Educational content presentation platform
DB Design

Contents

PART I: Analysis

1 Problem Definition and Data Requirements

1.1 Problem Description

We aim to develop a smart course platform that enables dynamic management of user accounts, course offerings, subscriptions, and learning progress. The system supports detailed tracking of student enrollment, assessment performance, and certificate issuance, while providing university-level administrative functionalities. Teachers benefit from automated earning calculations based on pay rate and enrolled student count, and can coordinate courses and assign learning outcomes. Subscriptions are managed with validity checks, status updates, and plan associations. Administrative features include logging of account changes and enforcement of business rules through constraints and triggers.

1.2 Data Requirements

- User account information including identity and subscription status
- Subscription status domain
- Physical address linked to accounts
- University data for affiliation of instructors
- Teacher employment, pay rate, university, and linked account
- Earnings record based on coordinated course engagement
- Certificate templates and digital signature storage
- Certificate issuance logs per user
- Subscriptions with duration and pricing
- Plan type per subscription
- Course details including pricing, content, coordinator, and certificate
- Domain-based course levels
- Enrollment tracking per user and course
- Payment tracking for course access
- Learning progress monitoring per user per course

- Assessment structure per course
- Type classification per assessment
- Teacher-course assignments
- Logging of changes to account data for audit purposes

1.3 Business Rules

- Passwords must be stored securely using SHA-256 hashing.
- Each user is allowed only one active subscription and must be linked to one subscription plan.
- All courses must be assigned a coordinator , and the coordinator must be one of the teachers teaching the course. The course should also be associated with a certificate.
- Students must have a valid payment status before enrollment is confirmed.
- Teacher earnings are calculated based on the number of students enrolled, the course price, and the pay rate.
- Subscription periods must have a valid start and end date with Sub_End > Sub_Start enforced.
- Teachers must be affiliated with a university.
- Each certificate must have a digital signature stored securely (as CLOB due to the fact that the signature is hashed using SHA-256).
- Assessment scores must be bounded by pass score requirements and point validations.
- Course enrollment automatically updates the course's user count via trigger.
- Any modification to account data must be logged in a change log table for audit purposes.

1.4 Intended Output of the system

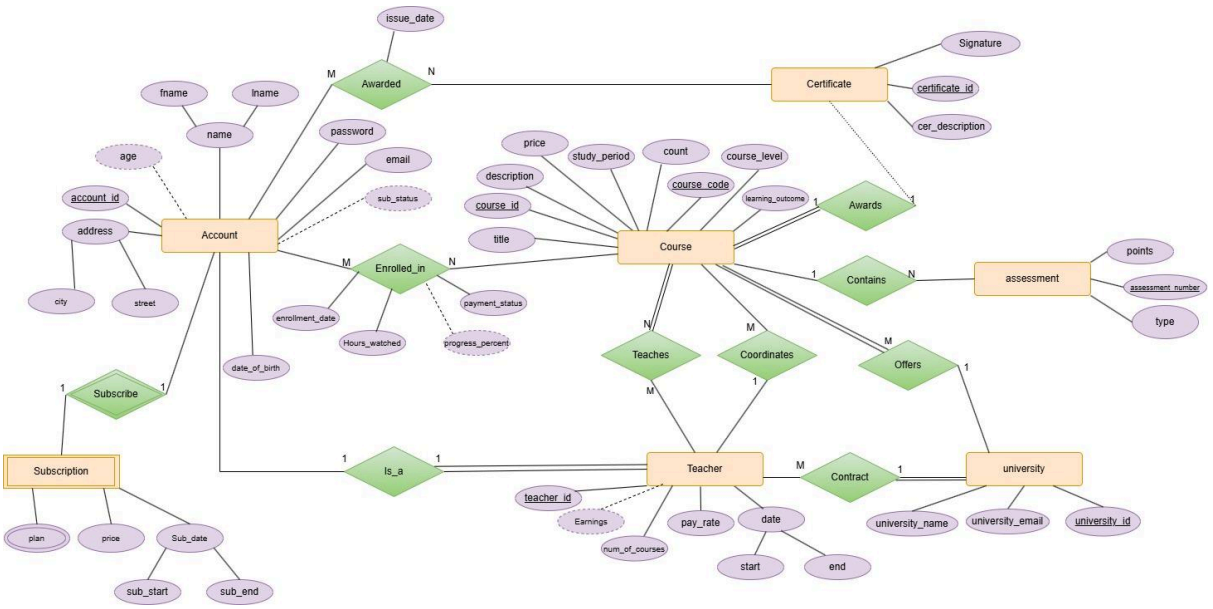
- View detailed student progress in each enrolled course, including percentage completed.
- Display calculated earnings for each teacher based on enrolled students and course pricing.
- List enrolled students with enrollment dates and payment status.

- Provide certificate validation including certificate details, description, associated course, and university.
- Show users with currently active subscriptions based on subscription date range.
- Track changes made to user accounts, including updates and deletions, for auditing purposes.
- Allow visualization of assessments assigned to students and their associated pass scores.
- Provide teachers with a list of courses they are coordinating or teaching.
- Summarize subscription plans and pricing models linked to each user.
- Support administrative queries for monitoring enrollment numbers across all courses.

PART II: DB DESIGN

2 ER Diagram Design

2.1 ER diagram



2.2 Design of Business Rules

Business Rule	Design Decisions	Justification
Passwords are stored securely	VARCHAR(64), hashed using SHA-256	Ensures data confidentiality and aligns with security best practices
Subscription status uses domain table	Foreign key to Subscription_Status	Promotes normalization and reduces data redundancy
Students can earn certificates	Awarded table (many-to-many relationship)	Captures relationship between accounts and certificates
Teachers have earnings	Earnings table, foreign key to Teacher	Enables storage and reporting of calculated earning values
Enrollment affects user count	Triggers on Enrolled_In to update Count_USR	Automates course enrollment tracking and ensures data consistency
changes to the tables are tracked	Triggers inserts into the change_log	Supports system auditing and change history monitoring
Subscription periods must be valid	CHECK constraint on Sub_End > Sub_Start	Prevents logical errors and enforces valid subscription duration
Certificate must include a signature	Signature stored as CLOB or base64	Ensures certificate authenticity and secure representation
Teachers must belong to a university	Foreign key from Teacher to University	Reflects academic structure and ensures referential integrity
Valid payment status required	Foreign key in Enrolled_In to Payment_Statuses	Ensures payment data integrity during enrollment

Business Rule	Design Decisions	Justification
Passwords are stored securely	VARCHAR(64), hashed using SHA-256	Ensures data confidentiality and aligns with security best practices
Subscription status uses domain table	Foreign key to Subscription_Status	Promotes normalization and reduces data redundancy
Students can earn certificates	Awarded table (many-to-many relationship)	Captures relationship between accounts and certificates
Teachers have earnings	Earnings table, foreign key to Teacher	Enables storage and reporting of calculated earning values
Enrollment affects user count	Triggers on Enrolled_In to update Count_USR	Automates course enrollment tracking and ensures data consistency
changes to the tables are tracked	Triggers inserts into the change_log	Supports system auditing and change history monitoring
Score integrity on assessments	CHECK constraints on Type and Assessment	Maintains logical grading standards and validation for pass requirements

