Faculty of Education

Department of Computer Educational Technology

CET321 - Final project

Instructor: Oğuz Ak

Student Name: Tuğba Aslandere

Student ID: 2021100039

Date: 01 December 2025

Table of Contents

Project definition

CampusConnect is a web-based platform that helps university students support each other in their courses. Students can ask questions about a course, its content, exams, or even the instructor. Other students who already took that course can answer and share their experiences. When their answers are approved, they earn a small amount of pocket money.

In addition, students can upload and sell their own lecture notes safely through the platform.All transactions are recorded in the system, and payments are handled automatically.

The main purpose of this project is to create a useful, fair, and reliable system where students can both get academic help and earn small rewards by helping others.

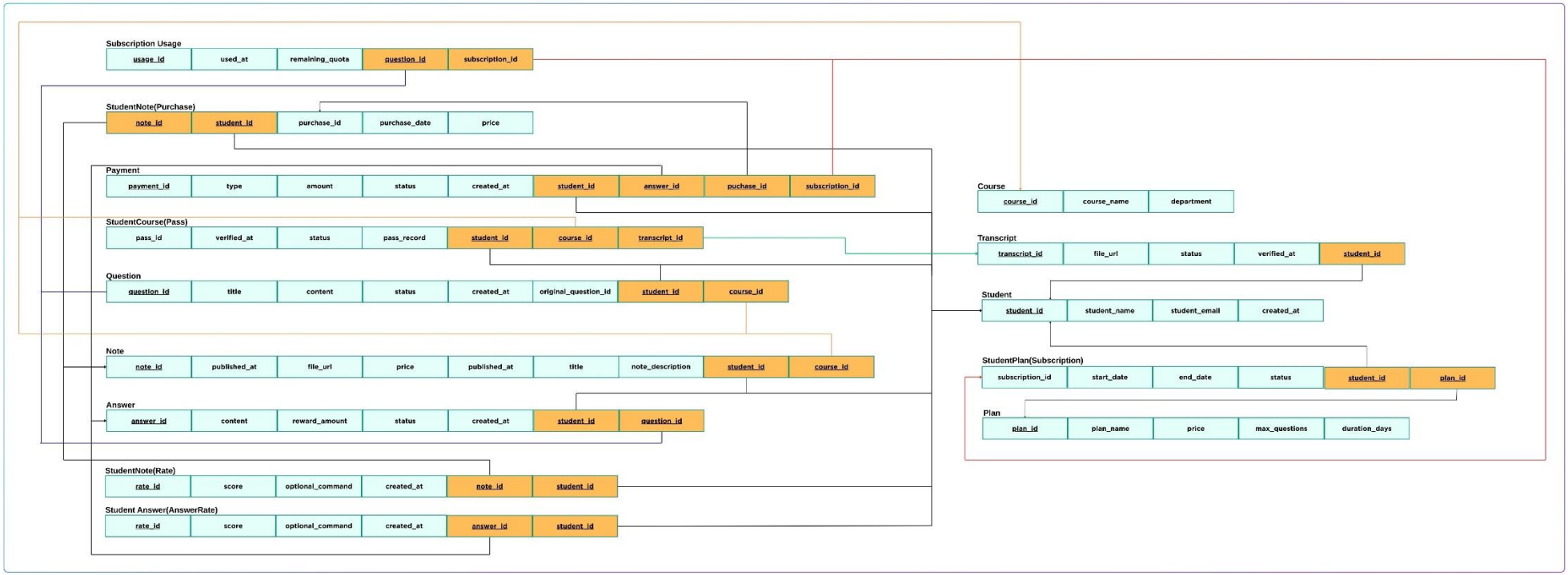
Updated Business Rules

- A student uploads one and only one transcript.
- Each transcript is uploaded by one and only one student.
- A student can pass zero, one, or many courses.
- Each course can be passed by zero, one, or many students.
- A pass record is derived from one and only one transcript.
- Each transcript can verify zero, one, or many pass records.
- A student can subscribe to zero or one plan at a time.
- Each plan can be subscribed to by zero, one, or many students.
- Each subscription has one start date, one end date, and one status.
- Each plan defines one price, one question limit, and one duration.
- Each subscription creates one and only one payment.
- Each payment is created by one and only one subscription.
- Each subscription belongs to one and only one plan.
- A plan can be subscribed to by zero, one, or many students,
- A student can ask zero, one, or many questions.
- Each question is asked by one and only one student.
- A course can include zero, one, or many questions.
- Each question belongs to one and only one course.
- A question can have zero, one, or many answers.
- Each answer belongs to one and only one question.

