

Team 15 — University Campus Management System

COSC 3380: Database Systems — HW2 Phase 2 Report

Biniam Habte Rohan Kancherla Ahmad Kaseb Ali Ismail

November 15, 2025

1 Introduction

This project implements a comprehensive University Campus Management System aligned with all Phase 2 requirements. It combines a normalized PostgreSQL database, a Node.js/Express backend, and a JavaScript/HTML frontend. The system provides ACID-compliant transactions for tuition payments and grade posting, analytical reporting, and concurrency testing.

Over 90% of business logic is executed inside SQL. JavaScript is used only to validate input, build requests, dispatch SQL to PostgreSQL, and render results in the browser. All state-changing operations (tuition payments, grade posting, wallet updates, and charge updates) are executed inside database transactions.

2 ER Design and Modeling

The final ER model focuses on two main domains:

- **Academic:** divisions, halls, terms, units, tutors, students, offerings, timeslots, and enrollments.
- **Financial:** wallets, charges, receipts, payment kinds, and a grade audit table.

Entities and relationships are mapped with primary keys, foreign keys, and composite keys that enforce realistic business constraints (e.g., one wallet per student, one charge per student-term, one enrollment per student-offering).

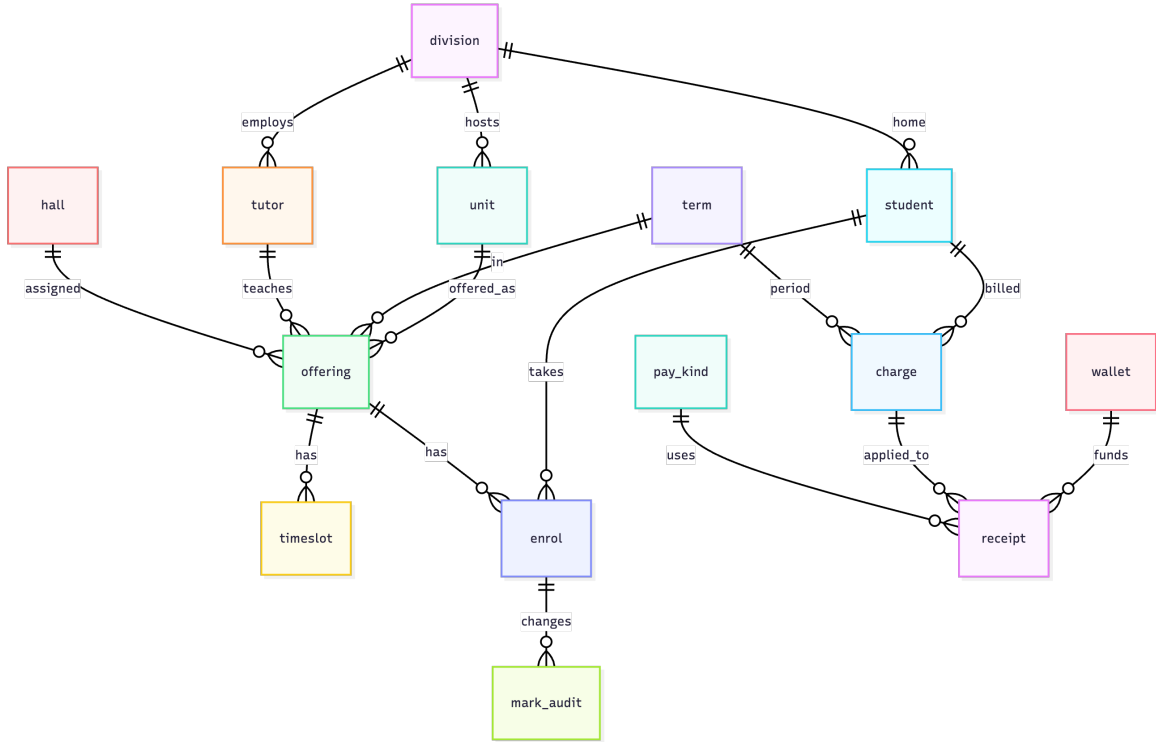


Figure 1: Final UML ER diagram for the Team 15 campus management system.