3 ER-to-logical schema mapping

3.1 Mapping of Regular Entity Types

- ***Account*** → *Account(Account_ID, First_Name, Last_Name, Date_of_Birth, Email, Password, Sub_Status_ID)*
- ***Subscription_Status*** → *Subscription_Status(Status_ID, Status_Name)*
- ***Address*** → *Address(Account_ID, City, Street, Building)*
- ***University*** → *University(Univ_ID, Univ_Email, Univ_Name)*
- ***Teacher*** → *Teacher(Teacher_ID, Pay_Rate, Start_Date, End_Date, Univ_ID, Acc_ID)*
- ***Earnings*** → *Earnings(Teacher_ID, Earned_Money)*
- ***Certificate*** → *Certificate(Certif_ID, Cert_Description, Signature)*
- ***Course_Levels*** → *Course_Levels(Level_ID, Level_Name)*
- ***Course*** → *Course(Course_ID, Course_Code, Title, Description, Course_Level_ID, Learning_Outcomes, Period, Price, Count_USR, Coord_ID, Uni_ID, CRTF_ID)*
- ***Payment_Statuses*** → *Payment_Statuses(Status_ID, Status_Name)*
- ***Assessment*** → *Assessment(Assessment_Number, Course_Code, Course_NO, Points)*
- ***Type*** → *Type(Assessment_Number, Type, Pass_Score)*
- ***Progress*** → *Progress(Account_ID, Course_code, Course_ID, Progress_percentage)*

3.2 Mapping of Weak Entity Types

- ***Subscription*** → *Subscription(Account_ID, Price, Sub_Start, Sub_End)*
  Dependent on Account; identified by Account_ID (total participation)
- ***Plan*** → *Plan(Account_ID, Plan_Type)*
  Also dependent on Account; shares primary key with Account

3.3 Mapping of binary 1-1 relationship types

- **Account to Address** → *Enforced via Address(Account_ID) where Account_ID is both PK and FK referencing Account(Account_ID).*

3.4 Mapping of binary 1-N relationship types

- **Account to Subscription** → *Weak entity; FK Account_ID in Subscription, with total participation*
- **Account to Plan** → *Weak entity; FK Account_ID in Plan, with total participation*

- **Teacher to Earnings** → *Earnings.Teacher_ID is FK referencing Teacher*
- **Teacher to Course (as coordinator)** → *Course.Coord_ID references Teacher.Teacher_ID*
- **Teacher to University** → *Teacher.Univ_ID references University.Univ_ID*

3.5 Mapping of binary M-N relationship types

- **Enrolled_In** → *Enrolled_In(Account_ID [PK, FK], Course_ID [PK, FK], Course_Code, Enroll_Date, Hours_Watched, Payment_Status_ID [FK])*
- **Awarded** → *Awarded(Certificate_ID [PK, FK], Account_ID [PK, FK], Issue_Date)*
- **Teaches** → *Teaches(TECH_ID [PK, FK], Crs_Code, Crs_ID [PK, FK])*

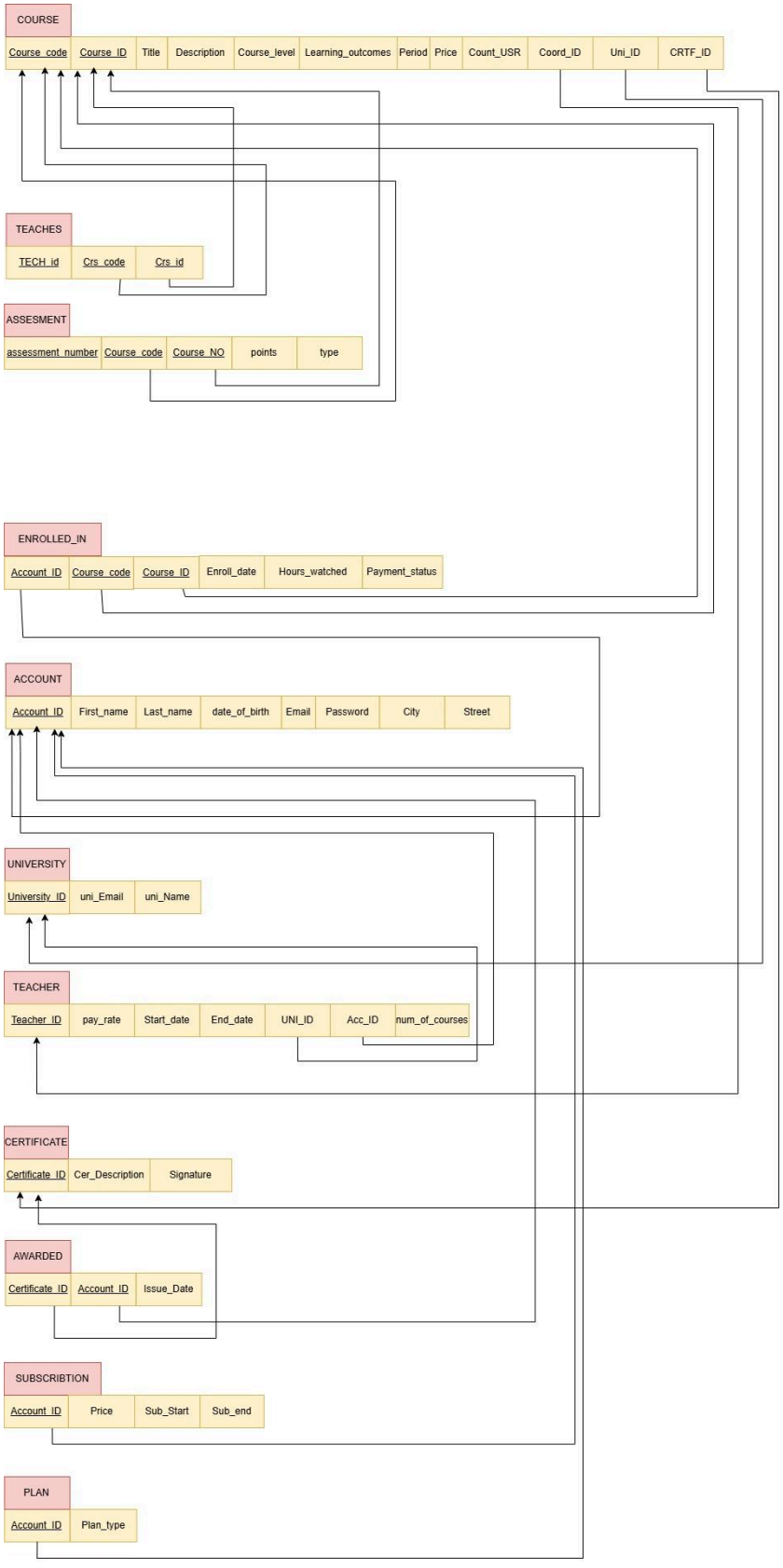
3.6 Mapping of multivalued attributes

Multivalued attributes were avoided in favor of normalized related entities where needed.

3.7 Mapping of n-ary relationship types

- **N/A —** The model contains no ternary or higher-order relationships.

