Banking Core - SQL CRUD & Query Mapping

Database Query Reference for DBMS Course

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

1. Introduction to CRUD in Banking Core

1.1 What is CRUD?

CRUD represents the four basic database operations:

Operation	SQL Command	Banking Example
Create	INSERT	Open new account
Read	SELECT	View balance
Update	UPDATE	Change password
Delete	DELETE	Not used (soft delete)

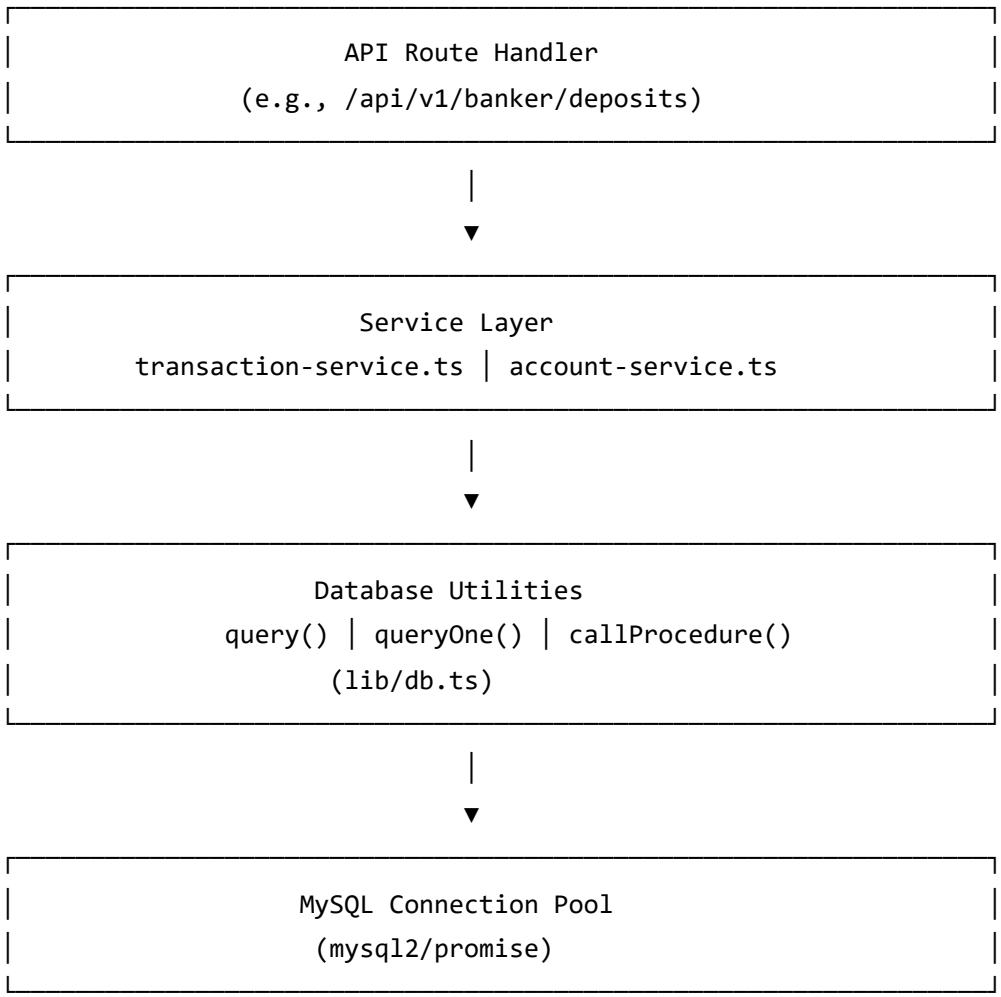
1.2 Banking Core CRUD Characteristics

Characteristic	Implementation
No Hard Deletes	Status columns used for soft delete
Append-Only Ledger	Financial records never modified
Prepared Statements	All queries use parameterized inputs

Characteristic	Implementation
Transaction Wrapping	Write operations grouped in transactions

