Banking Core - Transaction & ACID Compliance Case Study

A DBMS Course Case Study on Financial Transaction Management

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

1. Introduction to Transactions in Banking Systems

1.1 What is a Database Transaction?

A **transaction** is a sequence of database operations that are executed as a single logical unit of work. In banking systems, transactions are critical because:

1. **Money must never be "lost"** – Partial transfers are unacceptable
2. **Balances must always be accurate** – No double-spending
3. **Operations must be recoverable** – System crashes cannot corrupt data
4. **Concurrent access must be safe** – Multiple tellers operating simultaneously

1.2 The ACID Properties

Property	Meaning	Banking Relevance
Atomicity	All or nothing execution	Transfer either completes fully or not at all
Consistency	Valid state to valid state	Account balances always reflect reality
Isolation	Transactions don't interfere	Two tellers can't withdraw the same money
Durability	Committed data survives failures	Power failure doesn't lose deposits

1.3 Double-Entry Accounting Rule

Banking Core implements **double-entry accounting**:

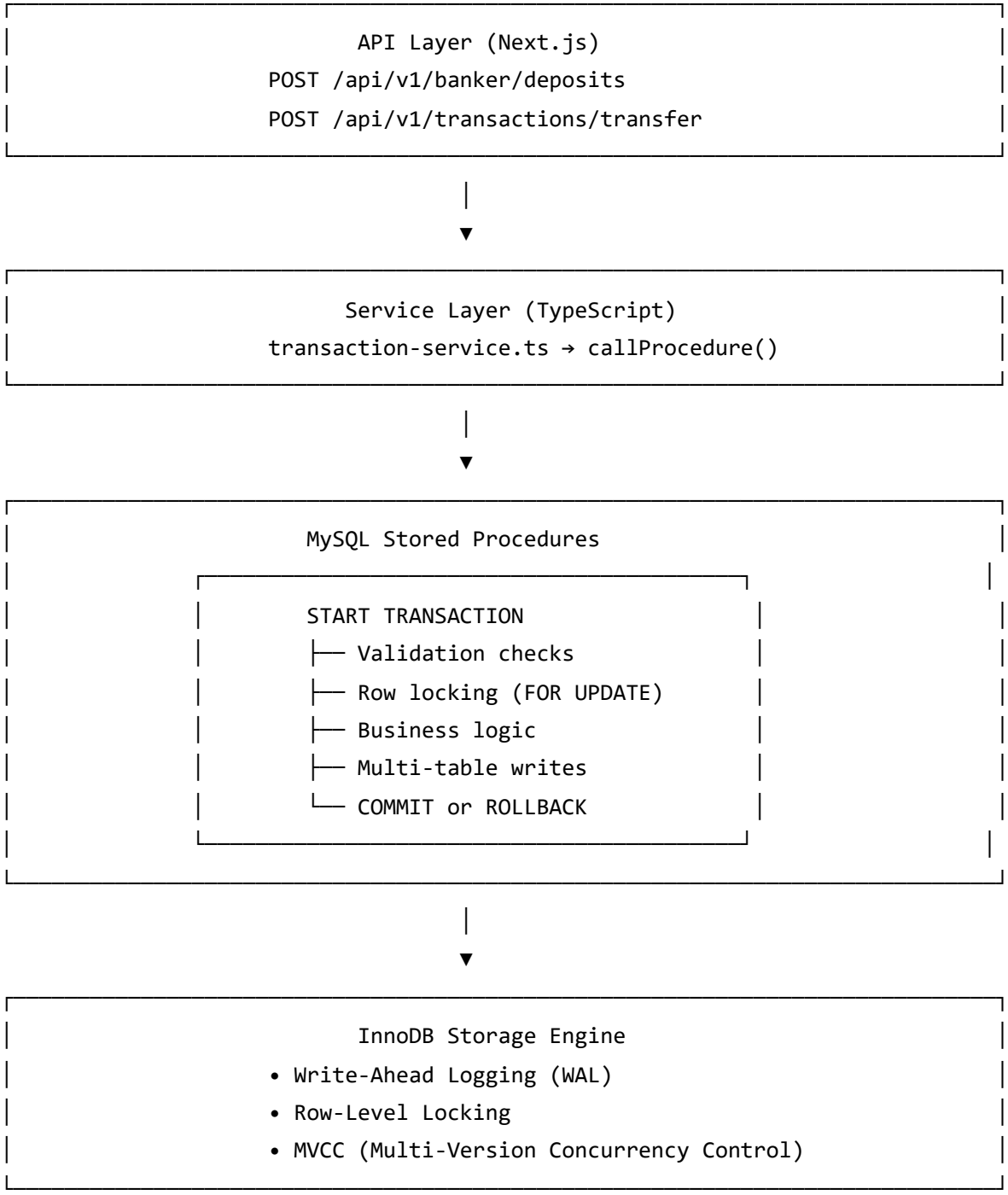
```
For every financial transaction:
    Total DEBITS = Total CREDITS
```