3 Normalization (Table-by-Table)

Each table was normalized to 3NF or BCNF.

Lookup Tables (BCNF)

division, hall, term, pay_kind

All non-key attributes fully depend on the primary key; there are no partial or transitive dependencies. Examples:

- `division(div_id, title, office)` — `title` and `office` depend only on `div_id`.
- `hall(hall_id, bldg, room, seats)` — `seats` depends only on `hall_id`. A uniqueness constraint on `(bldg, room)` prevents duplicates.
- `term(term_id, code, start_on, end_on, status)` — all attributes describe the term only.
- `pay_kind(kind_code, label)` — `label` depends only on `kind_code`.

Academic Tables

unit, tutor, student

These tables are in BCNF: all attributes describe the entity identified by the primary key, with no derived attributes.

offering

`offering(offering_id, unit_id, term_id, tutor_id, hall_id, cap, mode)`

The primary key is `offering_id`. Capacity and mode depend only on `offering_id`. Foreign keys link to unit, term, tutor, and hall. A uniqueness constraint on (`unit_id`, `term_id`, `tutor_id`, `hall_id`) avoids duplicate sections.

timeslot

`timeslot(offering_id, dow, tstart, tend)`

Composite primary key (`offering_id`, `dow`, `tstart`). All attributes (`dow`, `tstart`, `tend`) depend on the full key, and a CHECK enforces `tstart < tend`. This table is in 3NF.

Transaction Tables

enrol

`enrol(student_id, offering_id, enrolled_at, estatus, grade)`

Composite primary key (`student_id`, `offering_id`). Grade and status depend on the full key, not on a subset, so the table is in 3NF.

charge

`charge(student_id, term_id, due_amt, paid_amt, cstatus, made_at)`

Composite PK (`student_id`, `term_id`). Monetary amounts and status depend on the full key. The table is in 3NF with one intentional denormalization:

- **Intentional 3NF violation:** `cstatus` (UNPAID/PARTIAL/PAID) is a cached value derived from `due_amt` and `paid_amt`. A trigger maintains it to speed up queries and simplify WHERE clauses.

wallet

`wallet(wallet_id, owner_type, owner_id, balance)`

There is a unique constraint on (`owner_type`, `owner_id`), guaranteeing at most one wallet per owner. All attributes depend on `wallet_id`, so the table is in BCNF.

receipt

`receipt(receipt_id, student_id, term_id, wallet_id, kind_code, amount, paid_at)`

Every attribute describes a single receipt. Foreign keys enforce the connection to charge, wallet, and pay_kind. This table is in BCNF.

mark_audit

`mark_audit(audit_id, student_id, offering_id, old_grade, new_grade, by_tutor, at_time)`

This table records grade history; all attributes describe the audit row. It is in BCNF.

4 ER to Relational Mapping

Key mappings:

- Strong entities (student, tutor, unit, term, hall, division, wallet, receipt) map directly to tables with surrogate primary keys.
- The many-to-many relationship between students and offerings becomes the `enrol(student_id, offering_id)` table with a composite primary key.
- A multi-valued attribute (schedule) on offering is represented by the `timeslot` table.
- One-to-many relationships such as `unit → offering` and `term → charge` are enforced via foreign keys.
- Financial flows are captured by charge (per student-term bill), wallet (for balances), and receipt (per payment) with foreign keys to keep everything consistent.

All referential integrity is enforced by PostgreSQL foreign keys, and cascading deletes are used where appropriate (e.g., timeslots for a deleted offering).