2. Database Access Architecture

2.1 Query Execution Flow



2.2 Query Helper Functions

```
// Simple query with parameters
await query<RowType>('SELECT * FROM users WHERE id = ?', [userId]);

// Single row query
await queryOne<RowType>('SELECT * FROM accounts WHERE id = ?', [id]);

// Stored procedure call
await callProcedure('sp_deposit', [accountId, amount, description, userId],
                    ['p_transaction_id', 'p_status', 'p_message']);
```

3. CRUD Classification by Module

3.1 Auth Module

Feature	Route	Operation	Table(s)	Query Type
User Login	POST /auth/login	READ	users, roles	SELECT + JOIN
Customer Login	POST /auth/login	READ	customers	SELECT
Update Last Login	POST /auth/login	UPDATE	users/customers	UPDATE
Logout	POST /auth/logout	CREATE	audit_logs	INSERT
Token Refresh	POST /auth/refresh	READ	refresh_tokens	SELECT

Sample Query - User Login:

```
SELECT u.id, u.email, u.password_hash, u.first_name, u.last_name,
       u.status, r.id as role_id, r.code as role_code,
       r.name as role_name, r.permissions
FROM users u
JOIN roles r ON u.role_id = r.id
WHERE u.email = ?
  AND u.status = 'ACTIVE'
```

3.2 Customer Module

Feature	Route	Operation	Table(s)	Query Type
Get Profile	GET /customer/profile	READ	customers	SELECT
Update Profile	PUT /customer/profile	UPDATE	customers	UPDATE
Change Password	POST /customer/profile/password	UPDATE	customers	UPDATE
View Statistics	GET /customer/stats	READ	accounts, transactions	SELECT + Aggregate

Sample Query - Get Profile:

```
SELECT id, customer_number, email, first_name, last_name,
       phone, address_line1, city, country, status, kyc_status
FROM customers
WHERE id = ?
```

Sample Query - Change Password:

```
UPDATE customers
SET password_hash = ?,
    updated_at = NOW()
WHERE id = ?
```

3.3 Banker Module

Feature	Route	Operation	Table(s)	Query Type
List Customers	GET /banker/customers	READ	customers	SELECT
Create Customer	POST /banker/customers/create	CREATE	customers, accounts,	INSERT (multi)

Feature	Route	Operation	Table(s)	Query Type
			account_balances	
Get Customer	GET /banker/customers/[id]	READ	customers, accounts	SELECT + JOIN
Update Customer	PUT /banker/customers/[id]	UPDATE	customers	UPDATE
List Accounts	GET /banker/accounts	READ	accounts, account_balances, customers	SELECT + JOIN
Open Account	POST /banker/accounts	CREATE	accounts, account_balances	INSERT
Close Account	POST /banker/accounts/[id]/close	UPDATE	accounts	UPDATE
Freeze Account	POST /banker/accounts/[id]/freeze	UPDATE	accounts	UPDATE
Unfreeze	POST /banker/accounts/[id]/unfreeze	UPDATE	accounts	UPDATE
Deposit	POST /banker/deposits	CREATE	Via sp_deposit	Stored Procedure
Withdrawal	POST /banker/withdrawals	CREATE	Via sp_withdraw	Stored Procedure
View Ledger	GET /banker/ledger	READ	ledger_entries, accounts	SELECT + JOIN

Sample Query - List Customers with Pagination:

```
SELECT id, customer_number, email, first_name, last_name,
       phone, status, kyc_status, created_at
FROM customers
WHERE status != 'DELETED'
ORDER BY created_at DESC
LIMIT ? OFFSET ?
```

Sample Query - Create Customer:

```
INSERT INTO customers (
    customer_number, email, password_hash, first_name, last_name,
    phone, national_id, address_line1, city, country,
    status, kyc_status, created_by, created_at
) VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?, ?, 'PENDING', 'PENDING', ?, NOW())
```

Sample Query - Open Account:

```
-- Step 1: Insert account
INSERT INTO accounts (
    account_number, customer_id, account_type_id,
    status, created_by, opened_at
) VALUES (?, ?, ?, 'ACTIVE', ?, NOW());

-- Step 2: Create balance record
INSERT INTO account_balances (
    account_id, available_balance, pending_balance,
    hold_balance, currency, version
) VALUES (LAST_INSERT_ID(), 0, 0, 0, 'BDT', 1);
```