This means every transfer creates exactly two ledger entries:

- **DEBIT**: Money leaving one account
- **CREDIT**: Money entering another account

2. Overview of Banking Core Transaction Architecture

2.1 Architecture Diagram



2.2 Stored Procedures as Transaction Boundaries

All money movement occurs through **stored procedures**:

Procedure	Purpose	Tables Affected
`sp_deposit`	Cash deposit by teller	transactions, ledger_entries (×2), account_balances (×2)
`sp_withdraw`	Cash withdrawal by teller	transactions, ledger_entries (×2), account_balances (×2)
`sp_transfer`	Account-to-account transfer	transactions, ledger_entries (×2), account_balances (×2), idempotency_keys

2.3 Why Stored Procedures?

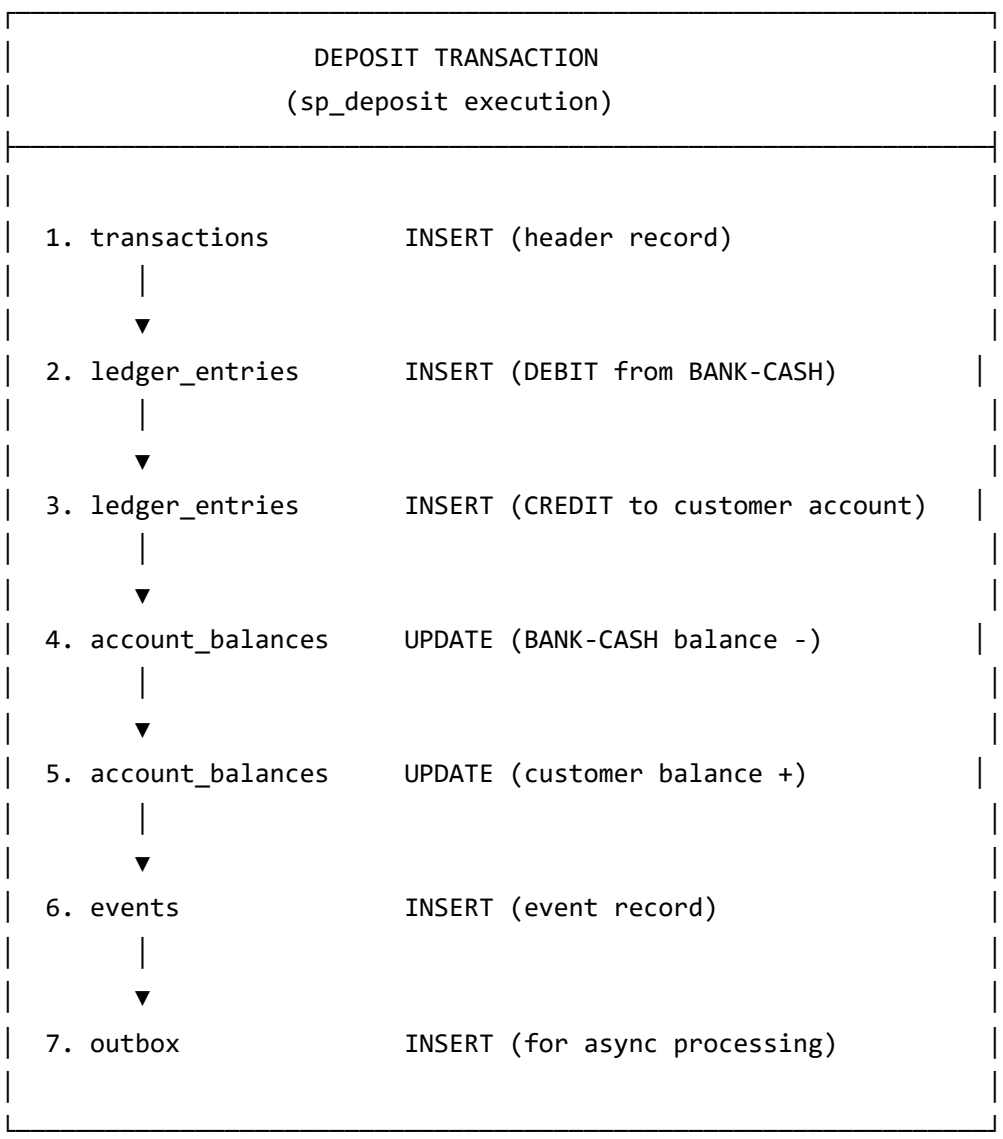
1. **Atomic Boundary** – All statements in procedure share same transaction
2. **Reduced Network Round-Trips** – Complex logic runs in database
3. **Consistent Enforcement** – Business rules cannot be bypassed
4. **Row Locking** – `SELECT ... FOR UPDATE` prevents race conditions