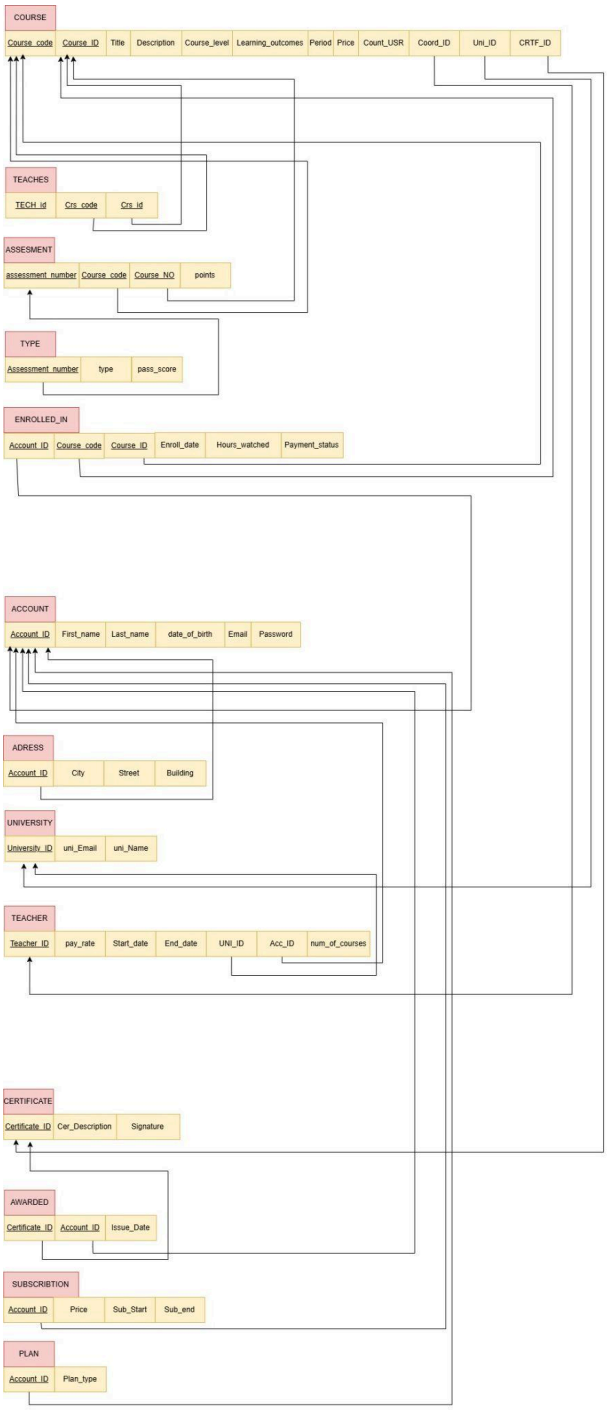
3.8 Schema Diagram



4 Normalization

4.1 First Normal Form

To achieve First Normal Form (1NF), the repeated "type" attribute in the original table was identified as a violation of the atomicity rule, which requires each column to contain indivisible values without repeating groups. This issue was resolved by moving the "type" attribute into a separate table, where each assessment (e.g., quiz, homework, or project) is distinctly categorized.



4.2 Second Normal Form

No modifications were necessary to satisfy Second Normal Form (2NF) because all non-key attributes in the existing tables already demonstrated full functional dependence on their composite primary keys. The absence of partial dependencies confirmed that every non-key attribute relied entirely on the entire key, thereby meeting 2NF requirements.

4.3 Third Normal Form

To achieve Third Normal Form, an analysis of transitive dependencies was performed across the schema. Several refinements were made:

- **Address Normalization**: The interdependent attributes "city" and "street" were moved into a new table named "Address" to eliminate transitive dependency.

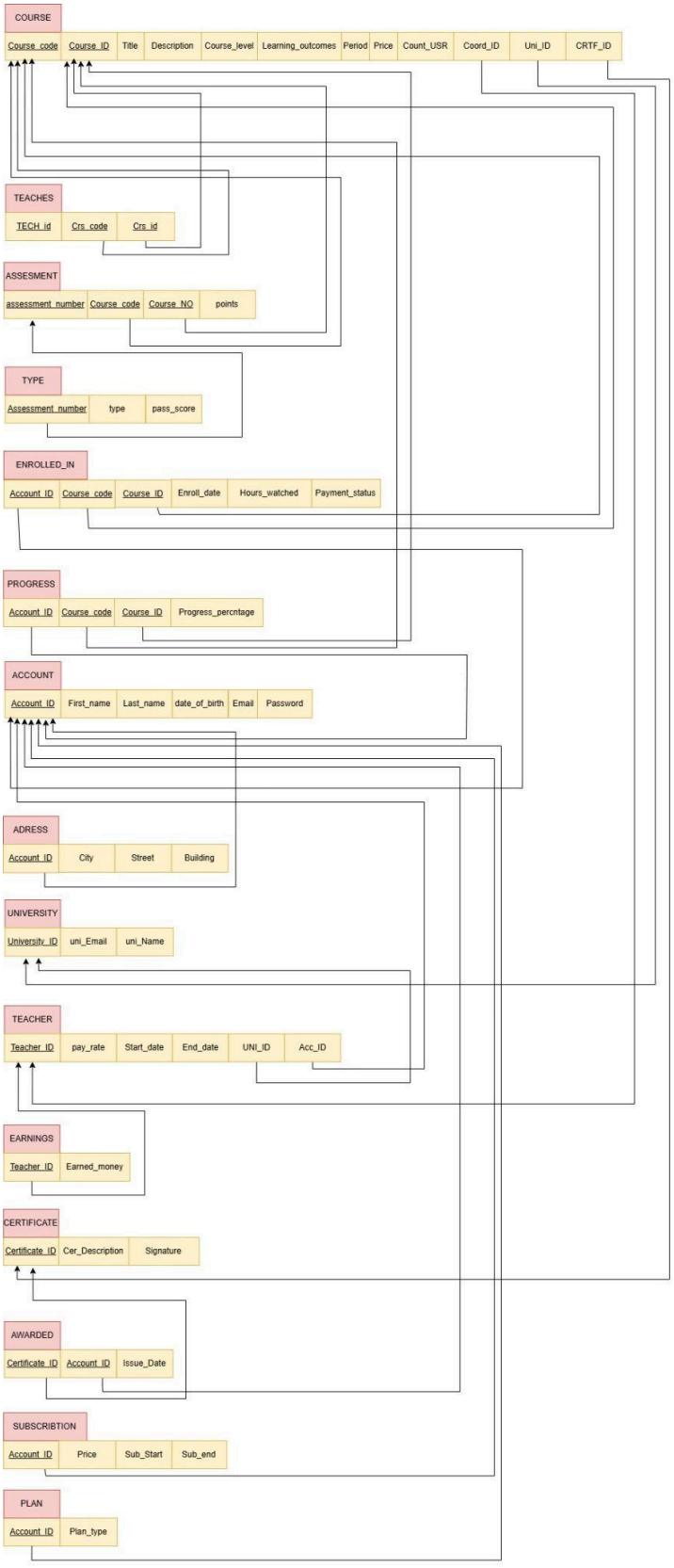
- **Earnings Calculation**: The derived attribute "earnings," which depended on "pay rate," was separated into its own "Earnings" table, since it was not fully dependent on the primary key alone.

- **Progress Tracking**: To represent "progress percent" correctly — which is dependent on "hours watched" — a dedicated table was created for progress tracking.

- **Removal of Redundant Data**: The attribute "num_of_courses," having a transitive dependency with "pay rate," was removed to maintain normalization. Additionally, important derived attributes such as "Earnings" and "Progress_Percentage" were separated into their own tables to allow more efficient and frequent querying by the system, supporting better performance from an application layer perspective.

These changes ensured full compliance with 3NF by eliminating all transitive dependencies and keeping only direct dependencies between non-key attributes and primary keys.