3.4 Accounts Module

Feature	Route	Operation	Table(s)	Query Type
List Accounts	GET /accounts	READ	accounts, account_balances, account_types	SELECT + JOIN
Get Account	GET /accounts/[id]	READ	accounts, account_balances	SELECT + JOIN
Get Statement	GET /accounts/[id]/statement	READ	transactions, ledger_entries	SELECT + JOIN
Export PDF	GET /accounts/[id]/statement/pdf	READ	Same as above	SELECT + JOIN

Sample Query - List Customer's Accounts:

```
SELECT a.id, a.account_number, a.status, a.currency,
       at.code as account_type, at.name as account_type_name,
       ab.available_balance, ab.pending_balance, ab.hold_balance
FROM accounts a
JOIN account_types at ON a.account_type_id = at.id
JOIN account_balances ab ON a.id = ab.account_id
WHERE a.customer_id = ?
  AND a.status != 'CLOSED'
ORDER BY a.created_at DESC
```

Sample Query - Account Statement:

```
SELECT t.id, t.transaction_reference, t.amount, t.currency,
       t.description, t.status, t.processed_at,
       tt.code as transaction_type,
       le.entry_type, le.balance_after
FROM transactions t
JOIN transaction_types tt ON t.transaction_type_id = tt.id
JOIN ledger_entries le ON t.id = le.transaction_id
WHERE le.account_id = ?
  AND le.entry_date BETWEEN ? AND ?
ORDER BY t.processed_at DESC
LIMIT ? OFFSET ?
```

3.5 Transactions Module

Feature	Route	Operation	Table(s)	Query Type
Transfer	POST /transactions/transfer	CREATE	Via sp_transfer	Stored Procedure
Get Transaction	GET /transactions/[id]	READ	transactions, ledger_entries	SELECT + JOIN
Search	GET /transactions/search	READ	transactions, accounts	SELECT + JOIN

Sample Query - Get Transaction Details:

```
SELECT t.id, t.transaction_reference, t.amount, t.currency,
       t.description, t.status, t.processed_at,
       tt.code as type_code, tt.name as type_name,
       sa.account_number as source_account,
       da.account_number as destination_account,
       CONCAT(sc.first_name, ' ', sc.last_name) as source_owner,
       CONCAT(dc.first_name, ' ', dc.last_name) as destination_owner
FROM transactions t
JOIN transaction_types tt ON t.transaction_type_id = tt.id
LEFT JOIN accounts sa ON t.source_account_id = sa.id
LEFT JOIN accounts da ON t.destination_account_id = da.id
LEFT JOIN customers sc ON sa.customer_id = sc.id
LEFT JOIN customers dc ON da.customer_id = dc.id
WHERE t.id = ?
```

3.6 Auditor Module

Feature	Route	Operation	Table(s)	Query Type
Dashboard Summary	GET /auditor/summary	READ	Multiple	Aggregate queries
All Transactions	GET /auditor/transactions	READ	transactions, accounts	SELECT + JOIN
All Ledger	GET /auditor/ledger	READ	ledger_entries, accounts	SELECT + JOIN
Audit Logs	GET /auditor/audit-logs	READ	audit_logs	SELECT
All Customers	GET /auditor/customers	READ	customers	SELECT
All Accounts	GET /auditor/accounts	READ	accounts, account_balances	SELECT + JOIN

Feature	Route	Operation	Table(s)	Query Type
Export PDFs	GET /auditor/export-pdf/*	READ	Various	SELECT