3. Transactional Operations in the System

3.1 Operation Classification

Operation Type	Transactional?	Stored Procedure	Tables Modified
User Login	No	N/A	users (last_login_at)
Customer View Balance	No	N/A	Read-only
Cash Deposit	**Yes**	sp_deposit	4 tables
Cash Withdrawal	**Yes**	sp_withdraw	4 tables
Fund Transfer	**Yes**	sp_transfer	5 tables
Account Freeze	No	N/A	accounts (status)

3.2 Tables Modified Per Transaction



4. ACID Property Mapping

4.1 Atomicity

> **Definition:** A transaction is an indivisible unit – either all operations succeed, or none do.

Implementation in Banking Core

```
DELIMITER //
CREATE PROCEDURE sp_transfer(...)
BEGIN
    -- Error handler: Any SQL error triggers rollback
    DECLARE EXIT HANDLER FOR SQLEXCEPTION
    BEGIN
        ROLLBACK;
        SET p_status = 'FAILED';
        SET p_message = 'Transaction failed';
    END;

    -- Begin atomic unit
    START TRANSACTION;

    -- Multiple operations...
    INSERT INTO transactions (...);
    INSERT INTO ledger_entries (...);  -- DEBIT
    INSERT INTO ledger_entries (...);  -- CREDIT
    UPDATE account_balances ...;        -- Source
    UPDATE account_balances ...;        -- Destination

    -- Complete atomic unit
    COMMIT;
END //
```

Atomicity Guarantee

Scenario	Result
All statements succeed	COMMIT – all changes permanent
Any statement fails	ROLLBACK – database unchanged
Network failure mid-transaction	MySQL auto-rollback on disconnect
Server crash	Recovery log replays or rolls back

4.2 Consistency

> **Definition:** A transaction brings the database from one valid state to another valid state.

Consistency Rules in Banking Core

Rule	Enforcement	Layer
Account balance ≥ 0	`IF balance < amount THEN ROLLBACK`	Stored Procedure
Account must be ACTIVE	`IF status != 'ACTIVE' THEN ROLLBACK`	Stored Procedure
Amount must be positive	`IF amount <= 0 THEN ROLLBACK`	Stored Procedure
DEBIT sum = CREDIT sum	Double-entry logic in procedure	Stored Procedure
Account must exist	`SELECT ... FOR UPDATE` returns null	Stored Procedure

Example: Balance Consistency Check

```
-- In sp_transfer:
SELECT available_balance INTO v_source_balance
FROM account_balances
WHERE account_id = p_from_account_id
FOR UPDATE;

-- Consistency check
IF v_source_balance < p_amount THEN
    SET p_status = 'FAILED';
    SET p_message = 'Insufficient balance';
    ROLLBACK;
END IF;
```

4.3 Isolation

> **Definition:** Concurrent transactions do not interfere with each other.