5 Final DB Schema Diagram



PART III: IMPLEMENTATION

6 Table Creation Script Examples

Account table :

```
CREATE TABLE Account (
    Account_ID INT PRIMARY KEY,
    First_Name VARCHAR(50) NOT NULL,
    Last_Name VARCHAR(50) NOT NULL,
    Date_of_Birth DATE NOT NULL,
    Email VARCHAR(100) NOT NULL UNIQUE,
    Password VARCHAR(64) NOT NULL,
    Sub_Status_ID INT,
    FOREIGN KEY (Sub_Status_ID) REFERENCES
Subscription_Status(Status_ID)
);
```

Address table:

```
 CREATE TABLE Address (
    Account_ID INT PRIMARY KEY,
    City VARCHAR(50) NOT NULL,
    Street VARCHAR(50) NOT NULL,
```

```
    Building VARCHAR(10) NOT NULL,

    FOREIGN KEY (Account_ID) REFERENCES
Account(Account_ID)

);
```

University table :

```
CREATE TABLE University (

    Univ_ID INT PRIMARY KEY,

    Univ_Email VARCHAR(100) NOT NULL UNIQUE,

    Univ_Name VARCHAR(100) NOT NULL

);
```

Earnings table :

```
 CREATE TABLE Earnings (

    Teacher_ID INT PRIMARY KEY,

    Earned_Money DECIMAL(10, 2),

    FOREIGN KEY (Teacher_ID) REFERENCES
Teacher(Teacher_ID)

);
```

Teacher table :

```
 CREATE TABLE Teacher (

    Teacher_ID INT PRIMARY KEY,

   Pay_Rate DECIMAL(3, 2) CHECK(Pay_Rate BETWEEN 0 AND
1),

    Start_Date DATE NOT NULL,
```

```
    End_Date DATE,
    Univ_ID INT NOT NULL,
    Acc_ID INT,
    FOREIGN KEY (Univ_ID) REFERENCES University(Univ_ID),
    FOREIGN KEY (Acc_ID) REFERENCES Account(Account_ID)
);
```

Certificate table :

```
CREATE TABLE Certificate (
    Certif_ID INT PRIMARY KEY,
    Cert_Description CLOB,
    Signature CLOB NOT NULL);
```

Awarded table :

```
CREATE TABLE Awarded (
    Certificate_ID INT,
    Account_ID INT,
    Issue_Date DATE NOT NULL,
    PRIMARY KEY (Certificate_ID, Account_ID),
    FOREIGN KEY (Certificate_ID) REFERENCES
Certificate(Certif_ID),
```

```
    FOREIGN KEY (Account_ID) REFERENCES
Account(Account_ID)
);
```

Subscription table :

```
CREATE TABLE Subscription (
    Account_ID INT PRIMARY KEY,
    Price DECIMAL(10,2) CHECK (Price >= 0),
    Sub_Start DATE NOT NULL,
    Sub_End DATE CHECK (Sub_End > Sub_Start),
    FOREIGN KEY (Account_ID) REFERENCES
Account(Account_ID)
);
```

Plan table :

```
CREATE TABLE Plan (
    Account_ID INT PRIMARY KEY,
    Plan_Type VARCHAR(20),
    FOREIGN KEY (Account_ID) REFERENCES
Account(Account_ID)
);
```

Course table:

```
CREATE TABLE Course (
```

```
    Course_ID INT,

    Course_Code VARCHAR(20),

    Title VARCHAR(25) NOT NULL,

    Description CLOB,

    Course_Level_ID INT,

    Learning_Outcomes CLOB,

    Period INT NOT NULL,

    Price DECIMAL(5, 2) NOT NULL,

    Count_USR INT,

    Coord_ID INT,

    Uni_ID INT,

    CRTF_ID INT,

    PRIMARY KEY (Course_ID, Course_Code),

    FOREIGN KEY (Course_Level_ID) REFERENCES
Course_Levels(Level_ID),

    FOREIGN KEY (Coord_ID) REFERENCES
Teacher(Teacher_ID),

    FOREIGN KEY (Uni_ID) REFERENCES University(Univ_ID),

    FOREIGN KEY (CRTF_ID) REFERENCES
Certificate(Certif_ID)

);
```

domain table for course levels:

```
CREATE TABLE Course_Levels (

    Level_ID INT PRIMARY KEY,
```

```
    Level_Name VARCHAR(20) UNIQUE
);
```

Enrolled_in table:

```
 CREATE TABLE Enrolled_In (
    Account_ID INT NOT NULL,
    Course_Code VARCHAR(20) NOT NULL,
    Course_ID INT NOT NULL,
    Enroll_Date DATE NOT NULL,
    Hours_Watched DECIMAL(5,2) NOT NULL CHECK
(Hours_Watched >= 0),
    Payment_Status_ID INT NOT NULL,
    PRIMARY KEY (Account_ID, Course_ID, Course_Code),
    FOREIGN KEY (Account_ID) REFERENCES
Account(Account_ID),
    FOREIGN KEY (Course_ID, Course_Code) REFERENCES
Course(Course_ID, Course_Code),
    FOREIGN KEY (Payment_Status_ID) REFERENCES
Payment_Statuses(Status_ID)
);
```

Domain table for payment status :

```
CREATE TABLE Payment_Statuses (
    Status_ID INT PRIMARY KEY,
    Status_Name VARCHAR(20) UNIQUE
);
```

Type table:

```
 CREATE TABLE Type (
    Assessment_Number INT PRIMARY KEY,
    Type VARCHAR(20),
    Pass_Score DECIMAL(5,2) CHECK (Pass_Score BETWEEN 0
AND 100),
    FOREIGN KEY (Assessment_Number) REFERENCES
Assessment(Assessment_Number)
);
```

Assessment table:

```
CREATE TABLE Assessment (
    Assessment_Number INT,
    Course_Code VARCHAR(20),
    Course_NO INT,
    Points INT CHECK (Points >= 0),
```

```
    PRIMARY KEY (Assessment_Number, Course_Code,
Course_NO),

    FOREIGN KEY (Course_NO, Course_Code) REFERENCES
Course(Course_ID, Course_Code)

);
```

Teaches table:

```
CREATE TABLE Teaches (

    TECH_id INT,

    Crs_code VARCHAR(20),

    Crs_id INT,

    PRIMARY KEY (TECH_id, Crs_code, Crs_id),

    FOREIGN KEY (TECH_id) REFERENCES Teacher(Teacher_ID),

    FOREIGN KEY (Crs_id, Crs_code) REFERENCES
Course(Course_ID, Course_Code) );
```

Progress table:

```
CREATE TABLE Progress (

    Account_ID INT,

    Course_code VARCHAR(20),

    Course_ID INT,

    Progress_percentage DECIMAL(5,2) CHECK
(Progress_percentage BETWEEN 0 AND 100),

    PRIMARY KEY (Account_ID, Course_code, Course_ID),
```

```
    FOREIGN KEY (Account_ID) REFERENCES
Account(Account_ID),

    FOREIGN KEY (Course_ID, Course_code) REFERENCES
Course(Course_ID, Course_Code)

);
```

Logs table:

```
CREATE TABLE Change_Log (

    Log_ID INT GENERATED ALWAYS AS IDENTITY PRIMARY KEY,

    Table_Name VARCHAR2(50),

    Action_Type VARCHAR2(10),  -- 'INSERT', 'UPDATE',
'DELETE'

    Record_ID VARCHAR2(100),   -- ID of the record
(flexible)

    Changed_By VARCHAR2(100),  -- Email or 'system'

    Change_Date TIMESTAMP DEFAULT CURRENT_TIMESTAMP,

    Change_Details CLOB    -- Summary of change

);
```

