Bilkent University

Department Of Computer Engineering

CS-353 Database Systems

Group 10

Project Design

27.03.2025

Instructor: Özgür Ulusoy

Teaching Assistant: Hasan Alp Caferoğlu

Group Members:

Name	ID	Section
Emre Furkan Akyol	22103352	2
Ayça Candan Ataç	22203501	2
İbrahim Çaycı	22103515	1
Mustafa Özkan İr	22103267	2
Cem Apaydın	21802270	1

1. Revised E-R

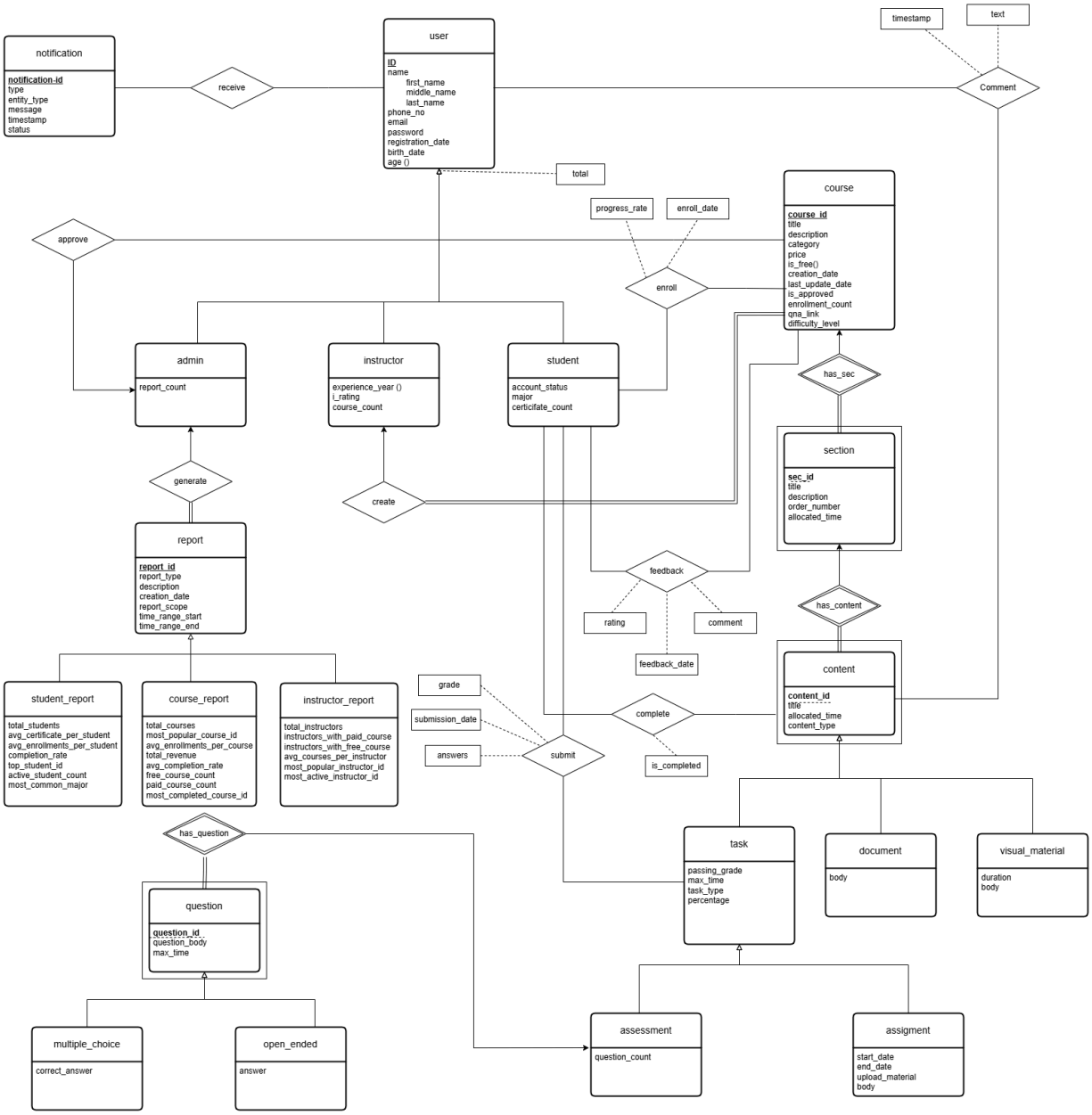
For high-resolution diagrams:

https://drive.google.com/file/d/1RqGTpdd39KD4W69ER-8BEV0qu-2jXui5/view?usp=sharing

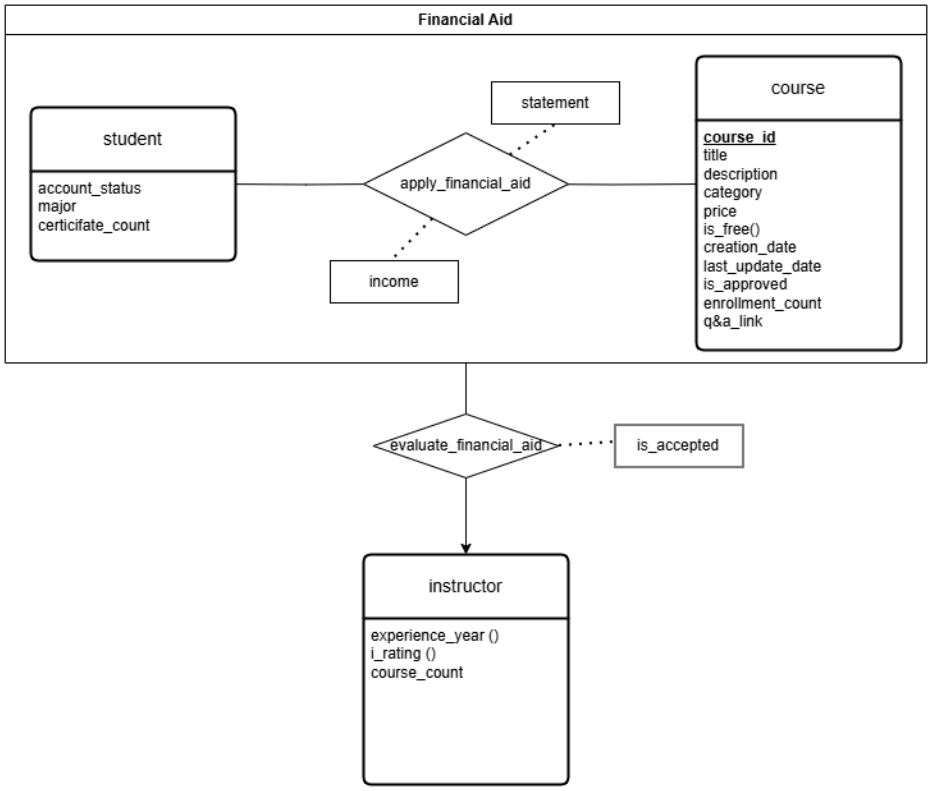
1.1. Changes to the E-R Diagram

- The report entity is made more detailed.
- The aggregation for feedback is converted into a binary relationship.
- The primary key of the task entity is removed as it inherits from the content entity.
- The cardinality of the submit relation is corrected.
- Minor attribute adjustments for various entities are made.

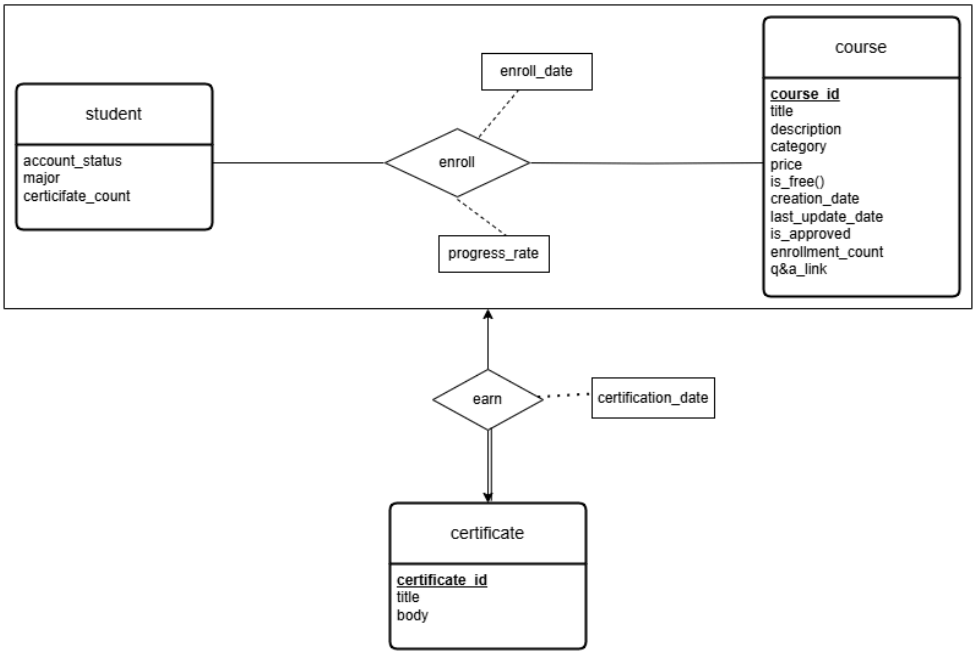
1.2. Main E-R Diagram



1.3. Financial Aid E-R Diagram



1.4. Certificate E-R Diagram



2. Table Schemas

2.1. user

Relation: user(ID, first_name, middle_name, last_name, phone_no, email, password, registration_date, birth_date)

Candidate Keys: {ID, email}

Primary Key: ID

Foreign Keys: -

Table Definition:

```
    CREATE TABLE user (
        ID VARCHAR(8),
        first_name VARCHAR(50) NOT NULL,
        middle_name VARCHAR(50),
        last_name VARCHAR(50) NOT NULL,
        phone_no VARCHAR(15),
        email VARCHAR(100) UNIQUE NOT NULL,
        password VARCHAR(100) NOT NULL,
        registration_date DATE NOT NULL,
        birth_date DATE NOT NULL,
        PRIMARY KEY (ID),
        CHECK (registration_date <= CURRENT_DATE)
    );

/* Although the derived attribute “age” is not included in the relation
schema (as recommended in the book), it can be calculated and used via
the following view:*/
CREATE VIEW user_with_age AS
SELECT
    ID,
    first_name,
    last_name,
    birth_date,
    EXTRACT(YEAR FROM CURRENT_DATE) - EXTRACT(YEAR FROM birth_date) AS
age
FROM user;
```

2.2. admin

Relation: admin(ID, report_count)

Candidate Keys: {ID}

Primary Key: ID

Foreign Keys: ID → user(ID)

