Hybrid Learning System Database

https://github.com/Sushma897sree/SQL_Assignment

Submitted by: Sushma Sree Laskar

Student_id: 23032632

Introduction:

The Hybrid Learning System Database has been designed to simulate the management of both online and offline educational courses. This database supports key functions managing all other tables information, ensuring an effective flow of informed decision making.

The database created comprises multiple tables such as Users(students), Instructors, courses, offline branches, enrolments, payments, reviews, each incorporated with mix of nominal, ordinal, interval and ratio data types. Randomised data has been generated to enhance realism, deliberate inclusion of missing and duplicate data to simulate real world scenarios and test data-handling capabilities.

Foreign and compound keys have been implemented across multiple tables to maintain relational integrity. This report outlines the data generation process, the database schema, and the rational for table structures and constraints. It also discusses ethical considerations and data privacy to ensure compliance with industry standards.

Database Schema:

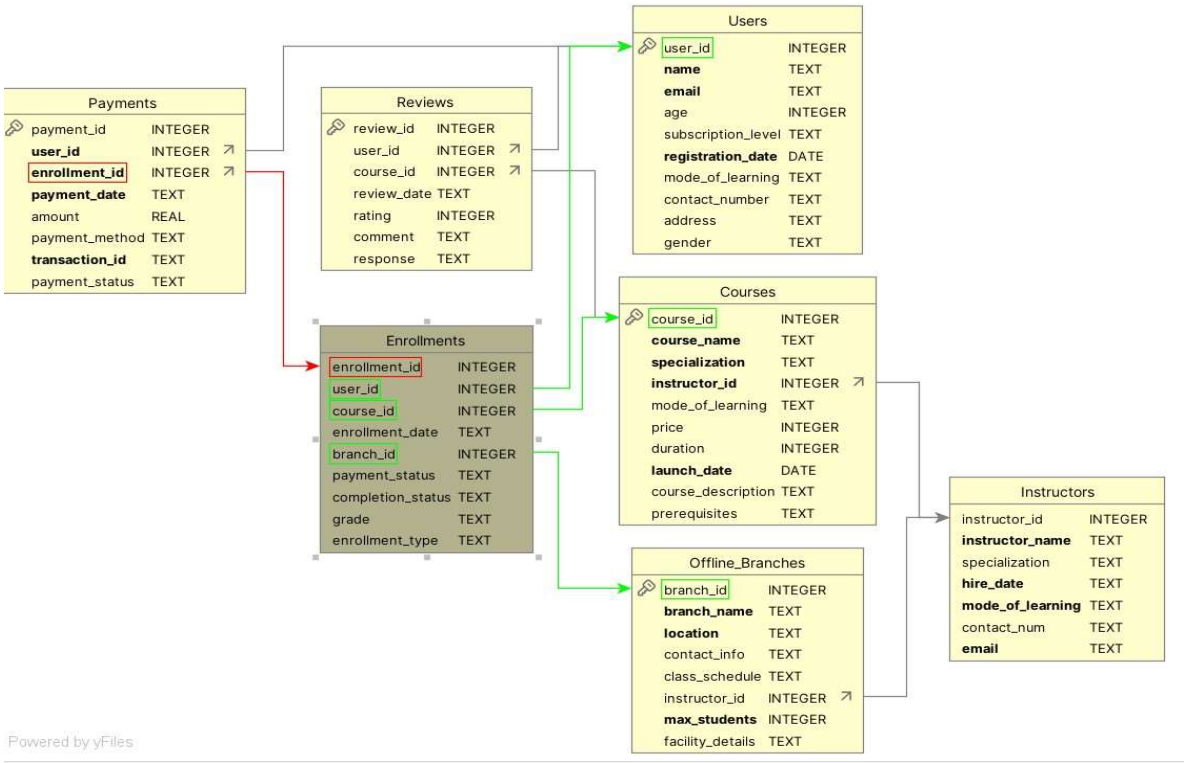


Fig 1. hybrid_learning@in.db Schema

Here green connection indicates usage of composite key and red connection and grey connection for foreign key and each table has its primary key to give unique records

Let's discuss in detail with each table.