Isolation Level

Banking Core uses MySQL's default: **REPEATABLE READ**

Isolation Level	Dirty Read	Non-Repeatable Read	Phantom Read
READ UNCOMMITTED	✅ Possible	✅ Possible	✅ Possible
READ COMMITTED	❌ Prevented	✅ Possible	✅ Possible
REPEATABLE READ	❌ Prevented	❌ Prevented	⚠️ Possible

Isolation Level	Dirty Read	Non-Repeatable Read	Phantom Read
SERIALIZABLE	❌ Prevented	❌ Prevented	❌ Prevented

Row-Level Locking

```
-- Lock source account row exclusively
SELECT available_balance, status
FROM account_balances
WHERE account_id = p_from_account_id
FOR UPDATE;  -- Other transactions wait here

-- Lock destination account row
SELECT available_balance, status
FROM account_balances
WHERE account_id = p_to_account_id
FOR UPDATE;
```

Effect: While Transaction A holds the lock, Transaction B trying to access the same account row will **wait** until A commits or rolls back.

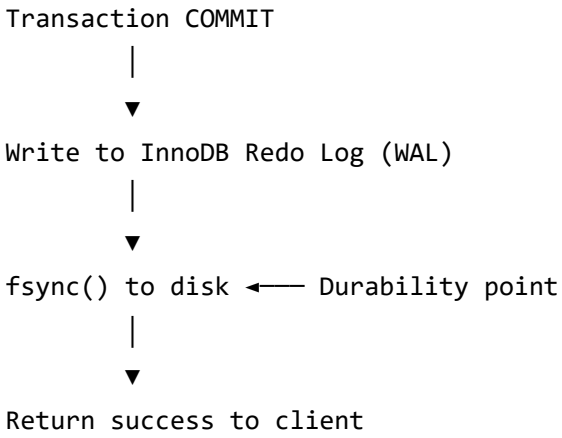
4.4 Durability

> **Definition:** Once a transaction is committed, its changes survive any subsequent system failures.

InnoDB Durability Mechanisms

Mechanism	Purpose
Write-Ahead Logging (WAL)	Changes written to redo log before data files
Double-Write Buffer	Prevents partial page writes
Checkpoint	Periodic flush of dirty pages to disk
innodb_flush_log_at_trx_commit = 1	Sync redo log on every commit

Durability Guarantee



If power fails after fsync(), the change survives.

5. Case Studies

Case Study 1: Deposit Transaction

Scenario: Teller deposits BDT 10,000 into customer Alice's savings account.

Transaction Flow

```
START TRANSACTION

Step 1: Validate
├── Check: amount > 0 ✓
├── Check: account exists ✓
└── Check: account status = 'ACTIVE' ✓

Step 2: Lock accounts
├── SELECT ... FOR UPDATE on BANK-CASH-001
└── SELECT ... FOR UPDATE on Alice's account

Step 3: Insert transaction record
└── INSERT INTO transactions (type=DEPOSIT, amount=10000, status=COMPLETED)

Step 4: Create ledger entries
├── INSERT INTO ledger_entries (BANK-CASH, DEBIT, 10000)
└── INSERT INTO ledger_entries (Alice, CREDIT, 10000)

Step 5: Update balances
├── UPDATE account_balances SET balance = balance - 10000 WHERE account = BANK-CASH
└── UPDATE account_balances SET balance = balance + 10000 WHERE account = Alice

Step 6: Record events
├── INSERT INTO events (DEPOSIT_COMPLETED)
└── INSERT INTO outbox (for notifications)

COMMIT
```

ACID Analysis

Property	How Applied
Atomicity	All 7 writes succeed or all fail
Consistency	Bank cash decreases, Alice increases by same amount
Isolation	Account rows locked during operation
Durability	COMMIT writes to redo log and syncs

Case Study 2: Withdrawal Transaction

Scenario: Teller withdraws BDT 5,000 from customer Bob's account.