Triggers :

1-This trigger is for whenever we insert (add) a new student to enrolled table the count of student enrolled in the course increases automatically with no need to do it manually

```
CREATE OR REPLACE TRIGGER trg_increment_course_enroll

AFTER INSERT ON Enrolled_In
```

```
FOR EACH ROW

BEGIN

  UPDATE Course

  SET Count_USR = Count_USR + 1

  WHERE Course_ID = :NEW.Course_ID AND Course_Code =
:NEW.Course_Code;

END;

/
```

2- this trigger is doing the opposite of the first one which is going to decrease the count by 1 whenever a student got removed from the course

```
CREATE OR REPLACE TRIGGER trg_dec_course_enroll

AFTER DELETE ON Enrolled_In

FOR EACH ROW

BEGIN

  UPDATE Course

  SET Count_USR = Count_USR - 1

  WHERE Course_ID = :OLD.Course_ID AND Course_Code =
:OLD.Course_Code;

END;

/
```

3-this trigger is for Auto-Award Certificate on Completion.Automatically award a certificate when Progress_percentage = 100

```
CREATE OR REPLACE TRIGGER trg_auto_award_certificate
```

```
AFTER UPDATE ON Progress

FOR EACH ROW

WHEN (NEW.Progress_percentage = 100)

BEGIN

  INSERT INTO Awarded (Certificate_ID, Account_ID,
Issue_Date)

  VALUES (

     (SELECT CRTF_ID FROM Course WHERE Course_ID =
:NEW.Course_ID AND Course_Code = :NEW.Course_Code),

     :NEW.Account_ID,

     SYSDATE

  );

EXCEPTION

  WHEN DUP_VAL_ON_INDEX THEN NULL; -- prevents duplicate
awards

END;
```

4- these triggers is for the log table so we can chase any change by any user with all of there data

Account trigger:

```
CREATE OR REPLACE TRIGGER trg_log_account_changes

AFTER INSERT OR UPDATE OR DELETE ON Account

FOR EACH ROW BEGIN

  IF INSERTING THEN

     INSERT INTO Change_Log VALUES (

     NULL, 'Account', 'INSERT',

     TO_CHAR(:NEW.Account_ID),

     :NEW.Email,
```

```
        DEFAULT,
        'Inserted account'
        );
    ELSIF UPDATING THEN
        INSERT INTO Change_Log VALUES (
        NULL, 'Account', 'UPDATE',
        TO_CHAR(:OLD.Account_ID),
        :NEW.Email,
        DEFAULT,
        'Updated account'
        );
    ELSIF DELETING THEN
        INSERT INTO Change_Log VALUES (
        NULL, 'Account', 'DELETE',
        TO_CHAR(:OLD.Account_ID),
        :OLD.Email,
        DEFAULT,
        'Deleted account'
        );
    END IF;
END;
```

Subscription trigger:

```
CREATE OR REPLACE TRIGGER trg_log_subscription_changes
AFTER INSERT OR UPDATE OR DELETE ON Subscription
FOR EACH ROW
```

```
BEGIN

  IF INSERTING THEN

     INSERT INTO Change_Log VALUES (

     NULL, 'Subscription', 'INSERT',

     TO_CHAR(:NEW.Account_ID), 'system', DEFAULT,
'Inserted subscription'

     );

  ELSIF UPDATING THEN

     INSERT INTO Change_Log VALUES (

     NULL, 'Subscription', 'UPDATE',

     TO_CHAR(:OLD.Account_ID), 'system', DEFAULT, 'Updated
subscription'

     );

  ELSIF DELETING THEN

    INSERT INTO Change_Log VALUES (

     NULL, 'Subscription', 'DELETE',

     TO_CHAR(:OLD.Account_ID), 'system', DEFAULT, 'Deleted
subscription'

     );

  END IF;

END;

/
```

Plan trigger:

```
CREATE OR REPLACE TRIGGER trg_log_plan_changes
```

```
AFTER INSERT OR UPDATE OR DELETE ON Plan

FOR EACH ROW

BEGIN

  IF INSERTING THEN

     INSERT INTO Change_Log VALUES (

     NULL, 'Plan', 'INSERT',

     TO_CHAR(:NEW.Account_ID), 'system', DEFAULT,
'Inserted plan'

     );

  ELSIF UPDATING THEN

     INSERT INTO Change_Log VALUES (

      NULL, 'Plan', 'UPDATE',

     TO_CHAR(:OLD.Account_ID), 'system', DEFAULT, 'Updated
plan'

     );

  ELSIF DELETING THEN

     INSERT INTO Change_Log VALUES (

     NULL, 'Plan', 'DELETE',

     TO_CHAR(:OLD.Account_ID), 'system', DEFAULT, 'Deleted
plan'

     );

  END IF;

END;

/
```

Course trigger:

```
CREATE OR REPLACE TRIGGER trg_log_course_changes

AFTER INSERT OR UPDATE OR DELETE ON Course

FOR EACH ROW

BEGIN

  IF INSERTING THEN

     INSERT INTO Change_Log VALUES (

     NULL, 'Course', 'INSERT',

     TO_CHAR(:NEW.Course_ID), 'system', DEFAULT, 'Inserted
course'

     );

  ELSIF UPDATING THEN

     INSERT INTO Change_Log VALUES (

     NULL, 'Course', 'UPDATE',

     TO_CHAR(:OLD.Course_ID), 'system', DEFAULT, 'Updated
course'

     );

  ELSIF DELETING THEN

     INSERT INTO Change_Log VALUES (

     NULL, 'Course', 'DELETE',

     TO_CHAR(:OLD.Course_ID), 'system', DEFAULT, 'Deleted
course'

     );

  END IF;

END;
```

```
/
```

Enrolled in trigger:

```
CREATE OR REPLACE TRIGGER trg_log_enrolled_in_changes
AFTER INSERT OR UPDATE OR DELETE ON Enrolled_In
FOR EACH ROW
BEGIN
  IF INSERTING THEN
     INSERT INTO Change_Log VALUES (
     NULL, 'Enrolled_In', 'INSERT',
     TO_CHAR(:NEW.Account_ID), 'system', DEFAULT, 'Student
enrolled'
     );
  ELSIF UPDATING THEN
     INSERT INTO Change_Log VALUES (
     NULL, 'Enrolled_In', 'UPDATE',
     TO_CHAR(:OLD.Account_ID), 'system', DEFAULT,
'Enrollment updated'
     );
  ELSIF DELETING THEN
     INSERT INTO Change_Log VALUES (
     NULL, 'Enrolled_In', 'DELETE',
     TO_CHAR(:OLD.Account_ID), 'system', DEFAULT, 'Student
unenrolled'
     );
```

```
  END IF;
END;
/
```

Progress trigger:

```
CREATE OR REPLACE TRIGGER trg_log_progress_changes
AFTER INSERT OR UPDATE OR DELETE ON Progress
FOR EACH ROW
BEGIN
  IF INSERTING THEN
     INSERT INTO Change_Log VALUES (
      NULL, 'Progress', 'INSERT',
     TO_CHAR(:NEW.Account_ID), 'system', DEFAULT,
'Inserted into Progress'
     );
  ELSIF UPDATING THEN
     INSERT INTO Change_Log VALUES (
     NULL, 'Progress', 'UPDATE',
     TO_CHAR(:OLD.Account_ID), 'system', DEFAULT, 'Updated
Progress'
     );
  ELSIF DELETING THEN
     INSERT INTO Change_Log VALUES (
     NULL, 'Progress', 'DELETE',
```

```
    TO_CHAR(:OLD.Account_ID), 'system', DEFAULT, 'Deleted
from Progress'

    );

  END IF;

END;

/
```