5 Physical Model: Indexes and Trigger

Indexes

We added indexes that match the main transaction and reporting queries:

- `ix_offering_term` on `offering(term_id)`.
- `ix_offering_unit` on `offering(unit_id)`.
- `ix_enrol_student` on `enrol(student_id)`.
- `ix_enrol_offering` on `enrol(offering_id)`.
- `ix_receipt_student` on `receipt(student_id, term_id)`.
- `ix_charge_status` on `charge(cstatus)` for fast UNPAID/PARTIAL filters.

Trigger: Charge Status Maintenance

The trigger keeps `charge.cstatus` consistent with `due_amt` and `paid_amt`:

```
CREATE OR REPLACE FUNCTION fx_charge_status()
RETURNS TRIGGER AS $$
BEGIN
    IF NEW.paid_amt >= NEW.due_amt THEN
        NEW.cstatus := 'PAID';
    ELSIF NEW.paid_amt > 0 THEN
        NEW.cstatus := 'PARTIAL';
    ELSE
        NEW.cstatus := 'UNPAID';
    END IF;
    RETURN NEW;
END; $$ LANGUAGE plpgsql;
```

6 DB-Side Processing Justification

The design intentionally pushes nearly all logic into SQL:

- **Transactions** (tuition payments and grade posting) are fully written in SQL, updating multiple tables under BEGIN/COMMIT with row-level locking.
- **Reporting** (fill rate, billing summaries, balances) is implemented using joins and GROUP BY directly in SQL.
- **Derived values** such as `cstatus` are handled via triggers, not JavaScript.
- **Simulation and concurrency** rely on repeated calls to SQL transactions, not manual looping over rows in JS.

JavaScript never walks over full table contents to “simulate” database logic. It only sends parameterized requests to the server, which then runs SQL inside PostgreSQL. This satisfies the requirement that at least 90% of processing is done by the DBMS.

7 Main SQL Queries (query.sql)

Offering Fill Rate

```
SELECT o.offering_id, u.code AS unit_code, t.code AS term_code,
       COUNT(e.student_id) AS enrolled, o.cap,
       FLOOR(100.0 * COUNT(e.student_id) / o.cap) AS pct_full
```

```

FROM offering o
JOIN unit u ON u.unit_id = o.unit_id
JOIN term t ON t.term_id = o.term_id
LEFT JOIN enrol e
    ON e.offering_id = o.offering_id
    AND e.estatus = 'ENROLLED'
GROUP BY o.offering_id, u.code, t.code, o.cap
ORDER BY pct_full DESC, o.offering_id;

```

Term Billing Summary

```

SELECT t.code,
       COUNT(c.student_id) AS billed,
       SUM(c.due_amt) AS total_due,
       SUM(c.paid_amt) AS total_paid,
       SUM(c.due_amt - c.paid_amt) AS outstanding
FROM charge c
JOIN term t ON t.term_id = c.term_id
GROUP BY t.code
ORDER BY t.code;

```

Student Balances for a Term

```

SELECT s.student_id, s.firstn, s.lastn,
       c.due_amt, c.paid_amt,
       (c.due_amt - c.paid_amt) AS balance
FROM charge c
JOIN student s ON s.student_id = c.student_id
JOIN term t ON t.term_id = c.term_id
WHERE t.code = :p_term
ORDER BY balance DESC, s.lastn;

```

8 Main Transactions (transaction.sql)

Tuition Payment Transaction

```

BEGIN;

WITH sel_term AS (
    SELECT term_id
    FROM term