Sample Query - Dashboard Summary (Aggregations):

```
-- Total customers
SELECT COUNT(*) as total FROM customers WHERE status = 'ACTIVE';

-- Total accounts
SELECT COUNT(*) as total FROM accounts WHERE status = 'ACTIVE';

-- Total transactions today
SELECT COUNT(*) as total, SUM(amount) as volume
FROM transactions
WHERE DATE(processed_at) = CURDATE() AND status = 'COMPLETED';

-- Transaction breakdown by type
SELECT tt.code, tt.name, COUNT(*) as count, SUM(t.amount) as total
FROM transactions t
JOIN transaction_types tt ON t.transaction_type_id = tt.id
WHERE t.status = 'COMPLETED'
GROUP BY tt.id;
```

Sample Query - Audit Logs with Filtering:

```
SELECT id, actor_id, actor_type, actor_role, action_type,
       entity_type, entity_id, before_state, after_state,
       metadata, created_at
FROM audit_logs
WHERE (:actionType IS NULL OR action_type = :actionType)
  AND (:entityType IS NULL OR entity_type = :entityType)
  AND created_at BETWEEN :fromDate AND :toDate
ORDER BY created_at DESC
LIMIT ? OFFSET ?
```

3.7 Admin Module

Feature	Route	Operation	Table(s)	Query Type
List Users	GET /admin/users	READ	users, roles	SELECT + JOIN
Create User	POST /admin/users	CREATE	users	INSERT
Update User	PUT /admin/users/[id]	UPDATE	users	UPDATE
List Roles	GET /admin/roles	READ	roles	SELECT
Get Config	GET /admin/config	READ	system_config	SELECT
Update Config	PUT /admin/config	UPDATE	system_config	UPDATE
Run EOD	POST /admin/eod/run	CREATE	daily_account_totals	INSERT
Post Interest	POST /admin/interest/post	CREATE	Via procedure	Stored Procedure

Sample Query - List Users:

```
SELECT u.id, u.email, u.first_name, u.last_name, u.status,
       u.last_login_at, u.created_at,
       r.id as role_id, r.code as role_code, r.name as role_name
FROM users u
JOIN roles r ON u.role_id = r.id
ORDER BY u.created_at DESC
```

4. Table-wise CRUD Summary

Table	Create	Read	Update	Delete	Notes
roles	Admin only	Auth, Admin	Admin	Never	Reference table

Table	Create	Read	Update	Delete	Notes
users	Admin	Auth, Admin	Admin, Self	Never	Soft delete via status
customers	Banker	Many	Banker, Self	Never	Soft delete via status
account_types	Seed only	Banker	Admin	Never	Reference table
accounts	Banker	Many	Banker (status)	Never	Soft delete via status
account_balances	With account	Many	Via procedure	Never	Always via sp_*
transaction_types	Seed only	Service	Never	Never	Reference table
transactions	Via procedure	Many	Never	Never	Append-only
ledger_entries	Via procedure	Many	Never	Never	Append-only
transaction_audit	Via trigger	Auditor	Never	Never	Append-only
audit_logs	Service layer	Auditor	Never	Never	Append-only
system_config	Admin	Admin	Admin	Never	Key-value store
idempotency_keys	Via procedure	Via procedure	Never	Expired cleanup	TTL-based
events	Via procedure	Internal	Never	Never	Event store
outbox	Via procedure	Internal	Published	Never	Transactional outbox

5. Complex Queries and Joins

5.1 Join Strategies Used

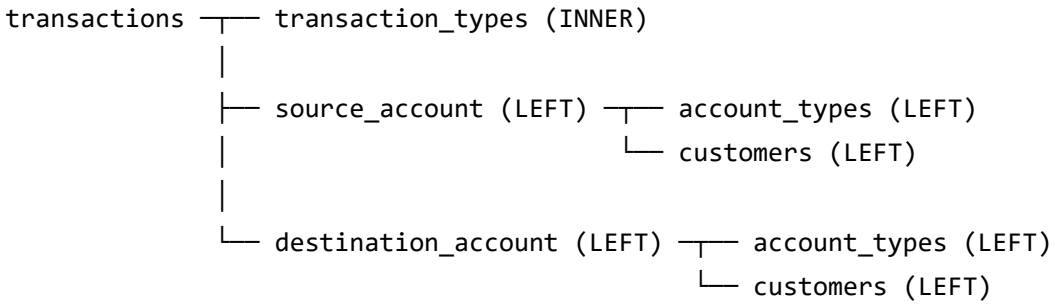
Join Type	Usage Example
INNER JOIN	users + roles (user must have role)
LEFT JOIN	transactions + accounts (source may be null for deposits)
Self-referential	Not used
Cross Join	Not used