Awarded trigger:

```
CREATE OR REPLACE TRIGGER trg_log_awarded_changes

AFTER INSERT OR UPDATE OR DELETE ON Awarded

FOR EACH ROW

BEGIN

  IF INSERTING THEN

    INSERT INTO Change_Log VALUES (

    NULL, 'Awarded', 'INSERT',

    TO_CHAR(:NEW.Account_ID), 'system', DEFAULT,
'Inserted into Awarded'

    );

  ELSIF UPDATING THEN

    INSERT INTO Change_Log VALUES (

    NULL, 'Awarded', 'UPDATE',

    TO_CHAR(:OLD.Account_ID), 'system', DEFAULT, 'Updated
Awarded'

    );

  ELSIF DELETING THEN
```

```
    INSERT INTO Change_Log VALUES (
    NULL, 'Awarded', 'DELETE',
    TO_CHAR(:OLD.Account_ID), 'system', DEFAULT, 'Deleted
from Awarded'
    );
  END IF;
END;
/
```

Assessment trigger:

```
CREATE OR REPLACE TRIGGER trg_log_assessment_changes
AFTER INSERT OR UPDATE OR DELETE ON Assessment
FOR EACH ROW
BEGIN
  IF INSERTING THEN
    INSERT INTO Change_Log VALUES (
    NULL, 'Assessment', 'INSERT',
    TO_CHAR(:NEW.Assessment_Number), 'system', DEFAULT,
'Inserted into Assessment'
    );
  ELSIF UPDATING THEN
    INSERT INTO Change_Log VALUES (
    NULL, 'Assessment', 'UPDATE',
    TO_CHAR(:OLD.Assessment_Number), 'system', DEFAULT,
'Updated Assessment'
```

```
    );
  ELSIF DELETING THEN
    INSERT INTO Change_Log VALUES (
    NULL, 'Assessment', 'DELETE',
    TO_CHAR(:OLD.Assessment_Number), 'system', DEFAULT,
'Deleted from Assessment'
    );
  END IF;
END;
/
```

Type trigger:

```
CREATE OR REPLACE TRIGGER trg_log_type_changes
AFTER INSERT OR UPDATE OR DELETE ON Type
FOR EACH ROW
BEGIN
  IF INSERTING THEN
    INSERT INTO Change_Log VALUES (
    NULL, 'Type', 'INSERT',
    TO_CHAR(:NEW.Assessment_Number), 'system', DEFAULT,
'Inserted into Type'
    );
  ELSIF UPDATING THEN
    INSERT INTO Change_Log VALUES (
    NULL, 'Type', 'UPDATE',
```

```
    TO_CHAR(:OLD.Assessment_Number), 'system', DEFAULT,
'Updated Type'

    );

  ELSIF DELETING THEN

    INSERT INTO Change_Log VALUES (

    NULL, 'Type', 'DELETE',

    TO_CHAR(:OLD.Assessment_Number), 'system', DEFAULT,
'Deleted from Type'

    );

  END IF;

END;

/
```

Certificate trigger:

```
CREATE OR REPLACE TRIGGER trg_log_certificate_changes

AFTER INSERT OR UPDATE OR DELETE ON Certificate

FOR EACH ROW

BEGIN

  IF INSERTING THEN

    INSERT INTO Change_Log VALUES (

    NULL, 'Certificate', 'INSERT',

    TO_CHAR(:NEW.Certif_ID), 'system', DEFAULT, 'Inserted
into Certificate'

    );

  ELSIF UPDATING THEN
```

```
    INSERT INTO Change_Log VALUES (
    NULL, 'Certificate', 'UPDATE',
    TO_CHAR(:OLD.Certif_ID), 'system', DEFAULT, 'Updated
Certificate'
    );
  ELSIF DELETING THEN
    INSERT INTO Change_Log VALUES (
    NULL, 'Certificate', 'DELETE',
    TO_CHAR(:OLD.Certif_ID), 'system', DEFAULT, 'Deleted
from Certificate'
    );
  END IF;
END;
/
```

Teacher trigger:

```
CREATE OR REPLACE TRIGGER trg_log_teacher_changes
AFTER INSERT OR UPDATE OR DELETE ON Teacher
FOR EACH ROW
BEGIN
  IF INSERTING THEN
    INSERT INTO Change_Log VALUES (
    NULL, 'Teacher', 'INSERT',
    TO_CHAR(:NEW.Teacher_ID), 'system', DEFAULT,
'Inserted into Teacher'
```

```
    );
  ELSIF UPDATING THEN
    INSERT INTO Change_Log VALUES (
    NULL, 'Teacher', 'UPDATE',
    TO_CHAR(:OLD.Teacher_ID), 'system', DEFAULT, 'Updated
Teacher'
    );
  ELSIF DELETING THEN
    INSERT INTO Change_Log VALUES (
    NULL, 'Teacher', 'DELETE',
    TO_CHAR(:OLD.Teacher_ID), 'system', DEFAULT, 'Deleted
from Teacher'
    );
  END IF;
END;
/
```

Earnings trigger:

```
CREATE OR REPLACE TRIGGER trg_log_earnings_changes
AFTER INSERT OR UPDATE OR DELETE ON Earnings
FOR EACH ROW
BEGIN
  IF INSERTING THEN
    INSERT INTO Change_Log VALUES (
    NULL, 'Earnings', 'INSERT',
```

```
    TO_CHAR(:NEW.Teacher_ID), 'system', DEFAULT,
'Inserted into Earnings'

    );

  ELSIF UPDATING THEN

    INSERT INTO Change_Log VALUES (

    NULL, 'Earnings', 'UPDATE',

    TO_CHAR(:OLD.Teacher_ID), 'system', DEFAULT, 'Updated
Earnings'

    );

  ELSIF DELETING THEN

    INSERT INTO Change_Log VALUES (

    NULL, 'Earnings', 'DELETE',

    TO_CHAR(:OLD.Teacher_ID), 'system', DEFAULT, 'Deleted
from Earnings'

    );

  END IF;

END;

/
```

Address trigger:

```
CREATE OR REPLACE TRIGGER trg_log_address_changes

AFTER INSERT OR UPDATE OR DELETE ON Address

FOR EACH ROW

BEGIN

  IF INSERTING THEN
```

```
    INSERT INTO Change_Log VALUES (

    NULL, 'Address', 'INSERT',

    TO_CHAR(:NEW.Account_ID), 'system', DEFAULT,
'Inserted into Address'

    );

  ELSIF UPDATING THEN

    INSERT INTO Change_Log VALUES (

    NULL, 'Address', 'UPDATE',

    TO_CHAR(:OLD.Account_ID), 'system', DEFAULT, 'Updated
Address'

    );

  ELSIF DELETING THEN

    INSERT INTO Change_Log VALUES (

    NULL, 'Address', 'DELETE',

    TO_CHAR(:OLD.Account_ID), 'system', DEFAULT, 'Deleted
from Address'

    );

  END IF;

END;

/
```