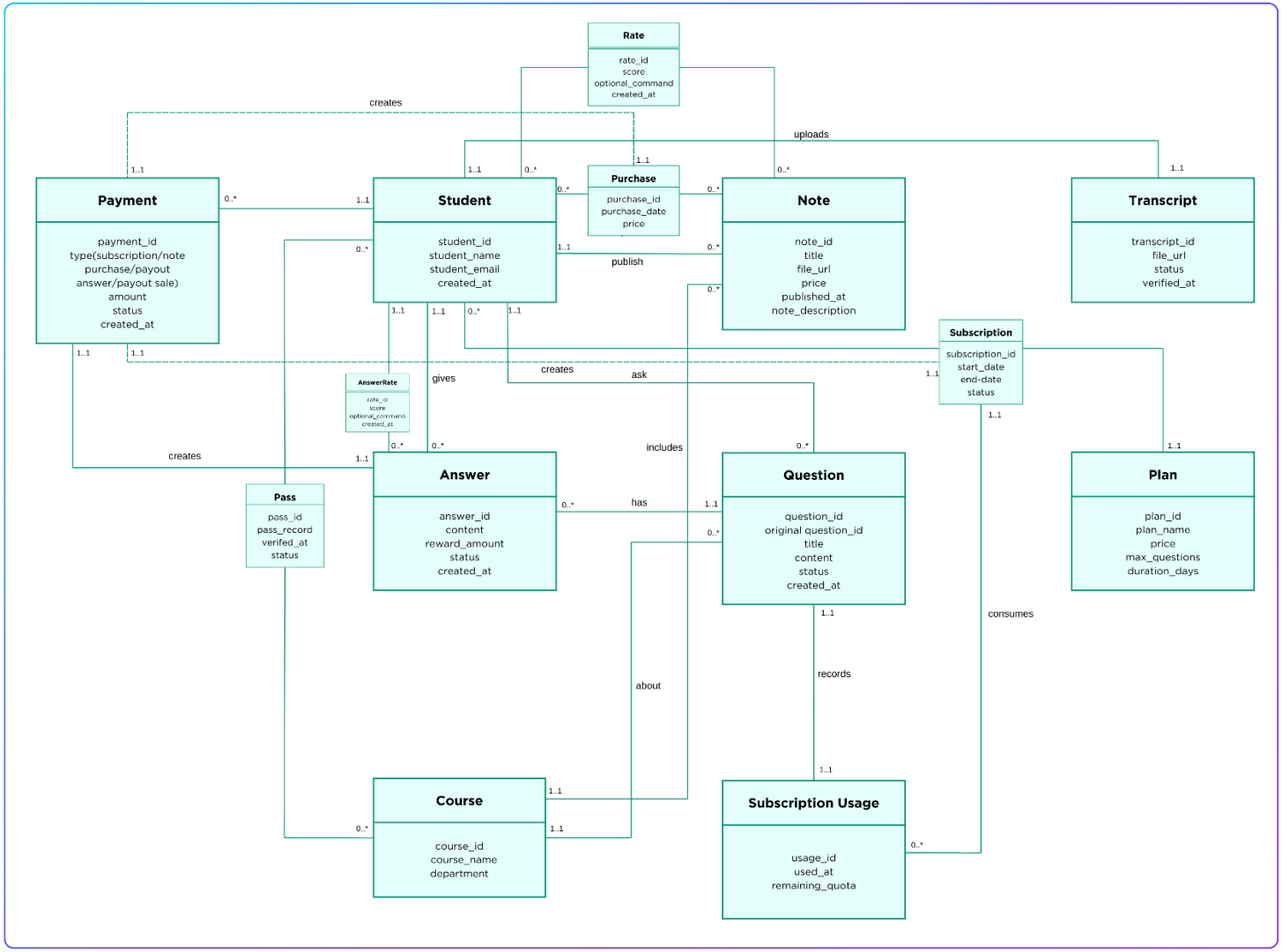
- A student can give zero, one, or many answers.
- Each answer is given by one and only one student.
- A question can be re-asked zero, one, or many times.
- Each re-asked question is linked to one and only one original question.
- A student can upload zero, one, or many notes.
- Each note is uploaded by one and only one student.
- A course can include zero, one, or many notes.
- Each note belongs to one and only one course.
- A student can rate zero, one, or many answers.
- Each answer can have zero, one, or many ratings
- A student can rate zero, one, or many notes.
- Each note rating is given by one and only one student.
- Each note rating belongs to one and only one note
- A student can rate a specific note at most once.
- Each note can have zero, one, or many ratings.
- The system can calculate an average rating value for each note based on all ratings given by students.
- A note can be purchased by zero, one, or many students.
- Each purchase is made for one and only one note.
- Each purchase creates two payments: one for the buyer and one for the seller.
- Each payment is created by one and only one purchase.
- A verified answer creates one and only one payment.
- Each payment is created by one and only one verified answer.
- A student can have zero, one, or many payments.
- Each payment belongs to one and only one student.
- A subscription can include zero, one, or many question usages.
- Each question usage belongs to one and only one subscription.
- Each question usage records one and only one question.