Table Definition:

```
    CREATE TABLE admin (
        ID VARCHAR(8),
        report_count INTEGER DEFAULT 0 CHECK (report_count >= 0) NOT
                                                                NULL,
        PRIMARY KEY (ID),
        FOREIGN KEY (ID) REFERENCES user(ID)
    );

/*Report_count of admin can be maintained with a trigger as follows: */

CREATE TRIGGER increment_report_count
AFTER INSERT ON report
REFERENCING NEW ROW AS nrow
FOR EACH ROW
BEGIN ATOMIC
    UPDATE admin
    SET report_count = report_count + 1
    WHERE ID = nrow.admin_id;
END;
```

2.3. instructor

Relation: instructor(ID, i_rating, course_count)

Candidate Keys: {ID}

Primary Key: ID

Foreign Keys: ID → user(ID)

Table Definition:

```
    CREATE TABLE instructor (
        ID VARCHAR(8),
```

```
        i_rating FLOAT CHECK (i_rating BETWEEN 0 AND 5),
        course_count INTEGER DEFAULT 0 CHECK (course_count >= 0),
        PRIMARY KEY (ID),
        FOREIGN KEY (ID) REFERENCES user(ID)
    );


/* Although the derived attribute “experience_year” is not included in
the relation schema (as recommended in the book), it can be calculated
and used via the following view:*/
CREATE VIEW instructor_with_experience_year AS
SELECT
    i.ID,
    u.first_name,
    u.last_name,
    EXTRACT(YEAR FROM CURRENT_DATE) - EXTRACT(YEAR FROM
u.registration_date) AS experience_year
FROM instructor i
JOIN user u ON i.ID = u.ID;

/*i_rating of the instructor can be maintained with a trigger as
follows:*/
CREATE TRIGGER update_i_rating
AFTER INSERT ON feedback
REFERENCING NEW ROW AS nrow
FOR EACH ROW
BEGIN ATOMIC
    UPDATE instructor
    SET i_rating = (
        SELECT AVG(f.rating)
        FROM feedback f
        JOIN course c ON f.course_id = c.course_id
        WHERE c.instructor_id = instructor.ID
    )
    WHERE instructor.ID = (
        SELECT c.instructor_id
        FROM course c
        WHERE c.course_id = nrow.course_id
    );
END;
```

2.4. student

Relation: student(ID, account_status, major, certificate_count)

Candidate Keys: {ID}

Primary Key: ID

Foreign Keys: ID → user(ID)

Table Definition:

```
    CREATE TABLE student (
        major VARCHAR(50),
        ID VARCHAR(8),
        account_status VARCHAR(20),
        certificate_count INTEGER DEFAULT 0 CHECK (certificate_count >=
      0),
        PRIMARY KEY (ID),
        FOREIGN KEY (ID) REFERENCES user(ID)
    );

/*The certificate_count of the student can be maintained with a trigger
as follows:*/

CREATE TRIGGER update_certificate_count
AFTER INSERT ON earn_certificate
REFERENCING NEW ROW AS nrow
FOR EACH ROW
BEGIN ATOMIC
    UPDATE student
    SET certificate_count = certificate_count + 1
    WHERE ID = nrow.student_id;
END;
```

2.5. report

Relation: report(report_id, admin_id, report_type, description, creation_date, report_scope, time_range_start, time_range_end)

Candidate Keys: {report_id}

Primary Key: report_id

Foreign Keys: admin_id → admin(ID)

Table Definition:

```
CREATE TABLE report (
      report_id VARCHAR(8),
      admin_id VARCHAR(8),
      report_type VARCHAR(20),
      description TEXT,
      creation_date DATETIME,
      report_scope VARCHAR(50),
      time_range_start DATE,
      time_range_end DATE,
      PRIMARY KEY (report_id)
      FOREIGN KEY (admin_id) REFERENCES admin(ID)
      CHECK (time_range_start <= time_range_end)
);
```

2.6. student_report

Relation: student_report(report_id, total_students, avg_certificate_per_student, avg_enrollments_per_student, completion_rate, top_student_id, active_student_count, most_common_major)

Candidate Keys: {report_id}

Primary Key: report_id

Foreign Keys:
- report_id → report(report_id)
- top_student_id → student(ID)

Table Definition:

```
CREATE TABLE student_report (
    report_id VARCHAR(8),
    total_students INTEGER CHECK (total_students >= 0),
    avg_certificate_per_student FLOAT,
    avg_enrollments_per_student FLOAT,
    completion_rate FLOAT,
    top_student_id VARCHAR(8),
    active_student_count INTEGER CHECK (active_student_count >= 0),
    most_common_major VARCHAR(10),
    PRIMARY KEY (report_id),
    FOREIGN KEY (report_id) REFERENCES report(report_id),
    FOREIGN KEY (top_student_id) REFERENCES student(ID),
```

```
        CHECK (avg_certificate_per_student >= 0),
        CHECK (avg_enrollments_per_student >= 0)
    );
```

2.7. course_report

Relation: course_report(report_id, total_courses, most_popular_course_id, avg_enrollments_per_course, total_revenue, avg_completion_rate, free_course_count, paid_course_count, most_completed_course_id)

Candidate Keys: {report_id}

Primary Key: report_id

Foreign Keys:
- report_id → report(report_id)
- most_popular_course_id → course(course_id)
- most_completed_course_id → course(course_id)

Table Definition:

```
    CREATE TABLE course_report (
        report_id VARCHAR(8),
        total_courses INTEGER CHECK (total_courses >= 0),
        most_popular_course_id VARCHAR(8),
        avg_enrollments_per_course FLOAT,
        total_revenue INTEGER CHECK (total_revenue >= 0),
        avg_completion_rate FLOAT,
        free_course_count INTEGER CHECK (free_course_count >= 0),
        paid_course_count INTEGER CHECK (paid_course_count >= 0),
        most_completed_course_id VARCHAR(8),
        PRIMARY KEY (report_id),
        FOREIGN KEY (report_id) REFERENCES report(report_id),
        FOREIGN KEY (most_popular_course_id) REFERENCES
                                        course(course_id),
        FOREIGN KEY (most_completed_course_id) REFERENCES
                                        course(course_id),
        CHECK (avg_enrollments_per_course >= 0),
    );
```

2.8. instructor_report

Relation: instructor_report(report_id, total_instructors, instructors_with_paid_course, instructors_with_free_course,

avg_courses_per_instructor, most_popular_instructor_id,
most_active_instructor_id)

Candidate Keys: {report_id}

Primary Key: report_id

Foreign Keys:
- report_id → report(report_id)
- most_popular_instructor_id → instructor(id)
- most_active_instructor_id → instructor(id)

Table Definition:

```
CREATE TABLE instructor_report (
  report_id VARCHAR(8),
  total_instructors INTEGER,
  instructors_with_paid_course INTEGER,
  instructors_with_free_course INTEGER,
  avg_courses_per_instructor FLOAT,
  most_popular_instructor_id VARCHAR(8),
  most_active_instructor_id VARCHAR(8),
  PRIMARY KEY (report_id),
  FOREIGN KEY (report_id) REFERENCES report(report_id),
  FOREIGN KEY (most_popular_instructor_id) REFERENCES instructor(id),
  FOREIGN KEY (most_active_instructor_id) REFERENCES instructor(id),
  CHECK (total_instructors >= 0),
  CHECK (instructors_with_paid_course  >= 0),
  CHECK (instructors_with_free_course >= 0),
  CHECK (avg_courses_per_instructor >= 0)
);
```

2.9. notification

Relation: notification(notification_id, type, entity_type, message,
timestamp, status)

Candidate Keys: {notification_id}

Primary Key: notification_id

Foreign Keys: -

Table Definition:

```
    CREATE TABLE notification (
```

```
        notification_id VARCHAR(8),
        type VARCHAR(30),
        entity_type VARCHAR(8),
        message TEXT NOT NULL,
        timestamp DATETIME NOT NULL,
        status VARCHAR(20) CHECK (
        status IN ('unread', 'read', 'archived')
        ),
        PRIMARY KEY (notification_id)
    );
```

2.10. receive

Relation: receive(notification_id, ID

Candidate Keys: {(notification_id, ID)}

Primary Key: (notification_id, ID)

Foreign Keys:
- notification_id → notification(notification_id)
- ID → user(ID)

Table Definition:

```
    CREATE TABLE receive(
        notification_id VARCHAR(8),
        ID VARCHAR(8),
        PRIMARY KEY (notification_id, ID),
        FOREIGN KEY notification_id REFERENCES
                                    notification(notification_id),
        FOREIGN KEY ID REFERENCES user(ID)
    );
```

2.11. content

Relation: content(course_id, sec_id, content_id, title,
allocated_time, content_type)

Candidate Keys: {(course_id, sec_id, content_id)}

Primary Key: (course_id, sec_id, content_id)

Foreign Keys:
- (course_id, sec_id) → section(course_id, sec_id)

Table Definition:

```
CREATE TABLE content(
   course_id VARCHAR(8),
   sec_id VARCHAR(8),
   content_id VARCHAR(8),
   title VARCHAR(150) NOT NULL,
   allocated_time INTEGER CHECK (allocated_time >= 0),
   content_type CHECK(content_type IN
            (‘task’, ‘document’, ‘visual_material’)),
   PRIMARY KEY (course_id, sec_id, content_id),
   FOREIGN KEY (course_id, sec_id) REFERENCES section(course_id, sec_id)
);
```

2.12. task

Relation: task(course_id, sec_id, content_id, passing_grade, max_time, task_type, percentage)

Candidate Keys: {(course_id, sec_id, content_id)}

Primary Key: (course_id, sec_id, content_id)

Foreign Keys:

- (course_id, sec_id, content_id) → content(course_id, sec_id, content_id)

Table Definition:

```
CREATE TABLE task(
      course_id VARCHAR(8),
      sec_id VARCHAR(8),
      content_id VARCHAR(8),
      passing_grade INTEGER NOT NULL CHECK (passing_grade BETWEEN 0 AND
100),
      max_time INTEGER CHECK (max_time > 0),
      task_type VARCHAR(20) CHECK (task_type IN ('assessment',
                                          'assignment')),
      percentage INTEGER NOT NULL CHECK (percentage BETWEEN 0 AND 100),
      PRIMARY KEY (course_id, sec_id, content_id),
      FOREIGN KEY (course_id, sec_id, content_id) REFERENCES
      content(course_id, sec_id, content_id)
);
```

2.13. assessment

Relation: assessment(course_id, sec_id, content_id, question_count)

Candidate Keys: {(course_id, sec_id, content_id)}

Primary Key: (course_id, sec_id, content_id)

Foreign Keys:
- (course_id, sec_id, content_id) → task(course_id, sec_id, content_id)

Table Definition:

```
CREATE TABLE assessment(
  course_id VARCHAR(8),
  sec_id VARCHAR(8),
  content_id VARCHAR(8),
  question_count INTEGER CHECK (question_count > 0),
  PRIMARY KEY (course_id, sec_id, content_id),
  FOREIGN KEY (course_id, sec_id, content_id)
        REFERENCES task(course_id, sec_id, content_id)
);
```