Teaches trigger:

```
CREATE OR REPLACE TRIGGER trg_log_teaches_changes

AFTER INSERT OR UPDATE OR DELETE ON Teaches

FOR EACH ROW
```

```
BEGIN

  IF INSERTING THEN

     INSERT INTO Change_Log VALUES (

     NULL, 'Teaches', 'INSERT',

     TO_CHAR(:NEW.TECH_id), 'system', DEFAULT, 'Inserted
into Teaches'

     );

  ELSIF UPDATING THEN

     INSERT INTO Change_Log VALUES (

     NULL, 'Teaches', 'UPDATE',

     TO_CHAR(:OLD.TECH_id), 'system', DEFAULT, 'Updated
Teaches'

     );

  ELSIF DELETING THEN

     INSERT INTO Change_Log VALUES (

     NULL, 'Teaches', 'DELETE',

     TO_CHAR(:OLD.TECH_id), 'system', DEFAULT, 'Deleted
from Teaches'

     );

  END IF;

END;

/
```

view tables :

Student Progress table :

```
CREATE VIEW Student_Progress_View AS

SELECT

     A.Account_ID,

     A.First_Name || ' ' || A.Last_Name AS Student_Name,

     C.Title AS Course_Title,

     P.Progress_percentage

FROM

     Progress P

JOIN Account A ON A.Account_ID = P.Account_ID

JOIN Course C ON C.Course_ID = P.Course_ID AND
C.Course_Code = P.Course_Code;
```

This table will view the student progress

Teacher earnings table :

```
CREATE VIEW Teacher_Earnings_View AS

SELECT

     T.Teacher_ID,

     TA.First_Name || ' ' || TA.Last_Name AS Teacher_Name,

     SUM(C.Price * C.Count_USR * T.Pay_Rate) AS
Calculated_Earnings

FROM

     Teacher T

JOIN Course C ON C.Coord_ID = T.Teacher_ID

JOIN Account TA ON T.Acc_ID = TA.Account_ID

GROUP BY T.Teacher_ID, TA.First_Name, TA.Last_Name;
```

Enrolled students table :

```
CREATE VIEW Enrolled_Students_View AS

SELECT

     ST.First_Name || ' ' || ST.Last_Name AS Student_Name,

     C.Title AS Course_Title,

     E.Enroll_Date,

     P.Status_Name AS Payment_Status

FROM

     Enrolled_In E

JOIN Account ST ON ST.Account_ID = E.Account_ID

JOIN Course C ON C.Course_ID = E.Course_ID AND
C.Course_Code = E.Course_Code

JOIN Payment_Statuses P ON E.Payment_Status_ID =
P.Status_ID;
```

Active subscriptable table :

```
CREATE VIEW Active_Subscriptions_View AS

SELECT

     A.Account_ID,

     A.First_Name || ' ' || A.Last_Name AS Full_Name,

     S.Sub_Start,

     S.Sub_End

FROM

     Subscription S

JOIN Account A ON S.Account_ID = A.Account_ID

WHERE CURRENT_DATE BETWEEN S.Sub_Start AND S.Sub_End;
```

7 Constraints Script

Business Rule	SQL Script	Table
Passwords are stored securely	`Password VARCHAR(64) NOT NULL` (stored hashed with SHA-256)	`Account`
Subscription status uses domain table	`FOREIGN KEY (Sub_Status_ID) REFERENCES Subscription_Status(Status_ID)`	`Account`
Students can earn certificates	`Awarded (Certificate_ID, Account_ID)` with `FOREIGN KEY (Certificate_ID) REFERENCES Certificate(Certif_ID)`, `FOREIGN KEY (Account_ID) REFERENCES Account(Account_ID)`	`Awarded`
Teachers have earnings	`FOREIGN KEY (Teacher_ID) REFERENCES Teacher(Teacher_ID)`	`Earnings`
Enrollment affects user count	Trigger `trg_increment_course_enroll` (after insert on `Enrolled_In`) and Trigger `trg_dec_course_enroll` (after delete on `Enrolled_In`) — both updating `Course.Count_USR`	`Enrolled_In` and `Course`
table changes are tracked	CREATE OR REPLACE TRIGGER trg_log_<table_name>_changes AFTER INSERT OR UPDATE OR DELETE ON <table_name> FOR EACH ROW BEGIN IF INSERTING THEN INSERT INTO Change_Log (...) ELSIF UPDATING THEN INSERT INTO Change_Log (...) ELSIF DELETING THEN INSERT INTO Change_Log (...) END IF; END; / (one trigger for each table: Account, Subscription, Plan, Course, Enrolled_In, Progress, Awarded, Assessment, Type, Certificate, Teacher, Earnings, Address, Teaches)	`log table`
Subscription periods must be valid	`CHECK (Sub_End > Sub_Start)` constraint on `Sub_End` and `Sub_Start`	`Subscription`

Certificate must include a signature	`Signature CLOB NOT NULL` — with note that it stores base64 or text hash	`Certificate`
Teachers must belong to a university	`FOREIGN KEY (Univ_ID) REFERENCES University(Univ_ID)`	`Teacher`
Valid payment status required	`FOREIGN KEY (Payment_Status_ID) REFERENCES Payment_Statuses(Status_ID)`	`Enrolled_In`
Score integrity on assessments	`CHECK (Pass_Score BETWEEN 0 AND 100)` on `Type`, and `CHECK (Points >= 0)` on `Assessment`	`Type`, `Assessment`

8 Queries

8.1 Student Progress with Coordinator

Description:
Retrieve each student's first and last name, the course title they are enrolled in, their course coordinator's name, and their progress percentage.

SQL Script:

```
SELECT
        A.First_Name AS Student_First_Name,
        A.Last_Name AS Student_Last_Name,
        C.Title AS Course_Title,
        (
        SELECT TA.First_Name || ' ' || TA.Last_Name
        FROM Teacher T
        JOIN Account TA ON T.Acc_ID = TA.Account_ID
        WHERE T.Teacher_ID = C.Coord_ID
        ) AS Coordinator_Name,
        P.Progress_percentage
FROM
        Progress P
JOIN Account A ON P.Account_ID = A.Account_ID
JOIN Course C ON P.Course_ID = C.Course_ID AND P.Course_Code =
C.Course_Code;
```

Sample Output (first five rows):

Student_First_Name	Student_Last_Name	Course_Title	Coordinator_Name	Progress_Percentage
Sara	Brown	Intro to CS	Alice Smith	60.00
Ali	Ahmed	Data Structures	Bob Jones	85.00
Lina	Omar	Algorithms	Alice Smith	45.00
Faisal	Zahrani	Databases	Bob Jones	75.00
Huda	Saleh	Operating Systems	Alice Smith	90.00

8.2 Teacher Earnings Calculation

Description:
Calculate the earnings of each teacher based on the price of their courses, the number of enrolled students, and their pay rate.

SQL Script:

```
SELECT
        T.Teacher_ID,
        A.First_Name || ' ' || A.Last_Name AS Teacher_Name,
        C.Title AS Course_Title,
        C.Price,
        C.Count_USR,
        T.Pay_Rate,
        (C.Price * C.Count_USR * T.Pay_Rate) AS Calculated_Earnings
FROM
        Teacher T
JOIN Account A ON T.Acc_ID = A.Account_ID
JOIN Course C ON C.Coord_ID = T.Teacher_ID;
```

Sample Output (first five rows):

Teacher_ID	Teacher_Name	Course_Title	Price	Count_USR	Pay_Rate	Calculated_Earnings
1	Alice Smith	Intro to CS	199.99	30	0.75	4499.78
2	Bob Jones	Data Structures	299.99	20	0.85	5098.30
3	Alice Smith	Algorithms	250.00	25	0.75	4687.50
4	Bob Jones	Databases	150.00	15	0.85	1912.50
5	Alice Smith	Operating Systems	180.00	28	0.75	3780.00

8.3 Student Progress Status

Description:

Display the journey of the students by showing their name, course title, teacher's name, progress percentage, and a status (Completed, In Progress, Just Started) based on their progress.