1.Users Table Design and Data Generation:

The Users Table is a core part of the database, serving as the central repository for user data. It has columns with datatypes as follows:

Nominal: user_id, name, email, mode_of_learning, contact number, address, gender
Interval: registration_date
Ordinal: subscription_level
Ratio: age
Key Design Considerations:

1. **Primary Key:** The user_id field is the primary key for the table, ensuring each user record is unique. This allows easy referencing from other tales like Enrolments, payments, reviews through foreign key relationships.
2. **Not Null Constraints**: name, email, registration_date are marked as NOT NULL because to ensure uniquely identify users and register them.
3. **Check Constraints**:
   Subscription_level: It restricts only to Basics, Premium and VIP values ensuring only valid entries.
   mode_of_learning: Online or Offline
   Gender: Male, Female, other
4. **Unique Constraints**: The email is unique constraint ensures no same email address for each entry.

```
# Create the Users table with necessary constraints
cursor.execute('''
    CREATE TABLE IF NOT EXISTS Users (
        user_id INTEGER PRIMARY KEY,
        name TEXT NOT NULL,
        email TEXT NOT NULL,
        age INTEGER,
        subscription_level TEXT CHECK (subscription_level IN ('Basic', 'Premium', 'VIP')),
        registration_date DATE NOT NULL,
        mode_of_learning TEXT CHECK (mode_of_learning IN ('Online', 'Offline')),
        contact_number TEXT,
        address TEXT,
        gender TEXT CHECK (gender IN ('Male', 'Female', 'Other'))
    )
''')
--conn.commit()
```

Fig 1.1 Users Table Creation

Data Generation:

Data for all tables generated using Faker library, which stimulates realistic data while introducing some randomness. Intentionally missing values included in columns like contact number, address and duplicate names were allowed.

```
for _ in range(num_records):
    # Generate realistic data for critical columns
    name = fake.name()  # Allow duplicates naturally by not tracking unique nam
    email = fake.unique.email()  # Unique email to avoid duplicates
    age = random.randint(18, 70)
    subscription_level = random.choice(['Basic', 'Premium', 'VIP'])
    registration_date = fake.date_this_decade()
    mode_of_learning = random.choice(['Online', 'Offline'])
    gender = random.choice(['Male', 'Female', 'Other'])

    # Optional fields, where missing data is simulated
    contact_number = fake.phone_number() if random.random() < 0.8 else None
                                    # 80% chance of having a contact number
    address = fake.address() if random.random() < 0.8 else None
                                    # 80% chance of having an address

    # Add user data to the list
    user_data.append((name, email, age, subscription_level,registration_date,
                      mode_of_learning, contact_number, address, gender))

return user_data
```

Fig 1.2 Data Generation for Users table

Constraints and Integrity Testing

No Null values in fields having critical constraints like email, supscription_level, mode_of_learning. Also, no violation of critical constraints is observed indicating data integrity. Almost 453 missing records were observed having missing values in either of these (age, contact_num, address) of these fields.11 Duplicate names were found in the Users table.

```
# 1. Check for NULL values in critical columns
print("1. Checking for NULL values in critical columns...")
cursor.execute('''
    SELECT COUNT(*)
    FROM Users
    WHERE name IS NULL OR email IS NULL OR subscription_level IS NULL OR registration_date IS NULL OR mode_of_learning IS NULL OR gender IS NULL
''')
null_values = cursor.fetchone()[0]
print(f"Number of records with NULL values in critical columns: {null_values}")

# 2. Check for constraint violations
print("\n2. Checking for constraint violations...")
cursor.execute('''
    SELECT COUNT(*)
    FROM Users
    WHERE subscription_level NOT IN ('Basic', 'Premium', 'VIP') OR
          mode_of_learning NOT IN ('Online', 'Offline') OR
          gender NOT IN ('Male', 'Female', 'Other')
''')
constraint_violations = cursor.fetchone()[0]
print(f"Number of records violating constraints: {constraint_violations}")

# 3. Check for missing and duplicate data
print("\n3. Checking for missing and duplicate data...")

# Missing data
cursor.execute('''
    SELECT COUNT(*)
    FROM Users
    WHERE age IS NULL OR contact_number IS NULL OR address IS NULL
''')
missing_data = cursor.fetchone()[0]
print(f"Number of records with missing optional data: {missing_data}")
```