2.14. assignment

Relation: assignment(course_id, sec_id, content_id, start_date, end_date, upload_material, body)

Candidate Keys: {(course_id, sec_id, content_id)}

Primary Key: (course_id, sec_id, content_id)

Foreign Keys:
- (course_id, sec_id, content_id) → task(course_id, sec_id, content_id)

Table Definition:

```
CREATE TABLE assignment(
      course_id VARCHAR(8),
      sec_id VARCHAR(8),
      content_id VARCHAR(8),
      start_date DATE NOT NULL,
      end_date DATE NOT NULL,
      upload_material VARCHAR(10) NOT NULL CHECK (upload_material IN
      ('zip', 'pdf', 'xls', 'xlsx', 'doc', 'docx', 'txt', 'ppt',
      'pptx')),
```

```
        body TEXT,
        PRIMARY KEY (course_id, sec_id, content_id),
        FOREIGN KEY (course_id, sec_id, content_id)
        REFERENCES task(course_id, sec_id, content_id),
        CHECK (end_date > start_date)
 );
```

2.15. document

Relation: document(course_id, sec_id, content_id, body)

Candidate Keys: {(course_id, sec_id, content_id)}

Primary Key: (course_id, sec_id, content_id)

Foreign Keys:
 - (course_id, sec_id, content_id) → content(course_id, sec_id, content_id)

Table Definition:

```
CREATE TABLE document(
        course_id VARCHAR(8),
        sec_id VARCHAR(8),
        content_id VARCHAR(8),
        body TEXT,
        PRIMARY KEY (course_id, sec_id, content_id),
        FOREIGN KEY (course_id, sec_id, content_id)
        REFERENCES content(course_id, sec_id, content_id)
 );
```

2.16. visual_material

Relation: visual_material(course_id, sec_id, content_id, duration, body)

Candidate Keys: {(course_id, sec_id, content_id)}

Primary Key: (course_id, sec_id, content_id)

Foreign Keys:
 - (course_id, sec_id, content_id) → content(course_id, sec_id, content_id)

Table Definition:

```
CREATE TABLE visual_material(
        course_id VARCHAR(8),
```

```
        sec_id VARCHAR(8),
        content_id VARCHAR(8),
        duration INTEGER CHECK (duration > 0),
        body TEXT,
        PRIMARY KEY (course_id, sec_id, content_id),
        FOREIGN KEY (course_id, sec_id, content_id)
        REFERENCES content(course_id, sec_id, content_id));
```

2.17. section

Relation: section(course_id, sec_id, title, description, order_number, allocated_time)

Candidate Keys: {(course_id, sec_id), (course_id, order_number)}

Primary Key: (course_id, sec_id)

Foreign Keys:
- course_id → course(course_id)

Table Definition:

```
CREATE TABLE section(
      course_id VARCHAR(8),
      sec_id VARCHAR(8),
      title VARCHAR(150) NOT NULL,
      description TEXT,
      order_number INTEGER NOT NULL CHECK (order_number >= 0),
      allocated_time INTEGER CHECK (allocated_time >= 0),
      PRIMARY KEY (course_id, sec_id),
      FOREIGN KEY course_id REFERENCES course(course_id)
);
```

2.18. multiple_choice

Relation: multiple_choice(course_id, sec_id, content_id, question_id, correct_answer)

Candidate Keys: {(course_id, sec_id, content_id, question_id)}

Primary Key: (course_id, sec_id, content_id, question_id)

Foreign Keys:
- (course_id, sec_id, content_id, question_id) → question(course_id, sec_id, content_id, question_id)

Table Definition:

```
CREATE TABLE multiple_choice(
      course_id VARCHAR(8),
      sec_id VARCHAR(8),
      content_id VARCHAR(8),
      question_id VARCHAR(8),
      correct_answer CHAR(1) CHECK (correct_answer IN ('A', 'B', 'C',
      'D', 'E')),
       PRIMARY KEY (course_id, sec_id, content_id, question_id),
       FOREIGN KEY (course_id, sec_id, content_id, question_id)
       REFERENCES question(course_id, sec_id, content_id,
      question_id)
);
```

2.19. open_ended

Relation: open_ended(course_id, sec_id, content_id, question_id, answer)

Candidate Keys: {(course_id, sec_id, content_id, question_id)}

Primary Key: (course_id, sec_id, content_id, question_id)

Foreign Keys:

- (course_id, sec_id, content_id, question_id) → question(course_id, sec_id, content_id, question_id)

Table Definition:

```
      CREATE TABLE open_ended(
      course_id VARCHAR(8),
      sec_id VARCHAR(8),
      content_id VARCHAR(8),
      question_id VARCHAR(8),
       answer TEXT,
       PRIMARY KEY (course_id, sec_id, content_id, question_id),
       FOREIGN KEY (course_id, sec_id, content_id, question_id)
       REFERENCES question(course_id, sec_id, content_id,
      question_id)
);
```

2.20. question

Relation: question(course_id, sec_id, content_id, question_id, question_body, max_time)

Candidate Keys: {(course_id, sec_id, content_id, question_id)}

Primary Key: (course_id, sec_id, content_id, question_id)

Foreign Keys:
- (course_id, sec_id, content_id) → assessment(course_id, sec_id, content_id)

Table Definition:

```
CREATE TABLE question(
  course_id VARCHAR(8),
   sec_id VARCHAR(8),
  content_id VARCHAR(8),
  question_id VARCHAR(8),
  question_body VARCHAR(2000),
  max_time INTEGER CHECK (max_time >= 0),
  PRIMARY KEY (course_id, sec_id, content_id, question_id),
  FOREIGN KEY (course_id, sec_id, content_id)
  REFERENCES assessment(course_id, sec_id, content_id));
```

2.21. course

Relation: course(course_id, title, description, category, price, creation_date, last_update_date, is_approved, enrollment_count, qna_link, difficulty_level, creator_id, approver_id)

Candidate Keys: {course_id, title, qna_link}

Primary Key: (course_id)

Foreign Keys:
- creator_id → instructor(ID)
- approver_id → admin(ID)

Table Definition:

```
CREATE TABLE course(
       course_id VARCHAR(8),
       title VARCHAR(150) NOT NULL,
       description VARCHAR(2000),
       category VARCHAR(50),
       price INTEGER CHECK (price >= 0),
       creation_date DATE,
       last_update_date DATE,
```

```
    is_approved BOOLEAN DEFAULT FALSE,
    enrollment_count INTEGER CHECK (enrollment_count >= 0),
    qna_link VARCHAR(100),
    difficulty_level INTEGER CHECK (difficulty_level BETWEEN 1 AND
    5),
    creator_id VARCHAR(8) NOT NULL,
    approver_id VARCHAR(8),
    PRIMARY KEY (course_id),
    FOREIGN KEY creator_id REFERENCES instructor(ID),
    FOREIGN KEY approver_id REFERENCES admin(ID));
```

```
/* Although the derived attribute “is_free” is not included in the
relation schema (as recommended in the book), it can be calculated and
used via the following view:*/
CREATE VIEW course_with_is_free AS
SELECT
    course_id,
    title,
    price,
    CASE
        WHEN price = 0 THEN TRUE
        ELSE FALSE
    END AS is_free
FROM
    course;
```

2.22. enroll

Relation: enroll(course_id, student_id, enroll_date, progress_rate)

Candidate Keys: {(course_id, student_id)}

Primary Key: (course_id, student_id)

Foreign Keys:
- student_id → student(ID)

Table Definition:

```
    CREATE TABLE enroll(
      course_id VARCHAR(8),
      student_id VARCHAR(8),
      enroll_date DATE,
```

```
        progress_rate INTEGER CHECK (progress_rate BETWEEN 0 AND 100),
        PRIMARY KEY (course_id, student_id),
        FOREIGN KEY course_id REFERENCES course(course_id),
        FOREIGN KEY student_id REFERENCES student(ID)
 );
```

2.23. submit

Relation: submit(course_id, sec_id, content_id, student_id, grade, submission_date, answers)

Candidate Keys: {(course_id, sec_id, content_id, student_id)}

Primary Key: (course_id, sec_id, content_id, student_id)

Foreign Keys:

- (course_id, sec_id, content_id) → task(course_id, sec_id, content_id)
- student_id → student(ID)

Table Definition:

```
CREATE TABLE submit(
        course_id VARCHAR(8),
        sec_id VARCHAR(8),
        content_id VARCHAR(8),
        student_id VARCHAR(8),
        grade INTEGER CHECK (grade BETWEEN 0 AND 100),
        submission_date DATE,
        answers TEXT,
        PRIMARY KEY (course_id, sec_id, content_id, student_id),
        FOREIGN KEY (course_id, sec_id, content_id)
        REFERENCES task(course_id, sec_id, content_id),
        FOREIGN KEY student_id REFERENCES student(ID)
 );
```

2.24. feedback

Relation: feedback(course_id, student_id, rating, comment, feedback_date)

Candidate Keys: {(course_id, student_id)}

Primary Key: (course_id, student_id)

Foreign Keys:
- course_id → course(course_id)
- student_id → student(ID)

Table Definition:

```
CREATE TABLE feedback(
      course_id VARCHAR(8),
      student_id VARCHAR(8),
      rating INTEGER NOT NULL CHECK (rating BETWEEN 0 AND 5),
      comment VARCHAR(500),
      feedback_date DATE,
      PRIMARY KEY (course_id, student_id),
      FOREIGN KEY course_id REFERENCES course(course_id),
      FOREIGN KEY student_id REFERENCES student(ID)
);
```

2.25. comment

Relation: comment(course_id, sec_id, content_id, user_id, text, timestamp)

Candidate Keys: {(course_id, sec_id, content_id, user_id)}

Primary Key: (course_id, sec_id, content_id, user_id)

Foreign Keys:
- (course_id, sec_id, content_id) → content(course_id, sec_id, content_id)
- user_id → user(ID)

Table Definition:

```
CREATE TABLE comment(
      course_id VARCHAR(8),
      sec_id VARCHAR(8),
      content_id VARCHAR(8),
      user_id VARCHAR(8),
      text VARCHAR(500) NOT NULL,
      timestamp DATETIME,
      PRIMARY KEY (course_id, sec_id, content_id, user_id),
      FOREIGN KEY (course_id, sec_id, content_id)
      REFERENCES content(course_id, sec_id, content_id),
      FOREIGN KEY user_id REFERENCES user(ID)
);
```

2.26. apply_financial_aid

Relation: apply_financial_aid(course_id, student_id, income, statement)

Candidate Keys: {(course_id, student_id)}

Primary Key: (course_id, student_id)

Foreign Keys:
- course_id → course(course_id)
- student_id → student(ID)

