LAB TEST – 04

Name : G.Sudeeksha

Roll no : 2403A510E0

Course : AI Assisted Coding

Batch : 05

Branch : CSE

SET –05

Q1. A student course registration system must enforce prerequisites and
seat limits.
a) Design schema including constraints and relations.
b) Write AI-assisted SQL to list students waiting for enrollment confirmation.

Prompt :

Generate SQL schema for a student course registration system with prerequisites and seat limits. Also write a query to list students waiting for enrollment confirmation.

Code :

registration.sql > ...

```sql
-- Course registration schema with prerequisites, seat limits, waitlist and helper function.
-- PostgreSQL SQL (single-file).
BEGIN;
-- Students
CREATE TABLE IF NOT EXISTS students (
  id          SERIAL PRIMARY KEY,
  student_no  TEXT NOT NULL UNIQUE,
  full_name   TEXT NOT NULL,
  email       TEXT UNIQUE,
  created_at  TIMESTAMP WITH TIME ZONE DEFAULT now()
);
-- Courses
CREATE TABLE IF NOT EXISTS courses (
  id     SERIAL PRIMARY KEY,
  code   TEXT NOT NULL UNIQUE,
  title  TEXT NOT NULL,
  credits INT NOT NULL DEFAULT 0
);
-- Course prerequisites (course -> prereq)
CREATE TABLE IF NOT EXISTS course_prereqs (
  course_id     INT NOT NULL REFERENCES courses(id) ON DELETE CASCADE,
  prereq_id     INT NOT NULL REFERENCES courses(id) ON DELETE CASCADE,
  PRIMARY KEY(course_id, prereq_id),
  CHECK (course_id <> prereq_id)
);
-- Sections (instances of a course with capacity)
CREATE TABLE IF NOT EXISTS sections (
  id             SERIAL PRIMARY KEY,
  course_id      INT NOT NULL REFERENCES courses(id) ON DELETE CASCADE,
  term           TEXT NOT NULL,
  section_no     TEXT NOT NULL,
  capacity       INT NOT NULL CHECK (capacity > 0),
  created_at     TIMESTAMP WITH TIME ZONE DEFAULT now(),
  UNIQUE(course_id, term, section_no)
);
-- Completed courses (student passed a course)
CREATE TABLE IF NOT EXISTS completed_courses (
  student_id  INT NOT NULL REFERENCES students(id) ON DELETE CASCADE,
  course_id   INT NOT NULL REFERENCES courses(id) ON DELETE CASCADE,
  grade       TEXT,
  completed_at TIMESTAMP WITH TIME ZONE DEFAULT now(),
  PRIMARY KEY(student_id, course_id)
);
-- Enrollments (current enrollments)
CREATE TABLE IF NOT EXISTS enrollments (
```

```
    id          SERIAL PRIMARY KEY,
    student_id  INT NOT NULL REFERENCES students(id) ON DELETE CASCADE,
    section_id  INT NOT NULL REFERENCES sections(id) ON DELETE CASCADE,
    status      TEXT NOT NULL CHECK (status IN ('enrolled','dropped','completed')),
    enrolled_at TIMESTAMP WITH TIME ZONE DEFAULT now(),
    UNIQUE(student_id, section_id)
  );
  -- Waitlist (ordered by position)
  CREATE TABLE IF NOT EXISTS waitlist (
    id          SERIAL PRIMARY KEY,
    student_id  INT NOT NULL REFERENCES students(id) ON DELETE CASCADE,
    section_id  INT NOT NULL REFERENCES sections(id) ON DELETE CASCADE,
    position    INT NOT NULL,
    created_at  TIMESTAMP WITH TIME ZONE DEFAULT now(),
    UNIQUE(student_id, section_id),
    UNIQUE(section_id, position)
  );
  CREATE INDEX IF NOT EXISTS idx_waitlist_section_pos ON waitlist(section_id, position);
  -- View: students waiting for enrollment confirmation (per section)
  CREATE OR REPLACE VIEW waitlist_view AS
  SELECT
    w.section_id,
    sec.course_id,
    sec.term,
    sec.section_no,
    w.position,
    w.created_at,
    s.id AS student_id,
    s.student_no,
    s.full_name,
    s.email
  FROM waitlist w
  JOIN students s ON s.id = w.student_id
  JOIN sections sec ON sec.id = w.section_id;
  -- Function: enroll_student
  -- Returns text: 'enrolled', 'waitlisted', 'missing_prereqs', 'already_enrolled', 'section_not_found'
  CREATE OR REPLACE FUNCTION enroll_student(p_student INT, p_section INT)
  RETURNS TEXT LANGUAGE plpgsql AS $$
  DECLARE
    v_course INT;
    missing INT;
    cap INT;
    enrolled_cnt INT;
    next_pos INT;
  BEGIN
```

