Stored:** Only references
- **Result:** ✅ 3NF compliant

5.3 Controlled Denormalization (3NF Exceptions)

Two tables contain **intentional denormalization** for performance:

account_balances.available_balance

```
Normalized approach: SUM(ledger_entries.amount) WHERE entry_type = 'CREDIT'
                   - SUM(ledger_entries.amount) WHERE entry_type = 'DEBIT'

Denormalized: account_balances.available_balance (cached value)
```

Justification:

- Balance queries happen on every transaction

- Summing 100,000+ ledger entries per query is O(n) expensive
- Cached balance provides O(1) lookup
- **Integrity maintained:** Stored procedures update both ledger AND balance atomically

ledger_entries.balance_after

```
Normalized approach: SUM all prior entries to calculate balance at any point

Denormalized: balance_after stored directly in each entry
```

Justification:

- Historical balance queries are common (statements, audits)
- Recalculating from genesis would be extremely expensive
- balance_after is calculated and stored at insert time
- **Never updated** – append-only table ensures integrity

5.4 3NF Compliance Summary

Table	2NF	Transitive Dependencies	Exception Justified	3NF?
roles	✅	None	N/A	✅
users	✅	None	N/A	✅
customers	✅	None	N/A	✅
account_types	✅	None	N/A	✅
accounts	✅	None	N/A	✅
account_balances	✅	Cached balance	✅ Performance	⚠️ Denormalized
transaction_types	✅	None	N/A	✅
transactions	✅	None	N/A	✅
ledger_entries	✅	balance_after	✅ Historical queries	⚠️ Denormalized
transaction_audit	✅	None	N/A	✅

Table	2NF	Transitive Dependencies	Exception Justified	3NF?
audit_logs	✅	None	N/A	✅
system_config	✅	None	N/A	✅
idempotency_keys	✅	None	N/A	✅
events	✅	None	N/A	✅
outbox	✅	None	N/A	✅

Result: 13/15 tables fully 3NF compliant. 2 tables have **justified denormalization**.

6. BCNF (Boyce-Codd Normal Form) Consideration

6.1 Requirements for BCNF

- For every functional dependency X → Y, X must be a **superkey**
- Stricter than 3NF: addresses anomalies when multiple candidate keys overlap

6.2 BCNF Analysis

Table	Candidate Keys	All Determinants Superkeys?	BCNF?
roles	{id}, {code}	✅	✅
users	{id}, {email}	✅	✅
customers	{id}, {customer_number}, {email}	✅	✅
account_types	{id}, {code}	✅	✅
accounts	{id}, {account_number}	✅	✅
account_balances	{account_id}	✅	✅
transaction_types	{id}, {code}	✅	✅

Table	Candidate Keys	All Determinants Superkeys?	BCNF?
transactions	{id}, {transaction_reference}	✅	✅
ledger_entries	{id}	✅	✅
transaction_audit	{id}, {ledger_entry_id}	✅	✅
audit_logs	{id}	✅	✅
system_config	{id}, {config_key}	✅	✅
idempotency_keys	{idempotency_key}	✅	✅
events	{id}	✅	✅
outbox	{id}	✅	✅

Result: All tables satisfy **BCNF**.

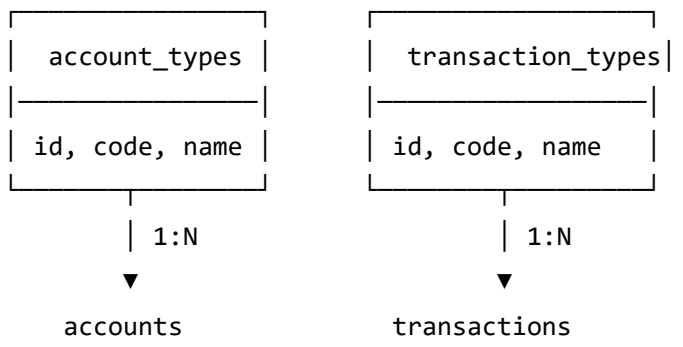
7. Schema Design Justification

7.1 Separation of Concerns

Design Decision	Justification
`users` vs `customers`	Staff and customers have different attributes, workflows, and security models
`transactions` vs `ledger_entries`	Transaction is a business event; ledger entries are accounting records
`accounts` vs `account_balances`	Account metadata (static) separated from balance (dynamic)
`transaction_audit` vs `ledger_entries`	Audit copy provides regulatory compliance verification