Table Definition:

```
CREATE TABLE apply_financial_aid (
     course_id     VARCHAR(8),
     student_id    VARCHAR(8),
     income        DECIMAL(10,2) CHECK (income >= 0),
     statement     TEXT,
     PRIMARY KEY (course_id, student_id),
     FOREIGN KEY (course_id) REFERENCES course(course_id),
     FOREIGN KEY (student_id) REFERENCES student(ID)
 );
```

2.27. evaluate_financial_aid

Relation: evaluate_financial_aid(course_id, student_id, instructor_id, is_accepted)

Candidate Keys: {(course_id, student_id, instructor_id)}

Primary Key: (course_id, student_id, instructor_id)

Foreign Keys:
- (course_id, student_id) → apply_financial_aid(course_id, student_id)
- instructor_id → instructor(ID)

Table Definition:

```
CREATE TABLE evaluate_financial_aid(
      course_id VARCHAR(8),
      student_id VARCHAR(8),
      instructor_id VARCHAR(8),
      is_accepted BOOLEAN DEFAULT FALSE,
      PRIMARY KEY (course_id, student_id, instructor_id),
```

```
        FOREIGN KEY (course_id, student_id)
        REFERENCES apply_financial_aid(course_id, student_id),
        FOREIGN KEY (instructor_id) REFERENCES instructor(ID)
 );
```

2.28. certificate

Relation: certificate(certificate_id, title, body)

Candidate Keys: {certificate_id, title}

Primary Key: (certificate_id)

Foreign Keys: -

Table Definition:

```
CREATE TABLE certificate(
        certificate_id VARCHAR(8),
        title VARCHAR(150),
        body VARCHAR(10000),
        PRIMARY KEY (certificate_id)
 );
```

2.29. earn_certificate

Relation: earn_certificate(course_id, student_id, certificate_id)

Candidate Keys: {(student_id, course_id, certificate_id)}

Primary Key: (student_id, course_id, certificate_id)

Foreign Keys:
- (student_id, course_id) → enroll(student_id, course_id)
- certificate_id → certificate(certificate_id)

Table Definition:

```
CREATE TABLE earn_certificate(
    student_id VARCHAR(8),
    course_id VARCHAR(8),
    certificate_id VARCHAR(8),
    certification_date DATE,
    PRIMARY KEY (student_id, course_id, certificate_id),
    FOREIGN KEY (student_id, course_id)
    REFERENCES enroll(student_id, course_id),
    FOREIGN KEY (certificate_id)
```

```
    REFERENCES certificate(certificate_id)
);
```

2.30. complete

Relation: complete(course_id, sec_id, content_id, student_id, is_completed)

Candidate Keys: {(course_id, sec_id, content_id, student_id)}

Primary Key: (course_id, sec_id, content_id, student_id)

Foreign Keys:

- (course_id, sec_id, content_id) → content(course_id, sec_id, content_id)
- student_id → student(ID)

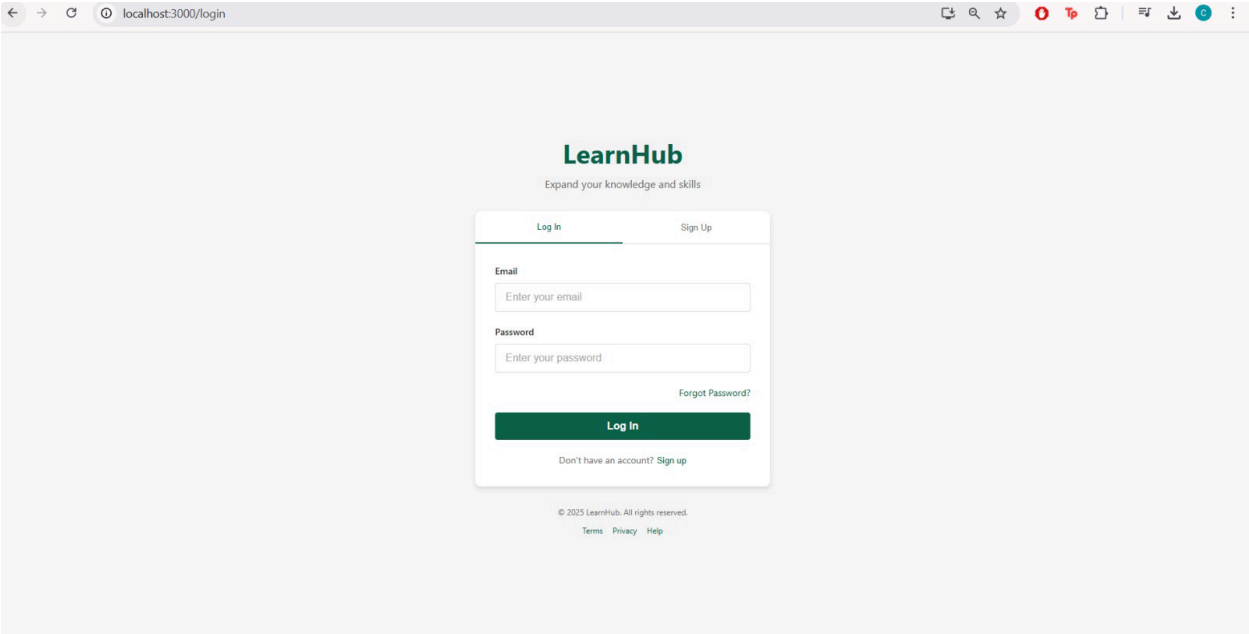
Table Definition:

```
CREATE TABLE complete(
       course_id VARCHAR(8),
       sec_id VARCHAR(8),
       content_id VARCHAR(8),
       student_id VARCHAR(8),
       is_completed BOOLEAN DEFAULT FALSE,
       PRIMARY KEY (course_id, sec_id, content_id, student_id),
       FOREIGN KEY (course_id, sec_id, content_id)
       REFERENCES content(course_id, sec_id, content_id),
       FOREIGN KEY student_id REFERENCES student(ID)
);
```

3. User Interface Design and Corresponding SQL Statements

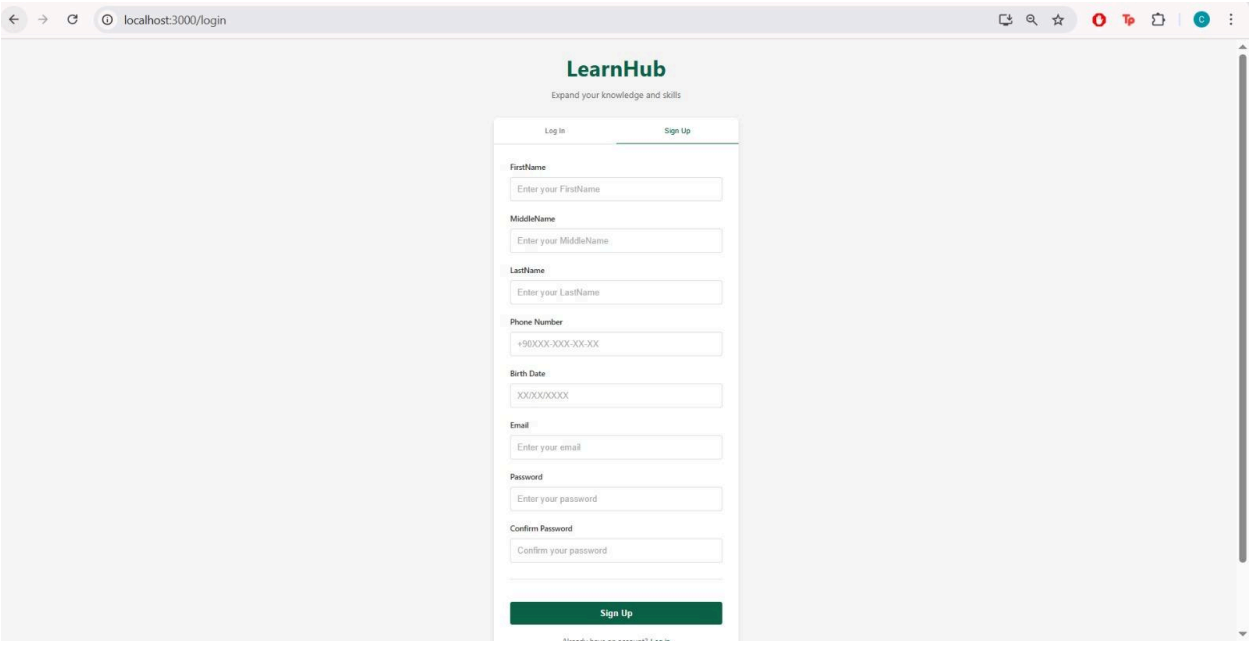
Note: @ sign is used to indicate the variables that are passed to queries from the application.

3.1 Login Page



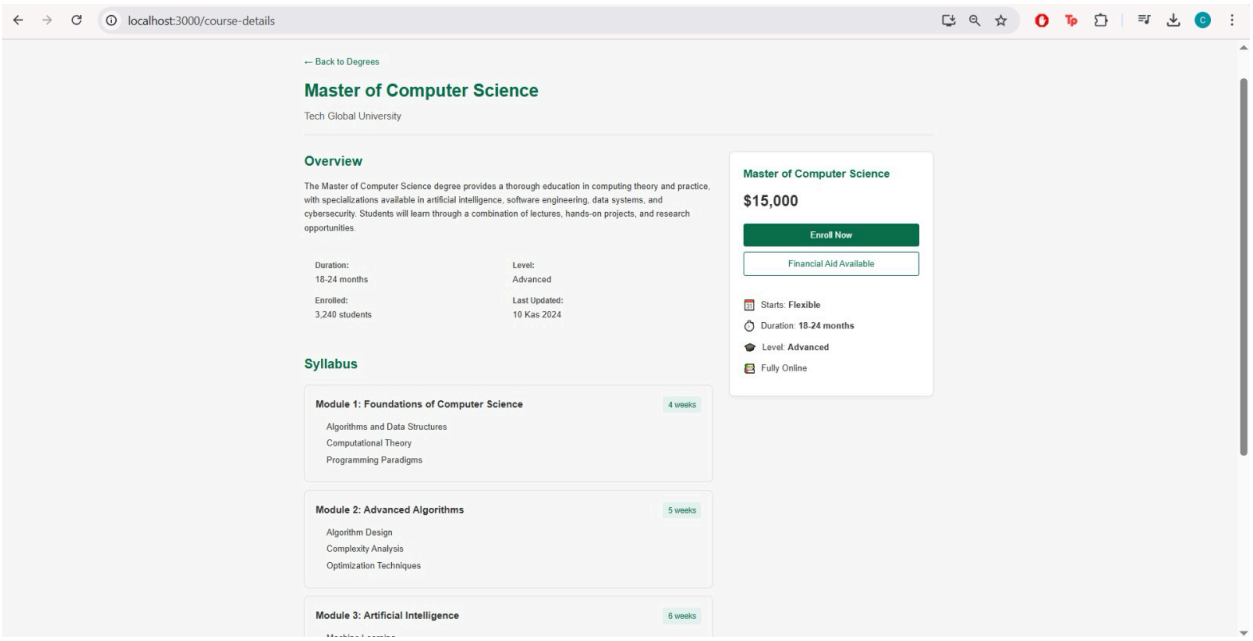

```
SELECT u.ID, u.first_name, u.middle_name, u.last_name, u.email
FROM user AS u
WHERE u.email = @email AND u.password = @password;
```