```

```

    WHERE code = :p_term_code
    FOR UPDATE
), lock_charge AS (
    SELECT c.*
    FROM charge c
    JOIN sel_term st ON st.term_id = c.term_id
    WHERE c.student_id = :p_student_id
    FOR UPDATE
)
INSERT INTO receipt(student_id, term_id, wallet_id, kind_code,
    amount)
VALUES (
    :p_student_id,
    (SELECT term_id FROM term WHERE code = :p_term_code),
    (SELECT wallet_id FROM wallet
        WHERE owner_type = CASE WHEN :p_kind = 'CASH'
                                THEN 'COMPANY' ELSE 'STUDENT' END
        AND (owner_id = :p_student_id OR owner_id IS NULL)),
    :p_kind,
    :p_amount
);

UPDATE charge
SET paid_amt = paid_amt + :p_amount
WHERE student_id = :p_student_id
    AND term_id = (SELECT term_id FROM term WHERE code = :p_term_code)
    ;

DO $$
DECLARE ws INT; wc INT;
BEGIN
    IF :p_kind <> 'CASH' THEN
        SELECT wallet_id INTO ws
        FROM wallet WHERE owner_type = 'STUDENT' AND owner_id = :
            p_student_id;

        SELECT wallet_id INTO wc
        FROM wallet WHERE owner_type = 'COMPANY' AND owner_id IS NULL;

        UPDATE wallet
        SET balance = balance - :p_amount
        WHERE wallet_id = ws AND balance >= :p_amount;

        UPDATE wallet
        SET balance = balance + :p_amount
        WHERE wallet_id = wc;
    END IF;

```

```
END $$;  
  
COMMIT;
```

Grade Posting Transaction

```
BEGIN;  
  
WITH prior AS (  
    SELECT grade  
    FROM enrol  
    WHERE student_id = :p_student_id  
        AND offering_id = :p_offering_id  
    FOR UPDATE  
)  
UPDATE enrol  
SET grade = :p_grade  
WHERE student_id = :p_student_id  
    AND offering_id = :p_offering_id;  
  
INSERT INTO mark_audit(student_id, offering_id, old_grade,  
                        new_grade, by_tutor)  
VALUES (  
    :p_student_id,  
    :p_offering_id,  
    (SELECT grade FROM prior),  
    :p_grade,  
    :p_tutor_id  
);  
  
COMMIT;
```

DML Policy (INSERT/UPDATE Only)

All modifications to core tables (`charge`, `wallet`, `enrol`, `receipt`, `mark_audit`) are performed using INSERT and UPDATE statements inside transactions. DELETE is intentionally avoided to keep history and simplify auditing, matching the homework guidelines.

9 GUI and Application Behavior

The application consists of a single-page HTML interface styled with CSS and driven by vanilla JavaScript. Buttons on the left select operations; results and forms appear in the main panel.

Front Desk (Student Interface)

- **Pay Tuition:** opens a form for student ID, term code, payment kind (CARD/ACH/-CASH), and amount. Submits to `/api/txn/pay-tuition`.
- **View Grades:** prompts for student ID and calls `/api/grades/:id`, displaying results in a dynamic table.
- **Check Wallet Balance:** returns the current student wallet balance.
- **Browse Tables:** allows selection of a table (students, offerings, enrolments, charges, etc.) and displays up to 10 rows.

Back Office (Admin Interface)

- **Create Tables:** calls `/api/create-tables` to execute `create_tables.sql`. In the final UI, this button can be guarded with a confirmation prompt because it recreates the schema.
- **Initialize Lookups:** truncates and reseeds data via `/api/seed-lookups`, which runs `lookup_seed.sql`.
- **Simulation (Auto-populate):** inserts 20 randomized payments using the same SQL payment transaction, exercising realistic workloads.
- **Concurrent Test (2 Simulations):** launches two simulations in parallel to stress-test locking.
- **Post Grades:** allows a tutor to change a student's grade for an offering, writing a row into `mark_audit`.
- **Reports:** runs the main reporting query and renders the table of results.

The UI also exposes links to the README and the demo video, so TAs can quickly read setup instructions and watch the walkthrough directly from the app.