- Each question can be recorded in zero, one, or many usage records.

Updated Logical diagram of the database



Updated Class diagram of the database



SQL Operations for Main Functions of the Database

The SQL script starts by creating the database and then creating all tables. In the CREATE TABLE parts, I used primary keys, NOT NULL rules, unique emails, and some CHECK constraints (for example, price cannot be negative). I also connected tables with foreign keys so the relationships (like student–note, question–course) are consistent.

After the tables, I inserted sample data with INSERT INTO statements (students, courses, plans, transcripts, notes, questions, and answers).

Then I created two views to make reporting easier. One view shows note rating information (average score and how many ratings). The other view shows questions with how many answers they have.

I also wrote two stored procedures for common actions. One procedure creates a subscription and also creates the related payment. The other procedure creates a purchase record for a note.

Finally, I added two triggers to automate payments. When someone buys a note, the trigger creates a buyer payment and a seller payout automatically. When an answer becomes verified, another trigger creates the answer payout payment. At the end, I included example SELECT queries and sample UPDATE/DELETE statements to show the main functions.

SQL Operations for Main Functions

DDL Example 1

This table stores basic student accounts and ensures that each email address is unique.

```
CREATE TABLE dbo.Student (
    student_id     INT IDENTITY(1,1) PRIMARY KEY,
    student_name   NVARCHAR(100) NOT NULL,
    student_email  NVARCHAR(150) NOT NULL UNIQUE,
    created_at     DATETIME2 NOT NULL DEFAULT SYSDATETIME()
);
```

DDL Example 2

This table stores uploaded notes and connects each note to a student and a course using foreign keys.

```
CREATE TABLE dbo.Note (
    note_id       INT IDENTITY(1,1) PRIMARY KEY,
    student_id    INT NOT NULL,
    course_id     INT NOT NULL,
    title         NVARCHAR(150) NOT NULL,
    file_url      NVARCHAR(300) NULL,
```

```
    price         DECIMAL(10,2) NOT NULL CHECK (price >= 0),
    published_at  DATETIME2 NULL,
    note_description NVARCHAR(300) NULL,
    CONSTRAINT FK_Note_Student FOREIGN KEY (student_id)
        REFERENCES dbo.Student(student_id),
    CONSTRAINT FK_Note_Course FOREIGN KEY (course_id)
        REFERENCES dbo.Course(course_id)
);
```

INSERT Example 1

This example inserts sample students into the database.

```
INSERT INTO dbo.Student(student_name, student_email) VALUES
(N'Tuğba Aslandere', N'tugba@example.com'),
(N'Ayşe Demir',     N'ayse@example.com'),
(N'Mehmet Yılmaz',  N'mehmet@example.com');
```

INSERT Example 2

This example inserts uploaded notes by different students for various courses.

```
INSERT INTO dbo.Note(student_id, course_id, title, file_url, price, published_at,
note_description) VALUES
(1, 1, N'Normalization Cheat Sheet', N'link://note1.pdf', 25.00, SYSDATETIME(), N'1NF-2NF-3NF-
BCNF'),
(2, 1, N'SQL Joins Summary',         N'link://note2.pdf', 30.00, SYSDATETIME(), N'JOIN
examples'),
(3, 3, N'Linear Algebra Notes',      N'link://note3.pdf', 20.00, SYSDATETIME(), N'Matrices and
vectors');
```

UPDATE Example 1

Updates an existing student's email address (similar to an "account update").

```
UPDATE dbo.Student
SET student_email = N'tugba_updated@example.com'
WHERE student_id = 1;
```

UPDATE Example 2

Adjusts the price of a note (for example, if the seller increases the price).

```
UPDATE dbo.Note
SET price = price + 5
WHERE note_id = 1;
```

DELETE Example 1

This transaction deletes a rating and rolls back to preserve data integrity.

```
BEGIN TRAN;
    DELETE FROM dbo.NoteRating
    WHERE student_id = 3 AND note_id = 1;
ROLLBACK;
```

DELETE Example 2

This transaction safely tests a delete without committing changes.

```
BEGIN TRAN;
    DELETE FROM dbo.SubscriptionUsage
    WHERE usage_id = 1;
ROLLBACK;
```

SELECT 1

Shows the questions asked by a student and related answers using multi-table joins.

```
SELECT s.student_name, q.title AS question_title, a.content AS answer_content
FROM dbo.Student s
JOIN dbo.Question q ON q.student_id = s.student_id
LEFT JOIN dbo.Answer a ON a.question_id = q.question_id
WHERE s.student_id = 1;
```