7.2 Reference Table Pattern

Lookup/reference tables are normalized separately:



Benefits:

- Type names updated in one place
- New types added without schema changes
- Consistent categorization across system

7.3 Double-Entry Ledger Design

```
transactions (1) ──────< ledger_entries (N)
```

Each transaction generates exactly 2 ledger entries:

- **DEBIT** entry (money leaving account)
- **CREDIT** entry (money entering account)

Normalization Benefit: Transaction metadata stored once, not duplicated per entry.

8. Anomaly Prevention

8.1 Insert Anomalies

Scenario	How Schema Prevents
Adding account without customer	`customer_id NOT NULL` enforces relationship
Adding transaction without type	`transaction_type_id NOT NULL` required
Adding ledger entry without transaction	`transaction_id NOT NULL` enforces parent

8.2 Update Anomalies

Scenario	How Schema Prevents
Renaming account type	Update once in `account_types`, all accounts inherit via FK
Changing role permissions	Update once in `roles`, all users inherit
Customer address change	Single record in `customers`, no duplication

8.3 Delete Anomalies

Scenario	How Schema Prevents
Deleting customer with accounts	Application-level FK check prevents orphans
Deleting transaction type in use	Referenced transactions would be orphaned (blocked)
Ledger entries never deleted	Append-only design preserves history

9. Trade-offs & Performance Considerations

9.1 Normalization vs. Performance Trade-offs

Trade-off	Decision	Reason
Balance calculation	**Denormalized** (cached)	O(1) vs O(n) lookup
Transaction type name	**Normalized** (FK join)	Infrequent updates, small table
Customer in ledger	**Normalized** (via account FK)	Avoids massive redundancy
Historical balance	**Denormalized** (balance_after)	Statement generation performance

9.2 Join Performance Analysis

Query Pattern	Tables Joined	Index Support	Performance
Get account balance	accounts + account_balances	PK lookup	O(1)

Query Pattern	Tables Joined	Index Support	Performance
List customer accounts	accounts WHERE customer_id	FK index	O(log n)
Transaction history	transactions + ledger_entries	FK index	O(log n)
Audit trail	audit_logs	Multiple indexes	O(log n)

9.3 Index Strategy

Table	Index	Purpose
users	email (unique)	Login lookup
accounts	customer_id	List customer's accounts
accounts	account_number (unique)	Transfer destination lookup
ledger_entries	transaction_id	Transaction details
ledger_entries	account_id, entry_date	Statement queries
audit_logs	(actor_id, actor_type)	Who did what
audit_logs	(entity_type, entity_id)	What happened to entity
audit_logs	created_at	Time-based queries

10. Conclusion

10.1 Normalization Summary

Normal Form	Tables Fully Compliant	Tables with Justified Exceptions
1NF	15/15 (100%)	0
2NF	15/15 (100%)	0
3NF	13/15 (87%)	2 (account_balances, ledger_entries)
BCNF	15/15 (100%)	0

10.2 Key Design Strengths

1. **Clean Separation** – Identity, accounts, transactions, and audit cleanly separated
2. **Minimal Redundancy** – Data stored once, referenced via FKs
3. **Controlled Denormalization** – Only where performance requires it
4. **Audit Integrity** – Immutable audit trails for regulatory compliance
5. **Flexible Extensions** – JSON columns for variable-structure data

10.3 Anomaly Prevention Summary

Anomaly Type	Status
Insert Anomalies	✅ Prevented via NOT NULL constraints
Update Anomalies	✅ Prevented via normalized reference tables
Delete Anomalies	✅ Prevented via FK enforcement + append-only tables

10.4 Final Assessment

The Banking Core database schema demonstrates **strong normalization discipline** appropriate for a financial system:

- Core transactional tables are in **3NF/BCNF**
- Denormalization is **intentional and justified** for performance-critical paths
- **Audit compliance** is maintained through append-only ledger and separate audit tables
- The design **prevents all three types of anomalies** while supporting high-performance banking operations

This document is suitable for DBMS coursework, viva examinations, and academic project reports.