5.2 Multi-Table Join Example

Transaction Details with Full Context:

```sql
SELECT
    t.id,
    t.transaction_reference,
    t.amount,
    t.currency,
    t.status,
    t.processed_at,

    -- Transaction type
    tt.code as type_code,
    tt.name as type_name,

    -- Source account details
    sa.account_number as source_account,
    sat.name as source_account_type,
    CONCAT(sc.first_name, ' ', sc.last_name) as source_customer,

    -- Destination account details
    da.account_number as destination_account,
    dat.name as destination_account_type,
    CONCAT(dc.first_name, ' ', dc.last_name) as destination_customer,

    -- Initiator
    CASE
        WHEN t.created_by IN (SELECT id FROM users)
        THEN (SELECT CONCAT(first_name, ' ', last_name) FROM users WHERE id = t.created_by)
        ELSE 'Customer'
    END as initiated_by

FROM transactions t
INNER JOIN transaction_types tt ON t.transaction_type_id = tt.id
LEFT JOIN accounts sa ON t.source_account_id = sa.id
LEFT JOIN account_types sat ON sa.account_type_id = sat.id
LEFT JOIN customers sc ON sa.customer_id = sc.id
LEFT JOIN accounts da ON t.destination_account_id = da.id
LEFT JOIN account_types dat ON da.account_type_id = dat.id
LEFT JOIN customers dc ON da.customer_id = dc.id
WHERE t.id = ?
```

Join Diagram:



6. Transaction-Scoped Queries

6.1 Stored Procedure: sp_deposit

```
CALL sp_deposit(
    @account_id,      -- Target account
    @amount,          -- Deposit amount
    @description,     -- Description
    @banker_id,       -- Who performed
    @out_tx_id,       -- OUT: Transaction ID
    @out_status,      -- OUT: COMPLETED/FAILED
    @out_message      -- OUT: Result message
);
```

Internal SQL Sequence:

```sql
START TRANSACTION;

-- 1. Lock bank cash account
SELECT available_balance INTO @cash_balance
FROM account_balances WHERE account_id = @bank_cash_id FOR UPDATE;

-- 2. Lock customer account
SELECT available_balance, status INTO @cust_balance, @cust_status
FROM account_balances ab JOIN accounts a ON ab.account_id = a.id
WHERE ab.account_id = @account_id FOR UPDATE;

-- 3. Validate
IF @cust_status != 'ACTIVE' THEN ROLLBACK; END IF;

-- 4. Insert transaction
INSERT INTO transactions (...) VALUES (...);
SET @tx_id = LAST_INSERT_ID();

-- 5. Insert ledger entries
INSERT INTO ledger_entries (transaction_id, account_id, entry_type, amount, balance_after)
VALUES (@tx_id, @bank_cash_id, 'DEBIT', @amount, @cash_balance - @amount);

INSERT INTO ledger_entries (transaction_id, account_id, entry_type, amount, balance_after)
VALUES (@tx_id, @account_id, 'CREDIT', @amount, @cust_balance + @amount);

-- 6. Update balances
UPDATE account_balances SET available_balance = available_balance - @amount WHERE account_id = @
UPDATE account_balances SET available_balance = available_balance + @amount WHERE account_id = @

-- 7. Insert events
INSERT INTO events (...) VALUES (...);
INSERT INTO outbox (...) VALUES (...);

COMMIT;
```