SELECT 2

Displays active subscriptions along with plan details using filtering and joins.

```
SELECT s.student_name, p.plan_name, p.max_questions, sub.start_date, sub.end_date, sub.status
FROM dbo.Student s
JOIN dbo.Subscription sub ON sub.student_id = s.student_id
JOIN dbo.[Plan] p ON p.plan_id = sub.plan_id
WHERE sub.status = 'active';
```

SELECT 3

Uses a view to show average note ratings per course.

```
SELECT *
FROM dbo.vw_NoteRatingSummary
WHERE course_name = N'Database Systems';
```

SELECT 4

Lists purchases and related payment records for a specific buyer.

```
SELECT s.student_name, n.title, pr.purchase_date, pay.payment_type, pay.amount
FROM dbo.Purchase pr
JOIN dbo.Student s ON s.student_id = pr.student_id
JOIN dbo.Note n ON n.note_id = pr.note_id
JOIN dbo.Payment pay ON pay.purchase_id = pr.purchase_id
WHERE pr.student_id = 1
ORDER BY pr.purchase_date DESC;
```

View 1

This view shows each note's average rating and total number of ratings.

```
CREATE VIEW dbo.vw_NoteRatingSummary
AS
SELECT
    n.note_id,
    n.title,
    c.course_name,
    s.student_name AS uploader_name,
    n.price,
    AVG(CAST(nr.score AS DECIMAL(10,2))) AS avg_score,
    COUNT(nr.note_rating_id) AS rating_count
FROM dbo.Note n
JOIN dbo.Course c ON c.course_id = n.course_id
JOIN dbo.Student s ON s.student_id = n.student_id
LEFT JOIN dbo.NoteRating nr ON nr.note_id = n.note_id
GROUP BY n.note_id, n.title, c.course_name, s.student_name, n.price;
```

View 2

This view shows how many answers each question has.

```
CREATE VIEW dbo.vw_QuestionAnswerSummary
AS
SELECT
    q.question_id,
    q.title,
    c.course_name,
    st.student_name AS asked_by,
    q.status,
```

```
    q.created_at,
    COUNT(a.answer_id) AS answer_count
FROM dbo.Question q
JOIN dbo.Course c ON c.course_id = q.course_id
JOIN dbo.Student st ON st.student_id = q.student_id
LEFT JOIN dbo.Answer a ON a.question_id = q.question_id
GROUP BY q.question_id, q.title, c.course_name, st.student_name, q.status, q.created_at;
```

Stored Procedure 1

Creates a subscription for a student and generates a payment automatically.

```
CREATE PROCEDURE dbo.sp_SubscribeStudent
    @student_id INT,
    @plan_id INT,
    @start_date DATE = NULL
AS
BEGIN
    INSERT INTO dbo.Subscription(student_id, plan_id, start_date, end_date, status)
    VALUES (@student_id, @plan_id, @start_date, DATEADD(DAY, 30, @start_date), 'active');

    INSERT INTO dbo.Payment(student_id, payment_type, amount, status, subscription_id)
    VALUES (@student_id, 'subscription', 49.90, 'paid', SCOPE_IDENTITY());
END
```

Stored Procedure 2

Inserts a purchase record for a note. The related payments are handled by triggers.

```
CREATE PROCEDURE dbo.sp_PurchaseNote
    @buyer_student_id INT,
    @note_id INT
AS
BEGIN
    DECLARE @price DECIMAL(10,2) = (SELECT price FROM dbo.Note WHERE note_id = @note_id);
    INSERT INTO dbo.Purchase(student_id, note_id, price)
    VALUES (@buyer_student_id, @note_id, @price);
END
```

Trigger 1

Automatically creates buyer and seller payment records after a note is purchased.

```
CREATE TRIGGER dbo.trg_Purchase_CreatePayments
ON dbo.Purchase
AFTER INSERT
AS
BEGIN
    INSERT INTO dbo.Payment(student_id, payment_type, amount, status, purchase_id)
    SELECT i.student_id, 'note_purchase', i.price, 'paid', i.purchase_id
    FROM inserted i;

    INSERT INTO dbo.Payment(student_id, payment_type, amount, status, purchase_id)
    SELECT n.student_id, 'note_payout', i.price, 'paid', i.purchase_id
    FROM inserted i
    JOIN dbo.Note n ON n.note_id = i.note_id;
END
```

Trigger 2

Creates a payment automatically when an answer status becomes verified.

```
CREATE TRIGGER dbo.trg_Answer_Verified_CreatePayment
ON dbo.Answer
AFTER UPDATE
AS
BEGIN
    INSERT INTO dbo.Payment(student_id, payment_type, amount, status, answer_id)
    SELECT i.student_id, 'answer_payout', i.reward_amount, 'paid', i.answer_id
    FROM inserted i
    JOIN deleted d ON d.answer_id = i.answer_id
    WHERE i.status = 'verified' AND d.status <> 'verified';
END
```