3.2 Registration Page




```
INSERT INTO user (ID, first_name, middle_name, last_name, phone_no, email,
                  password, registration_date, birth_date)
VALUES (@newUserID, @firstName, @middleName, @lastName, @phoneNO, @email,
                  @password, @CURRENT_DATE, @birthDate);
```

3.3 Course Overview Page



Enrollment queries:

```
INSERT INTO enroll (course_id,student_id, enroll_date,  progress_rate)
VALUES (@courseID, @studentID, CURRENT_DATE, 0);
```

Trigger for updating enrollment count of the course:

```
CREATE TRIGGER enrollment_count_updater
REFERENCING NEW ROW AS nrow
AFTER INSERT ON enroll
FOR EACH ROW
BEGIN ATOMIC
    UPDATE course
    SET enrollment_count = enrollment_count + 1
    WHERE course_id = nrow.course_id;
END;
```

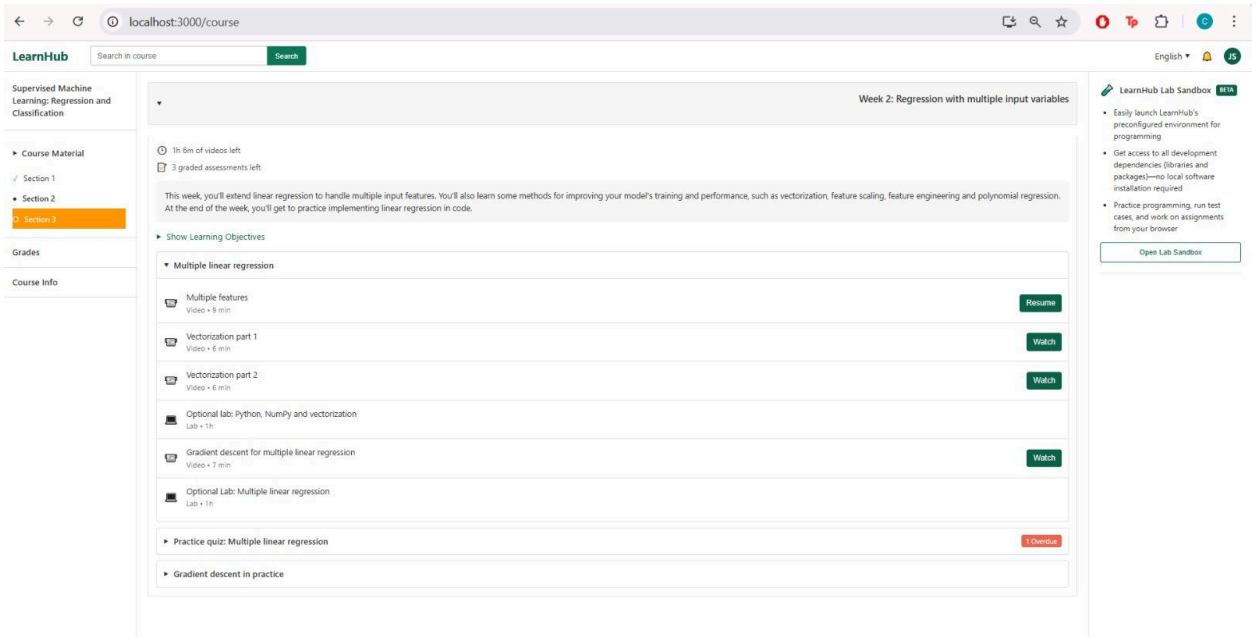
Overview Information Retrieval:

```
SELECT c.title, c.description, c.category, c.price, c.last_update_date,
  c.is_approved, c.enrollment_count, c.difficulty_level
FROM course AS c
WHERE c.course_id = @courseID;
```

Syllabus Retrieval:

```
SELECT s.sec_id, s.title, s.description, s.order_number, s.allocated_time
FROM section AS s
WHERE s.course_id = @courseID
ORDER BY s.order_number;
```

3.4 Course Content Page



Relevant Information in the UI:

```
SELECT c.title, c.description
FROM course c
WHERE c.course_id = @courseID;
```

Query for the Student's Grade (the "Grades Label")

```
SELECT s.grade AS student_grade
FROM content c
LEFT JOIN submit s ON (c.course_id, c.sec_id, c.content_id, s.student_id) =
                        (s.course_id, s.sec_id, s.content_id, @studentID)
WHERE c.course_id = @courseID AND c.sec_id = @sectionID AND
      s.student_id = @studentID
ORDER BY c.content_id;
```

Whenever a new row is inserted into content, recalculate the total allocated_time for that section and update the section.allocated_time accordingly.

```
CREATE TRIGGER trig_content_alloc
REFERENCING NEW ROW AS nrow
AFTER INSERT ON content
FOR EACH ROW
BEGIN ATOMIC
    UPDATE section
    SET allocated_time = (
        SELECT SUM(c.allocated_time)
        FROM content c
```

```
        WHERE c.course_id = nrow.course_id AND c.sec_id = nrow.sec_id
    )
    WHERE section.course_id = nrow.course_id
    AND section.sec_id = nrow.sec_id;
END;
```

For the order of section numbers

```
SELECT  s.sec_id, s.title AS section_title, s.order_number
FROM section s
WHERE s.course_id = @courseID
ORDER BY s.order_number;
```

Content Retrieval

```
SELECT
  c.content_id, c.title AS content_title, vm.duration AS video_duration,
  t.task_type AS task_type
FROM content c
LEFT JOIN visual_material vm
   ON (c.course_id, c.sec_id, c.content_id) =
      (vm.course_id, vm.sec_id, vm.content_id)
LEFT JOIN document d
   ON (c.course_id, c.sec_id, c.content_id) =
      (d.course_id,d.sec_id,d.content_id)
LEFT JOIN task t
```

```
  ON (c.course_id, c.sec_id, c.content_id) =
     (t.course_id, t.sec_id, t.content_id)
LEFT JOIN assessment a
   ON (t.course_id, t.sec_id, t.content_id) =
      (a.course_id, a.sec_id, a.content_id)
LEFT JOIN assignment assign
   ON (t.course_id, t.sec_id, t.content_id) =
      (assign.course_id, assign.sec_id, assign.content_id)
WHERE c.course_id = courseID
AND c.sec_id = sectionID
ORDER BY c.content_id;
```

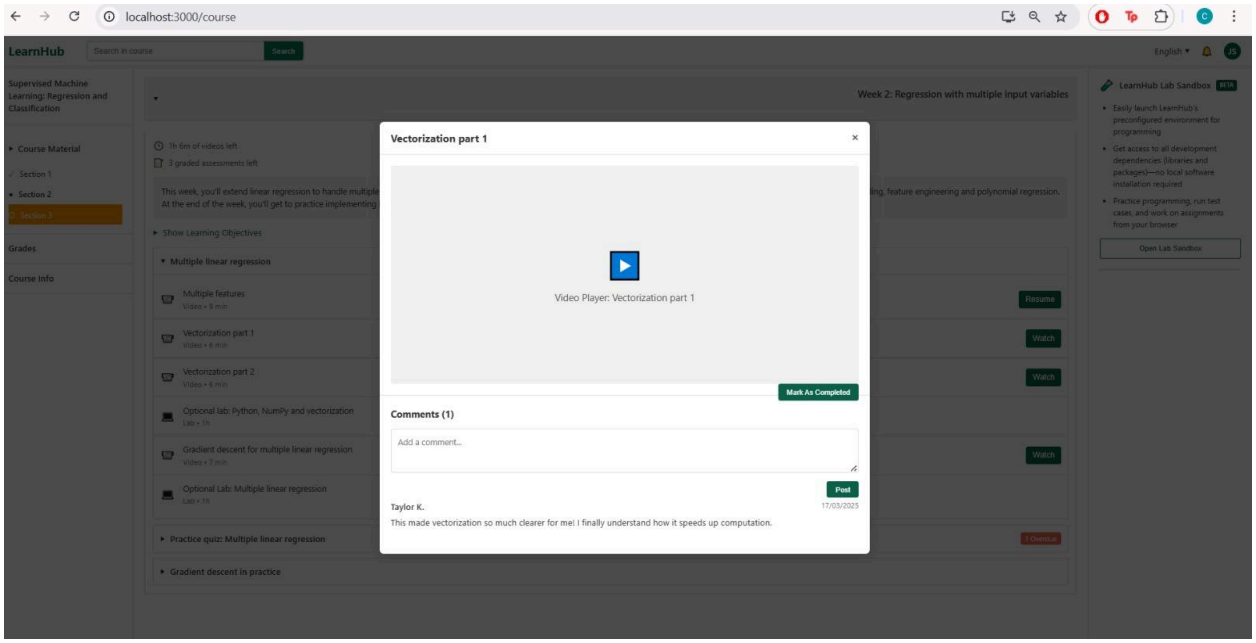
Next Deadlines For the Right Side

FOR overdue and upcoming contents

```
WITH overdue AS (
    SELECT a.content_id, ct.title  AS content_title, a.end_date
    FROM assignment a
    JOIN content ct ON  (a.course_id,a.sec_id,a.content_id) =
                        (ct.course_id, ct.sec_id,ct.content_id)
    WHERE a.course_id = courseID AND a.end_date < CURRENT_DATE()
    ORDER BY a.end_date
),
upcoming AS (
    SELECT a.content_id, ct.title AS content_title, a.end_date
```

```
    FROM assignment a
    JOIN content ct ON  a.course_id = ct.course_id
                        AND a.sec_id = ct.sec_id
                        AND a.content_id = ct.content_id
    WHERE a.course_id = courseID AND a.end_date >= CURRENT_DATE()
    ORDER BY a.end_date
)
SELECT 'upcoming' AS item_status, content_id, content_title, end_date
FROM upcoming
UNION ALL
SELECT
  'overdue' AS item_status,
  content_id, content_title, end_date
FROM overdue;
```