registration.sql > ...

```
    SELECT course_id INTO v_course FROM sections WHERE id = p_section;
    IF NOT FOUND THEN
      RETURN 'section_not_found';
    END IF;
    -- already enrolled?
    IF EXISTS (
      SELECT 1 FROM enrollments
      WHERE student_id = p_student AND section_id = p_section AND status = 'enrolled'
    ) THEN
      RETURN 'already_enrolled';
    END IF;
    -- check prerequisites: count prereqs not completed by student
    SELECT COUNT(*) INTO missing
    FROM course_prereqs cp
    LEFT JOIN completed_courses cc
      ON cc.course_id = cp.prereq_id AND cc.student_id = p_student
    WHERE cp.course_id = v_course AND cc.student_id IS NULL;
    IF missing > 0 THEN
      RETURN 'missing_prereqs';
    END IF;
    -- capacity & enroll atomically
    PERFORM 1 FROM sections WHERE id = p_section FOR UPDATE;
    SELECT capacity INTO cap FROM sections WHERE id = p_section;
    SELECT COUNT(*) INTO enrolled_cnt FROM enrollments
      WHERE section_id = p_section AND status = 'enrolled';
    IF enrolled_cnt < cap THEN
      INSERT INTO enrollments(student_id, section_id, status, enrolled_at)
        VALUES (p_student, p_section, 'enrolled', now());
      RETURN 'enrolled';
    ELSE
      -- append to waitlist (compute next position under lock)
      SELECT COALESCE(MAX(position),0)+1 INTO next_pos FROM waitlist WHERE section_id = p_section FOR UPDATE;
      INSERT INTO waitlist(student_id, section_id, position, created_at)
        VALUES (p_student, p_section, next_pos, now());
      RETURN 'waitlisted';
    END IF;
  END;
  $$;
  -- Function: promote_from_waitlist(section_id)
  -- Attempts to move the first waitlisted student into enrollment if capacity freed.
  CREATE OR REPLACE FUNCTION promote_from_waitlist(p_section INT)
  RETURNS TEXT LANGUAGE plpgsql AS $$
  DECLARE
    cap INT;
    enrolled_cnt INT;
    w_row RECORD;
  BEGIN
    PERFORM 1 FROM sections WHERE id = p_section FOR UPDATE;
    SELECT capacity INTO cap FROM sections WHERE id = p_section;
    SELECT COUNT(*) INTO enrolled_cnt FROM enrollments WHERE section_id = p_section AND status = 'enrolled';
    IF enrolled_cnt >= cap THEN
      RETURN 'no_capacity';
    END IF;
    SELECT * INTO w_row FROM waitlist WHERE section_id = p_section ORDER BY position, created_at LIMIT 1 FOR UPDATE;
    IF NOT FOUND THEN
```

```
    IF NOT FOUND THEN
      RETURN 'no_waitlist';
    END IF;
    -- enroll the student and remove from waitlist; shift positions
    INSERT INTO enrollments(student_id, section_id, status, enrolled_at)
      VALUES (w_row.student_id, p_section, 'enrolled', now());
    DELETE FROM waitlist WHERE id = w_row.id;
    -- re-compact positions (optional)
    WITH renum AS (
      SELECT id, ROW_NUMBER() OVER (ORDER BY position, created_at) AS rn
      FROM waitlist WHERE section_id = p_section
    )
    UPDATE waitlist w SET position = r.rn
    FROM renum r WHERE w.id = r.id;
    RETURN 'promoted';
  END;
  $$;
  COMMIT;
```

Output :

```
 section_id | position | student_no | full_name | email
------------+----------+------------+-----------+----------------
          1 |        1 | S002       | Bob       | bob@example.com
          1 |        2 | S003       | Carol     | carol@example.com
(2 rows)
```

Observation :

This single-file PostgreSQL schema defines students, courses, prerequisites, sections (with capacity), completed courses, enrollments and an ordered waitlist, and includes two PL/pgSQL helpers—enroll_student (enforces prerequisites and capacity, enrolling or waitlisting atomically) and promote_from_waitlist (promotes the next waiting student when a seat frees). Run the SQL file to create the schema, insert test data, then call enroll_student(...) and query waitlist_view to see waiting students.

Q2. AI suggests denormalizing the database for fast reads.
a) Evaluate pros & cons based on scenario.
b) Decide final approach and justify.

Prompt :

Explain the pros and cons of denormalizing a database for faster reads in a student course registration scenario. Recommend whether to keep normalization or apply denormalization, with clear justification.

Code :