10 Initialization Workflow

To make grading easy and repeatable, the system follows this workflow:

1. Click **Create Tables** to create the `campus` schema and all tables and indexes.
2. Click **Initialize Lookups** to populate divisions, halls, terms, pay kinds, units, tutors, students, wallets, offerings, timeslots, enrollments, and charges.
3. Use **Simulation** or **Concurrent Test** to generate payment traffic.
4. Use **Browse Tables** and **Reports** to verify that data is being updated correctly.

This matches the homework requirement that lookup and transaction tables can be initialized and reset via GUI buttons instead of manual SQL.

11 Concurrency Testing

Concurrency is tested by running two full simulations in parallel. Each simulation triggers 20 tuition-payment transactions using randomized student IDs, payment methods, and amounts. In the frontend, this is implemented with `Promise.all()` calling `/api/simulate` twice.

On the database side, row-level locking with `FOR UPDATE` ensures:

- No dirty writes on `charge` rows.
- Wallet balances are updated atomically.
- Charge and wallet records always match after concurrent runs.

Server logs show per-transaction durations and affected row counts, providing evidence that the system remains consistent under concurrent access.

12 Testing and Screenshots

Testing covered:

- Tuition payment with sufficient and insufficient wallet balances.
- Grade posting for valid and invalid student-offering combinations.
- Simulation and concurrent test with multiple browser sessions.

- Correct trigger behavior on `charge.cstatus`.
- Repeatable initialization and reseeding.

Representative screenshots are included below.

The top screenshot shows the 'Pay Tuition' form with a success message: 'Lookup tables seeded successfully'. The form includes fields for Student ID (1), Term (2025FA), Payment Method (Credit Card), and Amount (500). A 'SUBMIT PAYMENT' button is at the bottom.

The bottom screenshot shows the 'Showing 10 rows from charge' message and a table of charges. The table has columns: STUDENT_ID, TERM_ID, DUE_AMT, PAID_AMT, CSTATUS, and MADE_AT.

STUDENT_ID	TERM_ID	DUE_AMT	PAID_AMT	CSTATUS	MADE_AT
1	2	2275.00	500.00	PARTIAL	2025-11-18T00:29:46.690Z
2	2	2275.00	0.00	UNPAID	2025-11-18T00:29:46.690Z
3	2	2600.00	0.00	UNPAID	2025-11-18T00:29:46.690Z
4	2	1950.00	0.00	UNPAID	2025-11-18T00:29:46.690Z
5	2	2275.00	0.00	UNPAID	2025-11-18T00:29:46.690Z

Figure 2: Pay Tuition front-desk form showing a successful payment and updated status.