3.5 Content Page



Retrieve specific content:

```
SELECT c.title, c.allocated_time, c.content_type,
       d.body,
       vm.body, vm.duration,
       t.task_type, t.percentage, t.max_time, t.passing_grade,
       a.question_count,
       asgn.start_date, asgn.end_date, asgn.upload_material, asgn.body
FROM content c
LEFT JOIN document d ON (c.course_id, c.sec_id, c.content_id) =
                        (d.course_id,d.sec_id,d.content_id)
LEFT JOIN visual_material vm ON (c.course_id, c.sec_id, c.content_id) =
                                (vm.course_id, vm.sec_id, vm.content_id)
LEFT JOIN task t ON (c.course_id, c.sec_id, c.content_id) =
```

```
                    (t.course_id, t.sec_id, t.content_id)
LEFT JOIN assessment a ON (c.course_id, c.sec_id, c.content_id) =
                          (a.course_id, a.sec_id, a.content_id)
LEFT JOIN assignment asgn ON (c.course_id, c.sec_id, c.content_id) =
                              (asgn.course_id, asgn.sec_id, asgn.content_id)
WHERE c.course_id = @course_id AND c.sec_id = @sec_id
                  AND c.content_id = @content_id;
```

Retrieve comment for a specific content:

```
SELECT u.first_name, u.last_name, c.text, c.timestamp
FROM comment c JOIN user u ON u.ID = c.user_id
WHERE c.course_id = @course_id AND c.sec_id = @sec_id
      AND c.content_id = @content_id
ORDER BY c.timestamp DESC;
```

Retrieve comment count for a specific content:

```
SELECT COUNT(*)
FROM comment c
WHERE c.course_id = @course_id AND c.sec_id = @sec_id
      AND c.content_id = @content_id
```

Insert comment for a specific content:

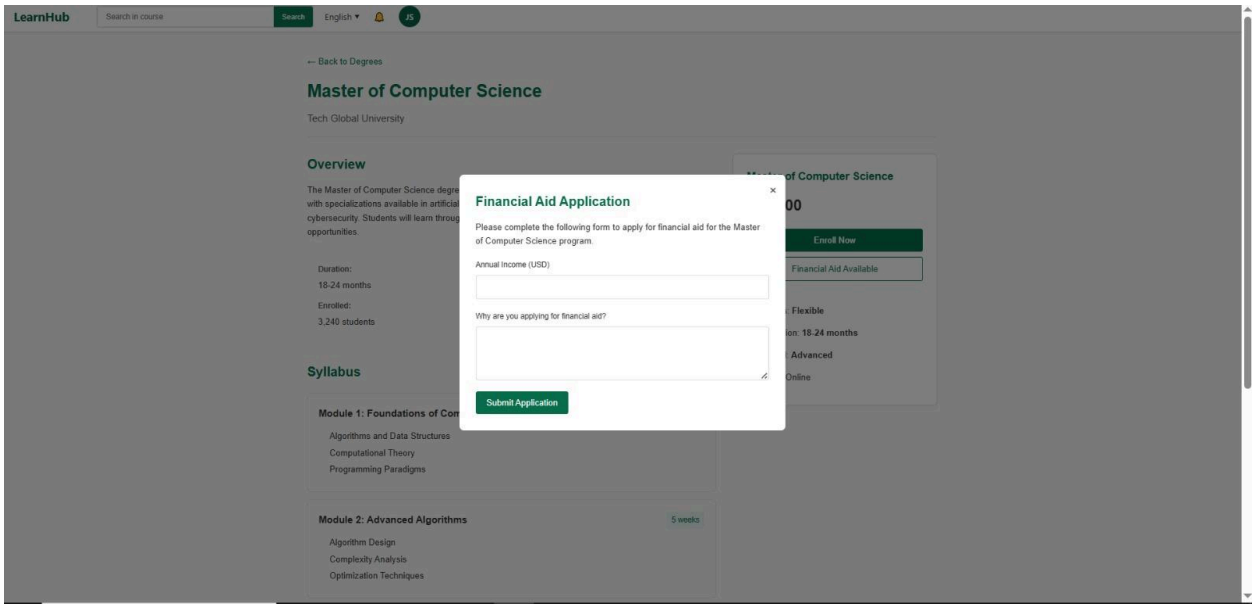
```
INSERT INTO comment
VALUES (@course_id, @sec_id, @content_id, @user_id, @text,
```

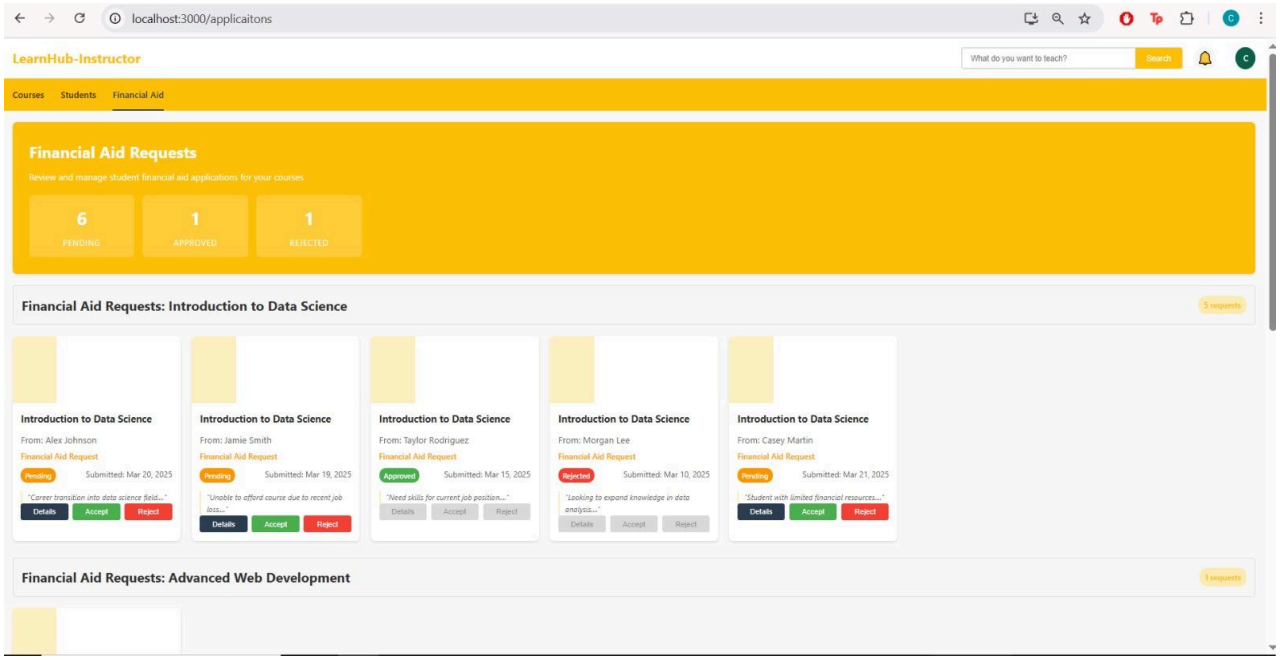
```
        CURRENT_TIMESTAMP );
```

Mark a specific content as completed:

```
UPDATE complete
SET is_completed = TRUE
WHERE complete.course_id = @course_id
        AND complete.sec_id = @sec_id
        AND complete.content_id = @content_id
        AND complete.student_id = @student_id
```

3.6 Financial Aid





Apply financial aid by student:

```
INSERT INTO apply_financial_aid (course_id,student_id,income,statement)
VALUES (@courseID, @studentID, @income, @text );
```

For the instructor page to observe applied students:

```
SELECT c.course_id, c.title AS course_title, afa.student_id, u.first_name AS
       student_first_name, u.last_name AS student_last_name, afa.statement,
       efa.is_accepted
FROM course AS c
JOIN apply_financial_aid AS afa
      ON c.course_id = afa.course_id
JOIN user AS u
      ON afa.student_id = u.ID
LEFT JOIN evaluate_financial_aid AS efa
```

```
      ON (afa.course_id,afa.student_id, efa.instructor_id) =
         (efa.course_id,efa.student_id,efa.creator_id)
WHERE c.creator_id = instructorID
ORDER BY c.title;
```

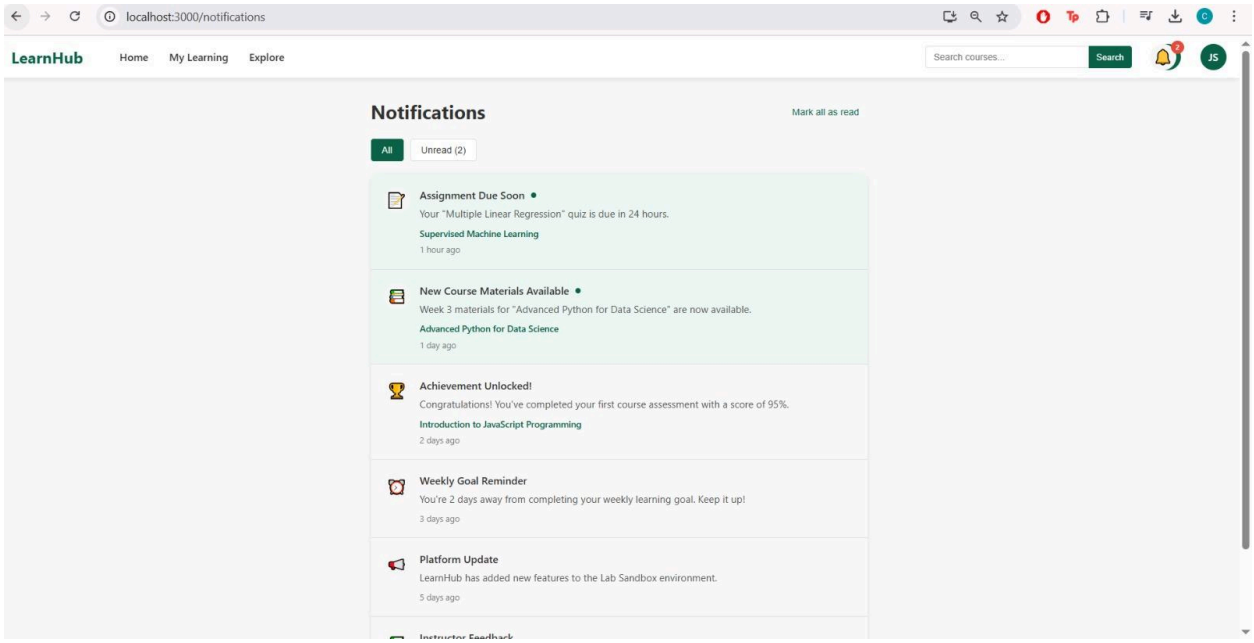
When the instructor accept or reject:

```
INSERT INTO evaluate_financial_aid (course_id, student_id, instructor_id,
                                    is_accepted)
VALUES (@courseID, @studentID, @instructorID, @boolValue)
```