registration_denorm.sql > ...

```sql
-- Denormalized read table for fast waitlist reads + trigger maintenance
BEGIN;
-- Read-table: one-row per waitlist entry with denormalized student/section/course fields
CREATE TABLE IF NOT EXISTS waitlist_read (
  section_id   INT NOT NULL,
  course_id    INT,
  term         TEXT,
  section_no   TEXT,
  position     INT,
  created_at   TIMESTAMP WITH TIME ZONE,
  student_id   INT NOT NULL,
  student_no   TEXT,
  full_name    TEXT,
  email        TEXT,
  PRIMARY KEY(section_id, student_id)
);
-- Convenience: populate initial content from existing view (if any)
INSERT INTO waitlist_read (section_id, course_id, term, section_no, position, created_at, student_id, student_no, full_name, email)
SELECT w.section_id, sec.course_id, sec.term, sec.section_no, w.position, w.created_at, s.id, s.student_no, s.full_name, s.email
FROM waitlist w
JOIN students s ON s.id = w.student_id
JOIN sections sec ON sec.id = w.section_id
ON CONFLICT DO NOTHING;
-- Trigger function: upsert on waitlist INSERT/UPDATE
CREATE OR REPLACE FUNCTION trg_waitlist_upsert()
RETURNS TRIGGER LANGUAGE plpgsql AS $$
BEGIN
  IF TG_OP = 'INSERT' OR TG_OP = 'UPDATE' THEN
    INSERT INTO waitlist_read(section_id, course_id, term, section_no, position, created_at, student_id, student_no, full_name, email)
    SELECT NEW.section_id, sec.course_id, sec.term, sec.section_no, NEW.position, NEW.created_at, s.id, s.student_no, s.full_name, s.email
    FROM students s JOIN sections sec ON sec.id = NEW.section_id
    WHERE s.id = NEW.student_id
    ON CONFLICT (section_id, student_id) DO UPDATE
      SET position = EXCLUDED.position,
          created_at = EXCLUDED.created_at,
          course_id = EXCLUDED.course_id,
          term = EXCLUDED.term,
          section_no = EXCLUDED.section_no,
          student_no = EXCLUDED.student_no,
          full_name = EXCLUDED.full_name,
          email = EXCLUDED.email;
    RETURN NEW;
  ELSIF TG_OP = 'DELETE' THEN
    DELETE FROM waitlist_read WHERE section_id = OLD.section_id AND student_id = OLD.student_id;
    RETURN OLD;
```

registration_denorm.sql > ...

```
    END IF;
  END;
  $$;
  -- Attach triggers to waitlist table (after change so read-table sees committed state)
  DROP TRIGGER IF EXISTS waitlist_upsert_trig ON waitlist;
  CREATE TRIGGER waitlist_upsert_trig
  AFTER INSERT OR UPDATE OR DELETE ON waitlist
  FOR EACH ROW EXECUTE FUNCTION trg_waitlist_upsert();
  -- Trigger: propagate student info updates to read-table
  CREATE OR REPLACE FUNCTION trg_student_update()
  RETURNS TRIGGER LANGUAGE plpgsql AS $$
  BEGIN
    IF TG_OP = 'UPDATE' THEN
      UPDATE waitlist_read
        SET student_no = NEW.student_no,
            full_name  = NEW.full_name,
            email      = NEW.email
      WHERE student_id = NEW.id;
    END IF;
    RETURN NEW;
  END;
  $$;
  DROP TRIGGER IF EXISTS student_update_trig ON students;
  CREATE TRIGGER student_update_trig
  AFTER UPDATE ON students
  FOR EACH ROW EXECUTE FUNCTION trg_student_update();
  -- Trigger: propagate section info updates (term/section_no) to read-table
  CREATE OR REPLACE FUNCTION trg_section_update()
  RETURNS TRIGGER LANGUAGE plpgsql AS $$
  BEGIN
    IF TG_OP = 'UPDATE' THEN
      UPDATE waitlist_read
        SET course_id = NEW.course_id,
            term = NEW.term,
            section_no = NEW.section_no
      WHERE section_id = NEW.id;
    END IF;
    RETURN NEW;
  END;
  $$;
  DROP TRIGGER IF EXISTS section_update_trig ON sections;
  CREATE TRIGGER section_update_trig
  AFTER UPDATE ON sections
  FOR EACH ROW EXECUTE FUNCTION trg_section_update();
```

```
  FOR EACH ROW EXECUTE FUNCTION trg_section_update();

  -- Full rebuild utility (call when you want to refresh from scratch)
  CREATE OR REPLACE FUNCTION rebuild_waitlist_read()
  RETURNS VOID LANGUAGE plpgsql AS $$
  BEGIN
    TRUNCATE waitlist_read;
    INSERT INTO waitlist_read (section_id, course_id, term, section_no, position, created_at, student_id, student_no, full_name, email)
    SELECT w.section_id, sec.course_id, sec.term, sec.section_no, w.position, w.created_at, s.id, s.student_no, s.full_name, s.email
    FROM waitlist w
    JOIN students s ON s.id = w.student_id
    JOIN sections sec ON sec.id = w.section_id
    ORDER BY w.section_id, w.position;
  END;
  $$;
  COMMIT;
```

Output :

```
psql -U postgres -d course_db -c "SELECT section_id, position, student_id, student_no, full_name, email FROM waitlist
 section_id | position | student_id | student_no | full_name |       email
------------+----------+------------+------------+-----------+--------------------
          1 |        1 |          2 | S002       | Bob       | bob@example.com
          1 |        2 |          3 | S003       | Carol     | carol@example.com
(2 rows)
```

Observation :

The file creates a denormalized waitlist_read table, seeds it from existing waitlist/students/sections, and installs AFTER triggers to upsert/delete rows and propagate student/section updates, plus a rebuild utility. It assumes the base tables exist (running it beforehand will error); triggers provide real-time reads but add write overhead and must be tested under concurrency—consider adding indexes on course_id/position, foreign keys, and a maintenance plan for full rebuilds.