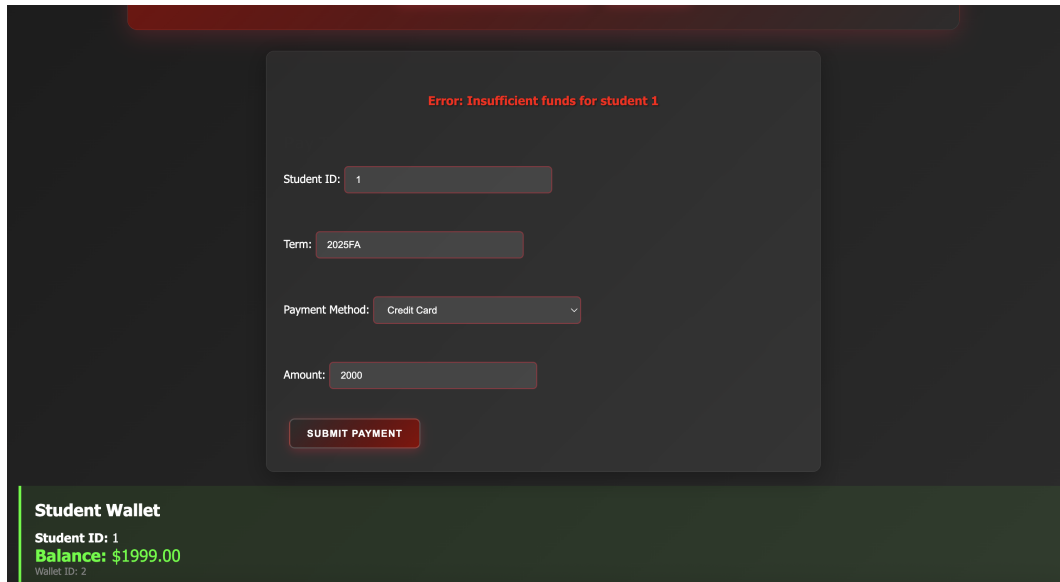


Figure 3: Tuition payment failure due to insufficient funds, demonstrating error handling and rollback.

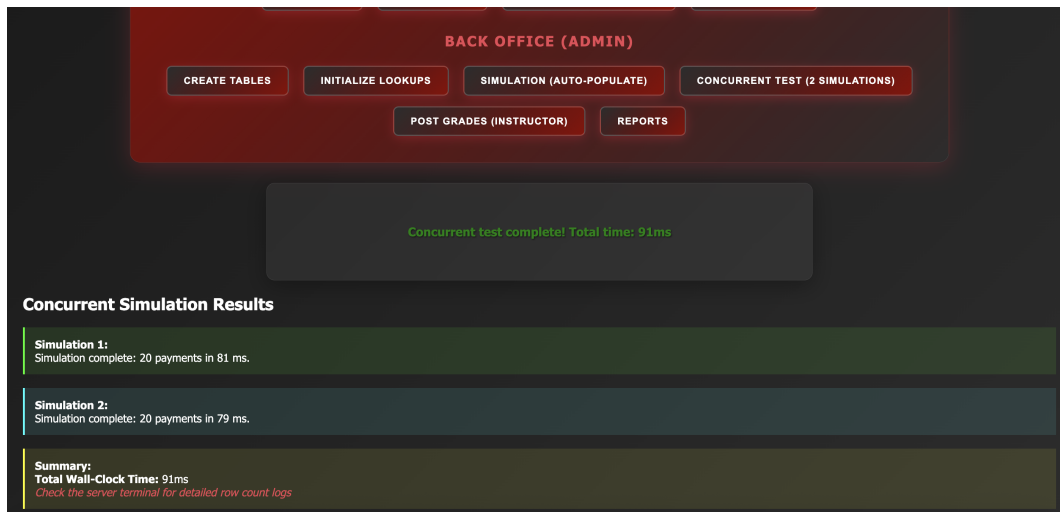


Figure 4: Concurrent simulation results with two parallel payment runs and wall-clock timing.

OFFERING_ID	UNIT_ID	TERM_ID	TUTOR_ID	HALL_ID	CAP	MODE
1	1	2	2	2	32	ONCAMP
2	2	2	3	3	32	BLEND
3	3	2	4	4	32	REMOTE
4	4	2	5	5	32	ONCAMP
5	5	2	6	6	32	BLEND
6	6	2	7	7	32	REMOTE
7	7	2	8	8	32	ONCAMP
8	8	2	9	9	32	BLEND
9	9	2	10	10	32	REMOTE
10	10	2	11	11	32	ONCAMP

Figure 5: Reporting screen showing an offering fill-rate report generated directly from SQL joins and aggregates.

13 Team Work, References, and Demo

Division of Work

- **Biniam Habte:** Frontend UI, rendering logic, integration.
- **Rohan Kancherla:** ER modeling, normalization, schema design.
- **Ahmad Kaseb:** ACID transaction design, triggers, concurrency.
- **Ali Ismail:** Reporting SQL, seed data, documentation.

References

PostgreSQL Documentation; Express.js Documentation; MDN Web Docs; COSC 3380 Lecture Slides; StackOverflow (syntax help); ChatGPT (language and structure assistance).

Demo Video Link

<https://www.youtube.com/watch?v=e0UhrTk1yPE>