# Bonus Laboratory Work

*Advanced Database Programming Challenge*

**Maximum Points: 5 | Difficulty: Advanced**

## Overview

This bonus laboratory work requires you to demonstrate comprehensive understanding of advanced PostgreSQL concepts by building a complete banking transaction system. You will integrate **Transactions**, **Stored Procedures**, **Views**, and **Indexes** into a cohesive solution that handles real-world banking scenarios.

***Note:** This is an individual assignment. All work must be original. Plagiarism will result in zero points.*

## Business Scenario

You are developing the core transaction processing module for **KazFinance Bank**, a digital bank operating in Kazakhstan. The system must handle multi-currency accounts (KZT, USD, EUR, RUB), support instant transfers between customers, enforce daily transaction limits, maintain complete audit trails, and provide real-time reporting for regulatory compliance.

The system must be designed to handle high concurrency (thousands of simultaneous transactions) while maintaining data integrity and ACID compliance.

## Database Schema

Create the following tables as the foundation for your solution:

| Table | Columns & Description |
|---|---|
| **customers** | customer_id (PK), iin (unique, 12 digits), full_name, phone, email, status (active/blocked/frozen), created_at, daily_limit_kzt |
| **accounts** | account_id (PK), customer_id (FK), account_number (unique, IBAN format), currency (KZT/USD/EUR/RUB), balance, is_active, opened_at, closed_at |
| **transactions** | transaction_id (PK), from_account_id (FK), to_account_id (FK), amount, currency, exchange_rate, amount_kzt (converted), type (transfer/deposit/withdrawal), status (pending/completed/failed/reversed), created_at, completed_at, description |
| **exchange_rates** | rate_id (PK), from_currency, to_currency, rate, valid_from, valid_to |
| **audit_log** | log_id (PK), table_name, record_id, action (INSERT/UPDATE/DELETE), old_values (JSONB), new_values (JSONB), changed_by, changed_at, ip_address |

*Populate each table with at least 10 meaningful records for testing.*

## Tasks

### Task 1: Transaction Management

Create a stored procedure **process_transfer** that handles money transfers between accounts with full ACID compliance.

**Requirements:**

- Accept parameters: from_account_number, to_account_number, amount, currency, description
- Validate both accounts exist and are active

- Check that sender's customer status is 'active' (not blocked or frozen)
- Verify sufficient balance in source account
- Check daily transaction limit (sum of today's transactions + current transfer ≤ daily_limit_kzt)
- Handle currency conversion using current exchange rates when currencies differ
- Use SELECT ... FOR UPDATE to prevent race conditions
- Implement proper SAVEPOINT usage for partial rollback scenarios
- Return detailed error messages with error codes for each failure scenario
- Log all operations to audit_log table (including failed attempts)

## Task 2: Views for Reporting

Create the following views for regulatory reporting and analytics:

**View 1: customer_balance_summary**

- Show each customer with all their accounts and balances
- Include total balance converted to KZT using current exchange rates
- Show daily limit utilization percentage
- Use window functions to rank customers by total balance

**View 2: daily_transaction_report**

- Aggregate transactions by date and type
- Show total volume, count, average amount per category
- Include running totals using window functions
- Calculate day-over-day growth percentage

**View 3: suspicious_activity_view (WITH SECURITY BARRIER)**

- Flag transactions over 5,000,000 KZT equivalent
- Identify customers with >10 transactions in a single hour
- Detect rapid sequential transfers (same sender, <1 minute apart)
- Must use SECURITY BARRIER to prevent information leakage

## Task 3: Performance Optimization with Indexes

Design and implement an optimal indexing strategy:

- Create at least 5 different types of indexes (B-tree, Hash, GIN, partial, composite)
- Justify each index choice with EXPLAIN ANALYZE output
- Create a covering index for the most frequent query pattern
- Implement a partial index for active accounts only
- Create an expression index for case-insensitive email search
- Add a GIN index on audit_log JSONB columns
- Document the performance improvement (before/after comparison)

## Task 4: Advanced Procedure - Batch Processing

Create a stored procedure **process_salary_batch** for processing monthly salary payments for a company:

**Requirements:**

- Accept parameters: company_account_number, JSONB array of payments [{iin, amount, description}, ...]
- Validate total batch amount against company account balance before starting
- Process each payment individually within a single transaction
- Use SAVEPOINT to allow partial batch completion (continue on individual failures)
- Return detailed results: successful_count, failed_count, failed_details (JSONB array)

- Implement advisory locks to prevent concurrent batch processing for same company
- All individual transfers must bypass daily limits (salary exception)
- Update all balances atomically at the end (not one-by-one)
- Generate a summary report viewable through a materialized view

## Submission Requirements

1. Single SQL file with all DDL, DML, and procedural code
2. Test cases demonstrating each scenario (successful and failed operations)
3. EXPLAIN ANALYZE outputs for all created indexes
4. Brief documentation explaining design decisions
5. Demonstration of concurrent transaction handling (using two psql sessions)

## Grading Criteria

| Criterion | Points | Weight |
|---|---|---|
| Correct transaction isolation and ACID compliance | 1.5 | 30% |
| View complexity and correct use of window functions | 1.0 | 20% |
| Index strategy effectiveness and documentation | 1.0 | 20% |
| Batch processing with proper error handling | 1.5 | 30% |
| **TOTAL** | **5.0** | **100%** |

## Helpful Tips

- Use pg_advisory_lock() for batch processing synchronization
- Remember that SELECT FOR UPDATE acquires row-level locks
- JSONB operators: ->, ->>, @>, ?, jsonb_array_elements()
- For SECURITY BARRIER views: CREATE VIEW ... WITH (security_barrier = true)
- Window functions: ROW_NUMBER(), RANK(), LAG(), LEAD(), SUM() OVER()
- Test concurrency with: BEGIN; SELECT ... FOR UPDATE; -- wait in another session
- Use RAISE EXCEPTION with custom SQLSTATE codes for error handling

*Good luck! This is your opportunity to demonstrate mastery of advanced PostgreSQL concepts.*