Transaction Flow

```
START TRANSACTION

Step 1: Validate
├── Check: amount > 0 ✓
├── Check: Bob's account exists ✓
├── Check: account status = 'ACTIVE' ✓
└── Check: balance >= 5000 ✓

Step 2: Lock accounts
├── SELECT ... FOR UPDATE on Bob's account
└── SELECT ... FOR UPDATE on BANK-CASH-001

Step 3: Insert transaction record
└── INSERT INTO transactions (type=WITHDRAWAL, amount=5000, status=COMPLETED)

Step 4: Create ledger entries
├── INSERT INTO ledger_entries (Bob, DEBIT, 5000)
└── INSERT INTO ledger_entries (BANK-CASH, CREDIT, 5000)

Step 5: Update balances
├── UPDATE account_balances SET balance = balance - 5000 WHERE account = Bob
└── UPDATE account_balances SET balance = balance + 5000 WHERE account = BANK-CASH

COMMIT
```

Insufficient Balance Scenario

```
START TRANSACTION

Step 1: Lock Bob's account
SELECT available_balance INTO v_balance
FROM account_balances WHERE account_id = ?
FOR UPDATE;
-- v_balance = 3000

Step 2: Check balance
IF 3000 < 5000 THEN
    SET p_status = 'FAILED';
    SET p_message = 'Insufficient balance';
    ROLLBACK;  ◄─── Transaction aborted, no changes
END IF;
```

Result: No money moved, no records created, database unchanged.

Case Study 3: Transfer Transaction

Scenario: Customer Carol transfers BDT 2,000 from her savings to her checking account.

Transaction Flow

```
START TRANSACTION

Step 1: Check idempotency
SELECT response_body FROM idempotency_keys
WHERE idempotency_key = 'uuid-12345';
-- Not found, proceed

Step 2: Lock source account (Carol's Savings)
SELECT available_balance, status INTO v_from_balance, v_from_status
FROM account_balances ab JOIN accounts a ON ab.account_id = a.id
WHERE a.id = 101
FOR UPDATE;

Step 3: Lock destination account (Carol's Checking)
SELECT available_balance, status INTO v_to_balance, v_to_status
FROM account_balances ab JOIN accounts a ON ab.account_id = a.id
WHERE a.id = 102
FOR UPDATE;

Step 4: Validate
├── v_from_status = 'ACTIVE' ✓
├── v_to_status = 'ACTIVE' ✓
├── v_from_balance >= 2000 ✓
└── source != destination ✓

Step 5: Calculate new balances
├── v_new_from = v_from_balance - 2000
└── v_new_to = v_to_balance + 2000

Step 6: Insert transaction
INSERT INTO transactions (type=TRANSFER, amount=2000, source=101, dest=102);

Step 7: Insert ledger entries
├── INSERT ledger_entries (account=101, DEBIT, 2000, balance_after=v_new_from)
└── INSERT ledger_entries (account=102, CREDIT, 2000, balance_after=v_new_to)

Step 8: Update balances
├── UPDATE account_balances SET balance = v_new_from, version = version + 1 WHERE id = 101
└── UPDATE account_balances SET balance = v_new_to, version = version + 1 WHERE id = 102

Step 9: Store idempotency key
```

```
INSERT INTO idempotency_keys (key='uuid-12345', response={...}, expires=NOW()+24h);

Step 10: Record events
INSERT INTO events (TRANSFER_COMPLETED, payload={...});

COMMIT
```

Case Study 4: Failure During Transfer (Rollback)

Scenario: Power failure occurs after Step 7 but before COMMIT.