7. Aggregation Queries

7.1 Dashboard Statistics

```sql
-- Daily transaction summary
SELECT
    DATE(processed_at) as date,
    COUNT(*) as transaction_count,
    SUM(amount) as total_volume,
    AVG(amount) as average_amount
FROM transactions
WHERE status = 'COMPLETED'
  AND processed_at >= DATE_SUB(NOW(), INTERVAL 30 DAY)
GROUP BY DATE(processed_at)
ORDER BY date DESC;

-- Top accounts by balance
SELECT
    a.account_number,
    CONCAT(c.first_name, ' ', c.last_name) as customer_name,
    ab.available_balance
FROM accounts a
JOIN customers c ON a.customer_id = c.id
JOIN account_balances ab ON a.id = ab.account_id
WHERE a.status = 'ACTIVE'
ORDER BY ab.available_balance DESC
LIMIT 10;

-- Transaction type distribution
SELECT
    tt.name,
    COUNT(*) as count,
    SUM(t.amount) as total,
    ROUND(COUNT(*) * 100.0 / (SELECT COUNT(*) FROM transactions WHERE status = 'COMPLETED'), 2)
FROM transactions t
JOIN transaction_types tt ON t.transaction_type_id = tt.id
WHERE t.status = 'COMPLETED'
GROUP BY tt.id
ORDER BY count DESC;
```

8. Performance & Indexing Considerations

8.1 Index Strategy

Table	Index	Columns	Purpose
users	PRIMARY	id	PK lookup
users	UNIQUE	email	Login lookup
customers	PRIMARY	id	PK lookup
customers	UNIQUE	email	Login lookup
customers	UNIQUE	customer_number	Business lookup
accounts	PRIMARY	id	PK lookup
accounts	UNIQUE	account_number	Transfer lookup
accounts	INDEX	customer_id	List customer's accounts
ledger_entries	PRIMARY	id	PK lookup
ledger_entries	INDEX	transaction_id	Transaction details
ledger_entries	INDEX	account_id, entry_date	Statement queries
transactions	PRIMARY	id	PK lookup
transactions	UNIQUE	transaction_reference	Idempotency
transactions	INDEX	source_account_id	Account history
transactions	INDEX	destination_account_id	Account history
transactions	INDEX	processed_at	Date filtering
audit_logs	INDEX	actor_id, actor_type	Who did what
audit_logs	INDEX	entity_type, entity_id	Entity history
audit_logs	INDEX	action_type	Action filtering
audit_logs	INDEX	created_at	Time filtering

8.2 Query Optimization Patterns

Pattern	Example	Benefit
Limit + Offset	`LIMIT 25 OFFSET 50`	Pagination without loading all
Index-only scan	`SELECT id, email FROM users WHERE email = ?`	Covered by index
Prepared statements	`WHERE id = ?`	Query plan caching
Selective columns	`SELECT id, name` not `SELECT *`	Reduced I/O

9. Prepared Statement Security

9.1 SQL Injection Prevention

❌ **Vulnerable (NOT used in Banking Core):**

```
const query = `SELECT * FROM users WHERE email = '${email}'`;
```

✅ **Safe (Used in Banking Core):**

```
await query('SELECT * FROM users WHERE email = ?', [email]);
```

9.2 Parameter Binding Flow

```
User Input: "admin@bank.com"
        │
        ▼
Prepared Statement: "SELECT * FROM users WHERE email = ?"
        │
        ▼
Parameter Binding: ['admin@bank.com'] → Escaped/typed value
        │
        ▼
Execution: No SQL injection possible
```

10. Conclusion

10.1 CRUD Summary Statistics

Operation	Count	Primary Use
SELECT	45+ queries	Data retrieval, validation
INSERT	15+ queries	Customer, account, transaction creation
UPDATE	10+ queries	Status changes, password updates
DELETE	0 queries	Not used (soft delete pattern)

10.2 Key Design Patterns

1. **Soft Delete** – Status columns instead of DELETE statements
2. **Prepared Statements** – All queries parameterized
3. **Stored Procedures** – Financial operations atomic
4. **Append-Only Ledger** – No UPDATE/DELETE on financial tables
5. **Multi-Table Joins** – Rich data retrieval with single query

10.3 Query Design Principles

Principle	Implementation
Single Responsibility	One query per data need
Minimal Data Fetching	SELECT only needed columns
Index Awareness	Queries designed for index usage
Transaction Safety	Write operations in procedures

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