Display Number of Accepted-Rejected-Pending:

```
SELECT
  SUM(CASE WHEN efa.is_accepted IS NULL THEN 1 ELSE 0 END) AS pending_count,
  SUM(CASE WHEN efa.is_accepted = TRUE THEN 1 ELSE 0 END) AS approved_count,
  SUM(CASE WHEN efa.is_accepted = FALSE THEN 1 ELSE 0 END) AS rejected_count
FROM course c
JOIN apply_financial_aid afa ON c.course_id = afa.course_id
LEFT JOIN evaluate_financial_aid efa
       ON afa.course_id  = efa.course_id AND afa.student_id = efa.student_id
WHERE c.creator_id = instructorID;
```

3.7 Notification Page



Retrieve all notifications for a specific user:

```
SELECT n
FROM notification n JOIN receive r ON n.notification_id = r.notification_id
WHERE r.ID = @userID
ORDER BY n.timestamp DESC;
```

Retrieve unread notifications for a specific user:

```
SELECT n
FROM notification n JOIN receive r ON n.notification_id = r.notification_id
WHERE r.ID = @userID AND n.status = 'unread'
ORDER BY n.timestamp DESC;
```

Mark all as read notifications for a specific user:

```
UPDATE notification
SET status = 'read'
WHERE notification_id IN (
    SELECT r.notification_id
    FROM receive r
    WHERE r.ID = @userID
) AND status = 'unread';
```

Search in notifications:

```
SELECT n
FROM notification n JOIN receive r ON n.notification_id = r.notification_id
WHERE r.ID = @userID AND (
                          n.type LIKE '%' || @searchTerm || '%' OR
                          n.message LIKE '%' || @searchTerm || '%' )
ORDER BY n.timestamp DESC;
```

Insert a new notification and send it to a user:

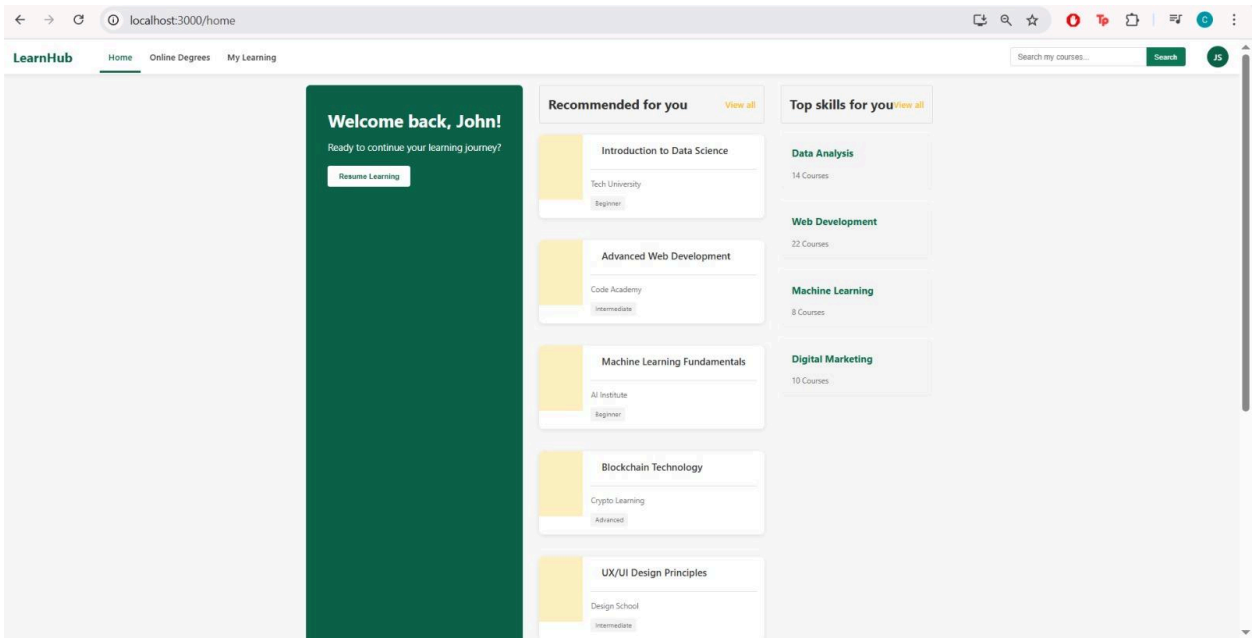
```
INSERT INTO notification (notification_id, type, entity_type, message,
                                                    timestamp, status)
VALUES (@notificationID, @type, @entity_type', @message, NOW(), 'unread');
```

Create a trigger for receive when notification is updated:

```
CREATE TRIGGER trg_for_notification
```

```
REFERENCING NEW ROW AS new
AFTER INSERT ON notification
FOR EACH ROW
BEGIN
    INSERT INTO receive (notification_id, ID)
    VALUES (new.notification_id, @userID);
END;
```

3.8 Home Page



Retrieve name and email:

```
SELECT first_name, middle_name, last_name, email
FROM user
WHERE ID = @userID;
```

Create a view for recommended course cards:

```
CREATE VIEW recommended_course AS (
    SELECT c.title, c.category, c.difficulty_level, c.enrollment_count, 1 AS
                                                                    priority
    FROM course c
    WHERE c.category IN (
        SELECT DISTINCT c2.category
        FROM enroll e JOIN course c2 ON e.course_id = c2.course_id
        WHERE e.student_id = @studentID
    )
    AND c.course_id NOT IN (
        SELECT course_id
        FROM enroll
        WHERE student_id = @studentID
    )
    AND c.is_approved = TRUE
)
UNION
(
    SELECT c.title, c.category, c.difficulty_level, c.enrollment_count, 2 AS
                                                                    priority
    FROM course c
    WHERE c.is_approved = TRUE AND c.course_id NOT IN (
```

```
        SELECT course_id
        FROM enroll
        WHERE student_id = @studentID
    )
)
```

Display all recommended course cards:

```
SELECT title, category, difficulty_level, enrollment_count
FROM recommended_course
ORDER BY priority, enrollment_count DESC, course_id;
```

Display 10 recommended course cards:

```
SELECT title, category, difficulty_level, enrollment_count
FROM recommended_course
ORDER BY priority, enrollment_count DESC, course_id
LIMIT 10;
```

Search in recommended course cards:

```
SELECT title, category, difficulty_level, enrollment_count
FROM recommended_course
WHERE title LIKE '%' || @searchTerm || '%'
      OR category LIKE '%' || @searchTerm || '%'
ORDER BY priority, enrollment_count DESC, course_id;
```

Create a view for recommended categories:

```
CREATE VIEW recommended_category AS
(
    SELECT c.category, COUNT(*) AS course_count, 1 AS priority
    FROM course c
    WHERE c.is_approved = TRUE
      AND c.category IN (
          SELECT DISTINCT c2.category
          FROM enroll e JOIN course c2 ON e.course_id = c2.course_id
          WHERE e.student_id = @studentID
      )
    GROUP BY c.category
)
UNION
(
    SELECT c.category, COUNT(*) AS course_count, 2 AS priority
    FROM course c
    WHERE c.is_approved = TRUE
    GROUP BY c.category
)
```

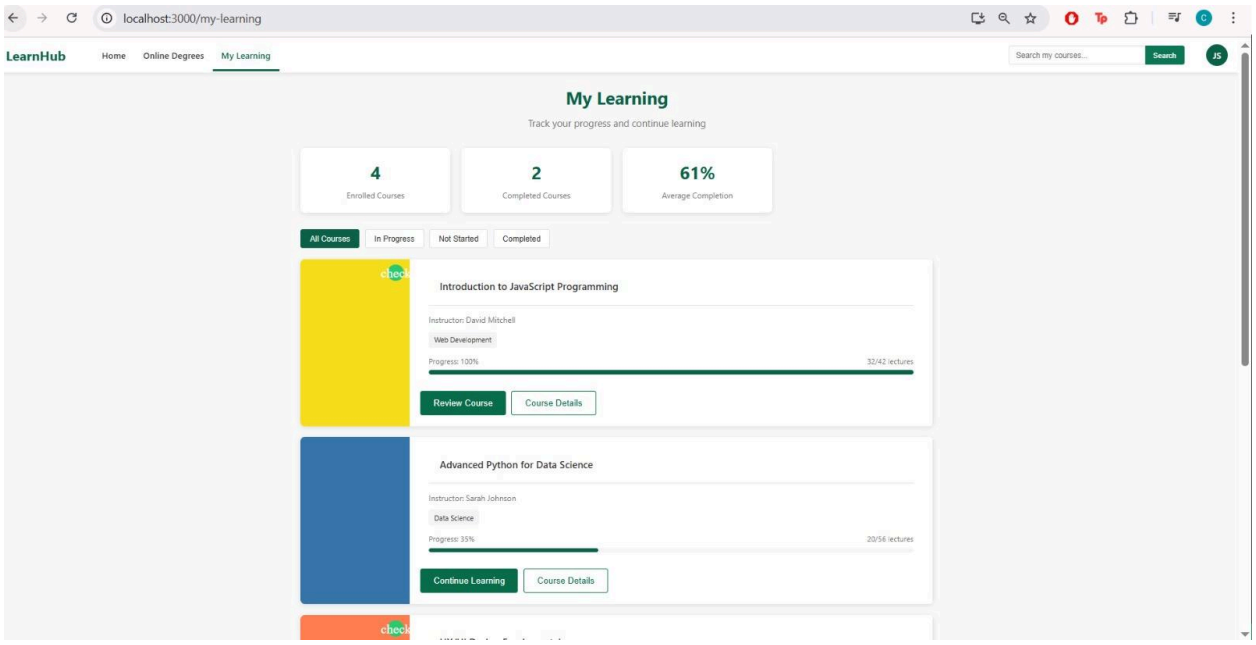
Display all recommended categories:

```
SELECT category, course_count
FROM recommended_category
ORDER BY priority, course_count DESC;
```

Display 5 recommended categories:

```
SELECT category, course_count
FROM recommended_category
ORDER BY priority, course_count DESC
LIMIT 5;
```

3.9 My Learning Page



Retrieve enrolled course count:

```
SELECT COUNT(*) AS enrolled_course_count
FROM enroll
WHERE student_id = @studentID;
```

Retrieve completed course count:

```
SELECT COUNT(*) AS completed_course_count
FROM enroll
WHERE student_id = @studentID AND progress_rate = 100;
```

Retrieve average course completion rate:

```
SELECT AVG(progress_rate) AS average_completion
FROM enroll
WHERE student_id = @studentID;
```

Create a view for the total and completed content count of courses:

```
CREATE VIEW course_content_count AS
SELECT e.student_id, c.course_id,
      SUM(CASE WHEN ct.content_id IS NULL THEN 0 ELSE 1 END) AS
                                                    total_content_count,
      SUM(CASE WHEN ct.is_completed = TRUE THEN 1 ELSE 0 END) AS
                                                    completed_content_count
FROM enroll e
```

```
        LEFT JOIN course c ON e.course_id = c.course_id
        LEFT JOIN section s ON s.course_id = c.course_id
        LEFT JOIN content ct ON ct.course_id = s.course_id AND ct.sec_id =
                                                                    s.sec_id
        LEFT JOIN complete cmp
            ON cmp.course_id = ct.course_id
           AND cmp.sec_id = ct.sec_id
           AND cmp.content_id = ct.content_id
           AND cmp.student_id = e.student_id
           AND cmp.is_completed = TRUE
GROUP BY e.student_id, c.course_id;
```