Timeline

```
T0: START TRANSACTION
T1: Lock accounts ✓
T2: Validation passed ✓
T3: INSERT into transactions ✓
T4: INSERT ledger entry (DEBIT) ✓
T5: INSERT ledger entry (CREDIT) ✓
T6: ███ POWER FAILURE ███

-- System restarts --

T7: MySQL recovery process reads redo log
T8: Transaction T0 has no COMMIT record
T9: All changes from T0 are ROLLED BACK
T10: Database state = exactly as before T0
```

What Happens to the Data?

Table	Changes in Memory	After Recovery
transactions	1 row inserted	**Rolled back** – row removed
ledger_entries	2 rows inserted	**Rolled back** – rows removed
account_balances	2 rows updated	**Rolled back** – original values

Result: The database is in the exact state it was before the transfer started. No money was lost or duplicated.

Case Study 5: Concurrent Withdrawal Scenario

Scenario: Two tellers simultaneously attempt to withdraw from the same account that has BDT 10,000.

- Teller A: Withdraw BDT 8,000
- Teller B: Withdraw BDT 7,000
- Available balance: BDT 10,000

Execution Timeline

```
Time    Teller A                        Teller B
____    ________                        ________
T1      START TRANSACTION
T2      SELECT ... FOR UPDATE           START TRANSACTION
        (balance = 10000) ✓
        Acquires LOCK on row
T3                                      SELECT ... FOR UPDATE
                                        ⏳ WAITING (row locked by A)
T4      IF 10000 >= 8000 ✓
        (validation passes)
T5      INSERT transaction              ⏳ WAITING...
T6      INSERT ledger entries           ⏳ WAITING...
T7      UPDATE balance = 2000           ⏳ WAITING...
T8      COMMIT                          Lock released!
        ________________
T9                                      SELECT ... FOR UPDATE
                                        (balance = 2000) ✓
                                        Acquires LOCK
T10                                     IF 2000 >= 7000 ✗
                                        VALIDATION FAILS!
T11                                     SET status = 'FAILED'
                                        SET message = 'Insufficient balance'
                                        ROLLBACK
```

Result

Teller	Amount	Result
A	BDT 8,000	✅ Success – balance now 2,000

Teller	Amount	Result
B	BDT 7,000	❌ Failed – insufficient balance

Key Point: The `FOR UPDATE` lock prevented the **lost update** anomaly. Without it, both transactions might have read 10,000 and both succeeded, resulting in -5,000 balance.

6. Concurrency & Isolation Deep Dive

6.1 Lock Types Used

Lock Type	SQL	Purpose
Exclusive Row Lock	`SELECT ... FOR UPDATE`	Prevent concurrent modifications
Shared Lock	`SELECT ... LOCK IN SHARE MODE`	Not used (all are exclusive)

6.2 Deadlock Prevention

Problem: If Transaction A locks Account 1 then Account 2, while Transaction B locks Account 2 then Account 1, a deadlock occurs.

Solution: Banking Core procedures always lock accounts in a **consistent order**:

```
-- Always lock lower account_id first
IF p_from_account_id < p_to_account_id THEN
    SELECT ... FROM ... WHERE account_id = p_from_account_id FOR UPDATE;
    SELECT ... FROM ... WHERE account_id = p_to_account_id FOR UPDATE;
ELSE
    SELECT ... FROM ... WHERE account_id = p_to_account_id FOR UPDATE;
    SELECT ... FROM ... WHERE account_id = p_from_account_id FOR UPDATE;
END IF;
```