SQL Script:

```
SELECT
    S.Account_ID AS Student_ID,
    S.First_Name || ' ' || S.Last_Name AS Student_Name,
    C.Title AS Course_Title,
    TCH.First_Name || ' ' || TCH.Last_Name AS Teacher_Name,
    P.Progress_percentage,
    CASE
    WHEN P.Progress_percentage = 100 THEN 'Completed'
    WHEN P.Progress_percentage >= 50 THEN 'In Progress'
    ELSE 'Just Started'
```

```
        END AS Completion_Status,
        SS.Status_Name AS Subscription_Status
FROM
        Progress P
JOIN Account S ON S.Account_ID = P.Account_ID
JOIN Course C ON C.Course_ID = P.Course_ID AND C.Course_Code =
P.Course_Code
JOIN Teacher T ON C.Coord_ID = T.Teacher_ID
JOIN Account TCH ON T.Acc_ID = TCH.Account_ID
JOIN Subscription_Status SS ON S.Sub_Status_ID = SS.Status_ID;
```

Sample Output (first five rows):

Student_Name	Course_Title	Teacher_Name	Progress_Percentage	Completion_Status	Subscription_Status
Sara Brown	Intro to CS	Alice Smith	60.00	In Progress	active
Ali Ahmed	Data Structures	Bob Jones	100.00	Completed	active
Lina Omar	Algorithms	Alice Smith	40.00	Just Started	inactive
Faisal Zahrani	Databases	Bob Jones	70.00	In Progress	active
Huda Saleh	Operating Systems	Alice Smith	100.00	Completed	active

8.4 Certificates Awarded to Students

Description:
List each student with the certificate they earned, description of the certificate, the university it came from, the stored signature, and the issue date.

SQL Script:

```
SELECT
        ST.First_Name || ' ' || ST.Last_Name AS Student_Name,
        C.Title AS Certificate_Title,
        Ce.Cert_Description AS Certificate_Description,
        U.Univ_Name AS University_Name,
        Ce.Signature,
        A.Issue_Date
FROM
        Awarded A
JOIN Certificate Ce ON A.Certificate_ID = Ce.Certif_ID
JOIN Course C ON C.CRTF_ID = Ce.Certif_ID
JOIN University U ON C.Uni_ID = U.Univ_ID
JOIN Account ST ON A.Account_ID = ST.Account_ID;
```

Sample Output (first five rows):

Student_Name	Certificate_Title	Certificate_Description	University_Name	Signature	Issue_Date
Sara Brown	Intro to CS	Completion Certificate	NYU	sig_abc	2024-05-01
Ali Ahmed	Data Structures	Honor Certificate	UCLA	sig_xyz	2024-05-02
Lina Omar	Algorithms	Completion Certificate	NYU	sig_abc	2024-06-01
Faisal Zahrani	Databases	Honor Certificate	UCLA	sig_xyz	2024-06-15

Huda Saleh	Operating Systems	Completion Certificate	NYU	sig_abc	2024-07-01

8.5 Courses with No Student Completions

Description:
This query identifies courses that currently have students enrolled but no students have yet completed them. It helps administrators and instructors spot courses where student success is very low and take early corrective actions if needed.

SQL Script:

```
SELECT
        C.Course_ID,
        C.Course_Code,
        C.Title AS Course_Title,
        COUNT(E.Account_ID) AS Enrolled_Students,
        SUM(CASE WHEN P.Progress_percentage = 100 THEN 1 ELSE 0 END) AS
Completed_Students
FROM
        Course C
JOIN Enrolled_In E ON C.Course_ID = E.Course_ID AND C.Course_Code =
E.Course_Code
JOIN Progress P ON E.Account_ID = P.Account_ID AND E.Course_ID =
P.Course_ID AND E.Course_Code = P.Course_Code
GROUP BY
        C.Course_ID, C.Course_Code, C.Title
HAVING
        SUM(CASE WHEN P.Progress_percentage = 100 THEN 1 ELSE 0 END) = 0;
```

Sample Output (first five rows):

Course_ID	Course_Code	Course_Title	Enrolled_Students	Completed_Students
5	AI101	Introduction to AI	25	0

7	ML202	Machine Learning	15	0
8	DS303	Data Science Basics	10	0
10	SE404	Software Engineering	20	0
12	DB505	Advanced Databases	12	0

APPENDIX:

Account Table

Account_ID	First_Name	Last_Name	Date_of_Birth	Email	Password	Sub_Status_ID
1	Alice	Smith	2000-01-01	alice@example.com	hash1	1
2	Bob	Jones	1995-08-22	bob@example.com	hash2	2
3	Sara	Brown	2003-04-10	sara@example.com	hash3	1

Address Table

Account_ID	City	Street	Building
1	New York	5th Ave	101
2	Chicago	Michigan Ave	202
3	Houston	Main St	303

University Table

Univ_ID	Univ_Email	Univ_Name
1	nyu@edu.com	NYU
2	ucla@edu.com	UCLA

Subscription_Status Table

Status_ID	Status_Name
1	active
2	inactive

Course_Levels Table

Level_ID	Level_Name
1	beginner
2	intermediate
3	advanced

Payment_Statuses Table

Status_ID	Status_Name
1	paid
2	pending
3	failed

Teacher Table

Teacher_ID	Pay_Rate	Start_Date	End_Date	Univ_ID	Acc_ID
1	0.75	2023-01-01	2023-12-31	1	1
2	0.85	2023-02-01	2023-11-30	2	2

Earnings Table

Teacher_ID	Earned_Money
1	4500.00
2	5000.00

Certificate Table

Certif_ID	Cert_Description	Signature
1	Completion Certificate	sig_abc
2	Honor Certificate	sig_xyz

Awarded Table

Certificate_ID	Account_ID	Issue_Date
1	3	2024-05-01
2	3	2024-05-10

Subscription Table

Account_ID	Price	Sub_Start	Sub_End
1	49.99	2024-01-01	2024-12-31
2	59.99	2024-03-01	2024-12-01
3	39.99	2024-04-01	2024-10-01

Plan Table

Account_ID	Plan_Type
1	monthly
2	yearly
3	free

Course Table

Course_ID	Course_Code	Title	Period	Price	Count_USR	Coord_ID	Uni_ID	CRTF_ID
1	CS101	Intro to CS	8	199.99	30	1	1	1
2	CS201	Data Structures	12	299.99	20	2	2	2

Enrolled_In Table

Account_ID	Course_Code	Course_ID	Enroll_Date	Hours_Watched	Payment_Status_ID
3	CS101	1	2024-04-10	4.5	1

Progress Table

Account_ID	Course_code	Course_ID	Progress_percentage
3	CS101	1	60.00

Assessment Table

Assessment_Number	Course_Code	Course_NO	Points
1	CS101	1	100
2	CS201	2	100

Type Table

Assessment_Number	Type	Pass_Score
1	quiz	60.00
2	exam	75.00

Teaches Table

TECH_id	Crs_code	Crs_id
1	CS101	1
2	CS201	2