Retrieve all courses:

```
SELECT c.course_id, c.title, c.category, e.progress_rate,
       u.first_name, u.middle_name,  u.last_name,
       ccc.total_content_count, ccc.completed_content_count
FROM enroll e
       JOIN course c ON e.course_id = c.course_id
       JOIN instructor i ON c.creator_id = i.ID
       JOIN user u ON u.ID = i.ID
       JOIN course_content_count ccc ON c.course_id = ccc.course_id
             AND e.student_id = ccc.student_id
WHERE e.student_id = @studentID;
```

Search in all courses:

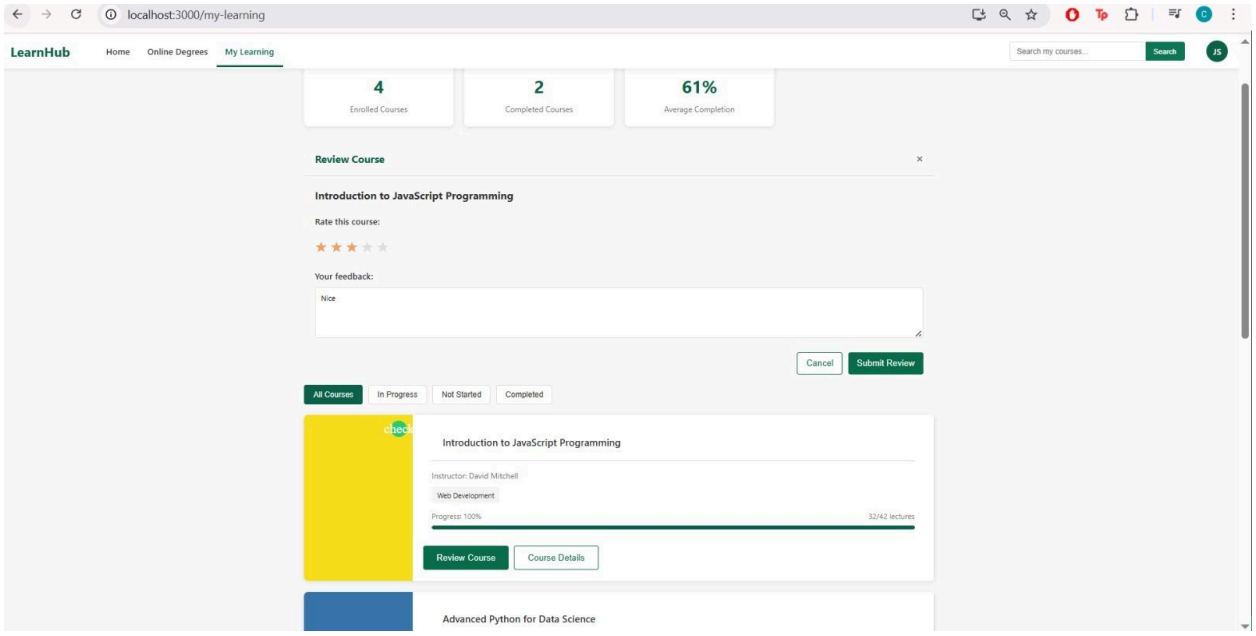
```
SELECT c.course_id, c.title, c.category, e.progress_rate,
      u.first_name, u.middle_name,  u.last_name,
      ccc.total_content_count, ccc.completed_content_count
FROM enroll e
      JOIN course c ON e.course_id = c.course_id
      JOIN instructor i ON c.creator_id = i.ID
      JOIN user u ON u.ID = i.ID
      JOIN course_content_count ccc ON c.course_id = ccc.course_id
            AND e.student_id = ccc.student_id
WHERE e.student_id = @studentID AND ( c.title LIKE '%' || @searchTerm || '%'
                                    OR c.category LIKE '%' || @searchTerm ||
'%' );
```

Trigger progress_rate when a new content is completed:

```
CREATE TRIGGER trg_update_progress_rate
AFTER UPDATE ON complete
REFERENCING NEW ROW AS newrow
FOR EACH ROW
WHEN (newrow.is_completed = TRUE)
BEGIN
      UPDATE enroll
      SET progress_rate = (SELECT
                           CASE
                               WHEN COUNT(DISTINCT ct.content_id) = 0 THEN 0
```

```
                    ELSE (100 * COUNT(DISTINCT cmp.content_id) /
                        COUNT(DISTINCT ct.content_id) )
                END
            FROM content ct LEFT JOIN complete cmp
                ON cmp.course_id = ct.course_id
                    AND cmp.sec_id = ct.sec_id
                    AND cmp.content_id = ct.content_id
                    AND cmp.student_id = newrow.student_id
                    AND cmp.is_completed = TRUE
            WHERE ct.course_id = newrow.course_id )
    WHERE enroll.course_id = newrow.course_id
        AND enroll.student_id = newrow.student_id;
END;
```

3.10 Feedback Page



Insert feedback from a student for a specific course:

```
INSERT INTO feedback (course_id, student_id, rating, comment, feedback_date)
      VALUES (@course_id, @student_id, @rating, @comment, CURRENT_DATE);
```

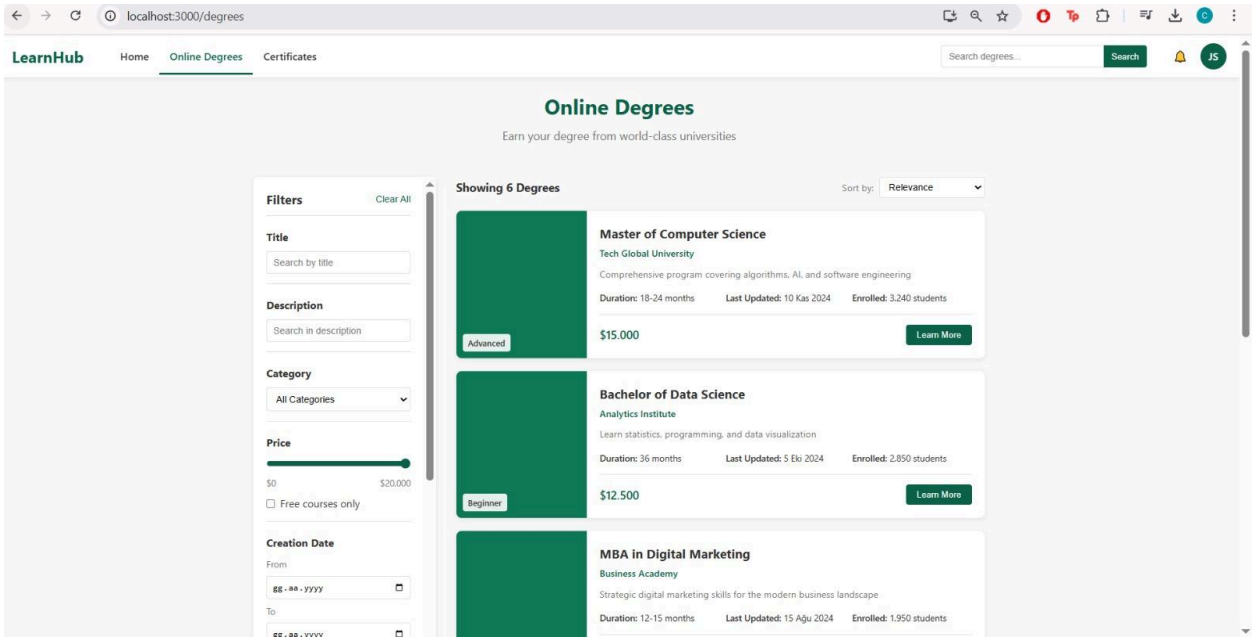
Retrieve feedback for a specific course:

```
SELECT f.rating, f.comment, f.feedback_date, u.first_name, u.last_name
FROM feedback f
      JOIN student s ON f.student_id = s.ID
      JOIN user u ON u.ID = s.ID
WHERE f.course_id = @course_id
ORDER BY f.feedback_date DESC;
```

Retrieve average rating for a course:

```
SELECT course_id, ROUND(AVG(rating), 2) AS avg_rating, COUNT(*) AS
                                                          total_reviews
FROM feedback
WHERE course_id = @course_id;
```

3.11 Online Degrees Page



Retrieve online degree courses:

```
SELECT c.title, c.description, c.category, c.price, c.difficulty_level,
      c.creation_date, c.last_update_date, c.enrollment_count,
      u.first_name, u.middle_name, u.last_name
FROM course c
      JOIN instructor i ON c.creator_id = i.ID
      JOIN user u ON u.ID = i.ID
WHERE c.is_approved = TRUE;
```

Conditions for filters:

Title:

```
AND c.title LIKE '%' || @searchTitle || '%'
```

Description:

```
AND c.description LIKE '%' || @searchDescription || '%'
```

Category:

```
AND c.category = @selectedCategory
```

Price:

```
AND c.price BETWEEN @minPrice AND @maxPrice
```

Free Courses Only:

```
AND c.price = 0
```

Creation Date:

```
AND c.creation_date BETWEEN @startDate AND @endDate
```

Orders for sorting:

Title:

```
ORDER BY c.title
```

Price:

```
ORDER BY c.price
```

Popularity:

```
ORDER BY c.enrollment_count DESC
```

Creation Date:

```
ORDER BY c.creation_date DESC
```

Search in online degree courses:

```
SELECT c.title, c.description, c.category, c.price, c.difficulty_level,
       c.creation_date, c.last_update_date, c.enrollment_count, u.first_name,
       u.middle_name, u.last_name
FROM course c
```

```
    JOIN instructor i ON c.creator_id = i.ID
    JOIN user u ON u.ID = i.ID
WHERE c.is_approved = TRUE
    AND (
        c.title LIKE '%' || @searchTerm || '%'
        OR c.category LIKE '%' || @searchTerm || '%'
        OR c.description LIKE '%' || @searchTerm || '%'
    );
```

4. Implementation Plan

The project will be implemented using **PostgreSQL** as the database management system since it supports all required modern features, such as views, triggers, and constraints. All database operations will be performed using **raw SQL queries**, as required.

The backend will be built using **Spring Boot (Java)**. It will handle the main logic of the system and connect to the PostgreSQL database. We'll use **raw SQL queries** to interact with the database instead of tools like Hibernate, which generate queries automatically.

The frontend will be built with **React**, providing a responsive and component-based user interface that communicates with the backend via RESTful APIs.