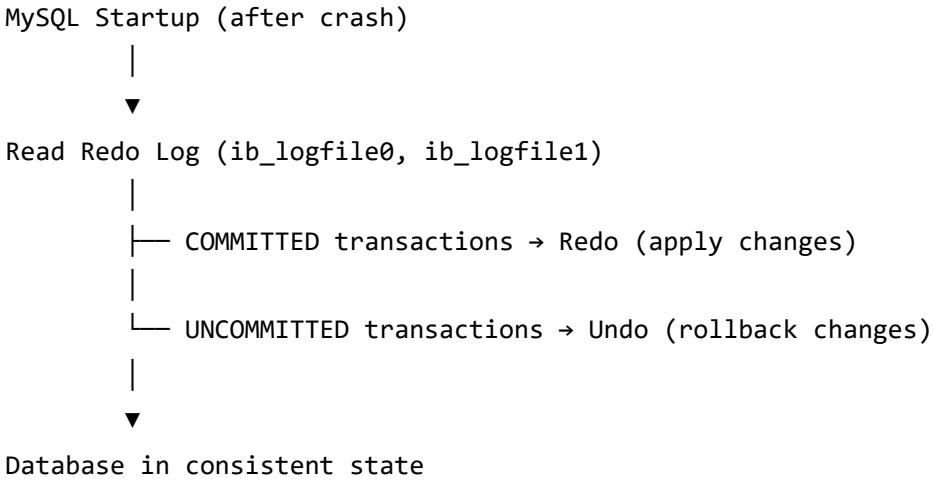
6.3 Optimistic Locking (Version Column)

```
-- Check version hasn't changed
UPDATE account_balances
SET available_balance = ?,
    version = version + 1
WHERE account_id = ?
  AND version = ?;  -- Expected version

-- If affected_rows = 0, concurrent modification detected
IF ROW_COUNT() = 0 THEN
    ROLLBACK;
    SET p_message = 'Concurrent modification detected';
END IF;
```

7. Recovery & Consistency Assurance

7.1 InnoDB Recovery Process



7.2 Point-in-Time Recovery

Backup Type	Method	Recovery Point
Full Backup	mysqldump or XtraBackup	Backup time
Binary Logs	MySQL binary log	Any point in time

7.3 Balance Verification

If corruption is suspected, balance can be recalculated:

```
-- Procedure: sp_rebuild_balance
SELECT
    SUM(CASE WHEN entry_type = 'CREDIT' THEN amount ELSE 0 END) -
    SUM(CASE WHEN entry_type = 'DEBIT' THEN amount ELSE 0 END)
INTO v_calculated_balance
FROM ledger_entries
WHERE account_id = p_account_id;

-- Compare with cached balance
SELECT available_balance INTO v_cached_balance
FROM account_balances WHERE account_id = p_account_id;

IF v_calculated_balance != v_cached_balance THEN
    -- Data integrity issue detected!
    -- Log and alert, then recalculate
    UPDATE account_balances
    SET available_balance = v_calculated_balance
    WHERE account_id = p_account_id;
END IF;
```

8. DBMS-Level Enforcement Summary

Feature	MySQL Implementation	Banking Core Usage
Transactions	`START TRANSACTION` , `COMMIT` , `ROLLBACK`	All financial procedures
Row Locking	`SELECT ... FOR UPDATE`	Account balance access
Isolation Level	REPEATABLE READ (default)	Used as-is
Durability	InnoDB redo log + fsync	Default settings
Error Handling	`DECLARE EXIT HANDLER`	All procedures
Atomic Writes	InnoDB WAL	Automatic

9. Conclusion

9.1 ACID Compliance Summary

Property	Implementation	Verification
Atomicity	Stored procedures with explicit COMMIT/ROLLBACK	Crash mid-transaction leaves no trace
Consistency	Validation before modification, double-entry rule	DEBIT sum always equals CREDIT sum
Isolation	FOR UPDATE row locks + REPEATABLE READ	Concurrent withdrawals safely serialized
Durability	InnoDB redo log with fsync on commit	Survives power failure post-commit

9.2 Key Design Decisions

1. **Stored Procedures** – Transaction boundary in database, not application
2. **Pessimistic Locking** – FOR UPDATE prevents race conditions
3. **Idempotency Keys** – Prevents duplicate transactions on retry
4. **Append-Only Ledger** – Ledger entries never modified, only appended
5. **Cached Balances** – Denormalized for O(1) lookup, verified against ledger

9.3 Banking Requirements Met

Requirement	How Met
Money never lost	Atomicity + Durability
Accurate balances	Consistency + Isolation
Full audit trail	Append-only ledger + audit tables
Concurrent access	Row-level locking + MVCC
Regulatory compliance	Complete transaction history preserved

This case study is suitable for DBMS coursework, viva examinations, and academic project documentation.