Fig 1.3 Data Integrity Testing queries

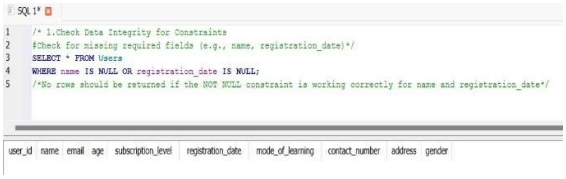
```
# Duplicate names
cursor.execute('''
    SELECT name, COUNT(*)
    FROM Users
    GROUP BY name
    HAVING COUNT(*) > 1
''')
duplicate_names = cursor.fetchall()
print(f"Number of duplicate names: {len(duplicate_names)}")
if len(duplicate_names) > 0:
    print("Examples of duplicate names:")
    print(duplicate_names[:5])  # Show a few duplicates

# 4. Check total rows
print("\n4. Checking total rows...")
cursor.execute('SELECT COUNT(*) FROM Users')
total_rows = cursor.fetchone()[0]
print(f"Total rows in the Users table: {total_rows}")

# Close the connection
conn.close()
```

```
1. Checking for NULL values in critical columns...
Number of records with NULL values in critical columns: 0

2. Checking for constraint violations...
Number of records violating constraints: 0

3. Checking for missing and duplicate data...
Number of records with missing optional data: 453
Number of duplicate names: 11
Examples of duplicate names:
[('Alyssa Johnson', 2), ('Christopher Spencer', 2), ('Daniel Young', 2), ('Larry Herrera', 2), ('Mary Johnson', 2)]

4. Checking total rows...
Total rows in the Users table: 1300
```

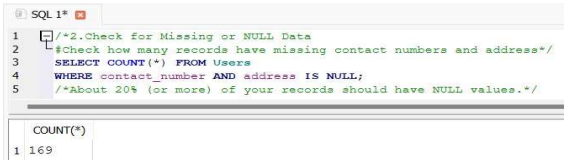
Fig 1.4 Data Integrity Testing Results

Few more queries:

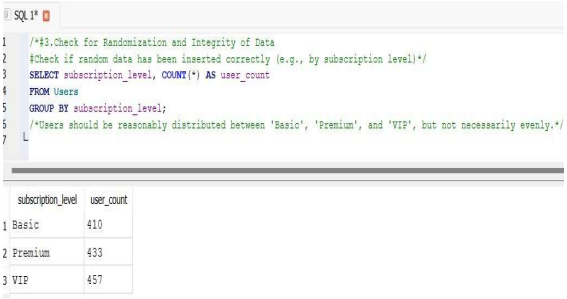
1.Check for missing required fields (e.g., name, registration_date):No rows should be returned if the NOT NULL constraint is working correctly for name and registration_date



2. Check for Missing or NULL Data:Count how many records have missing contact numbers and address:About 20% (or more) of records should have NULL values.



3. Check if random data has been inserted correctly (e.g., by subscription_level): Users should be reasonably distributed between, but not necessarily evenly.



2.Instructors Table: It comprises of 7 fields as follows: instructor_id, instructor_name, specialization, hire_data, mode of learning, contact_num, email.

```
# Creating the Instructors table with appropriate columns and constraints
cursor.execute('''
    CREATE TABLE IF NOT EXISTS Instructors (
        instructor_id INTEGER PRIMARY KEY AUTOINCREMENT,
        instructor_name TEXT NOT NULL,
        specialization TEXT,
        hire_date TEXT NOT NULL CHECK(hire_date LIKE '____-__-__'),  -- Ensure date format YYYY-MM-DD
        mode_of_learning TEXT NOT NULL CHECK(mode_of_learning IN ('Online', 'Offline')),  -- Ensure valid mode
        contact_num TEXT,
        email TEXT NOT NULL UNIQUE  -- Ensure email is unique
    )
''')
```

Fig 2.1 Instructors Table Creation

Key Design Considerations:

1.Primary Key:

The instructor_id field is an auto-incrementing primary key, ensuring unique identification for each instructor record. It allows foreign key relationships with courses and enrolments table.

2.Not Null Constraints:

Critical columns like (instructor_id, hire_date, email, mode_of_learning) are marked as NOT NULL to prevent incomplete entries.

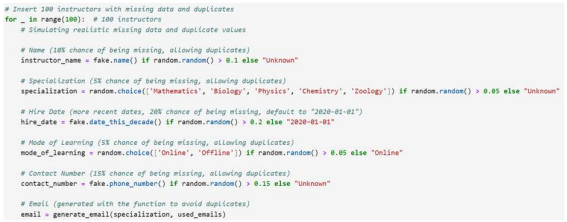
3. Check Constraints:

Mode_of_learning is restricted to valid values (online, offline) and hire_date uses valid date formatting YYYY-MM-DD

4.Unique constraints: Email column is ensured to unique to ensure no duplicates in maid id.

Data Generation: Here python's Faker library is used as mentioned before. Randomization and controlled duplication have been used in columns like specialization and contact_num.

Handling Missing and Duplicate Data: Here certain amount of missing and duplicates were added into noncritical constraint columns to ensure data integrity.



```
# Insert 100 instructors with missing data and duplicates
for _ in range(100):  # 100 instructors
    # Simulating realistic missing data and duplicate values

    # Name (10% chance of being missing, allowing duplicates)
    instructor_name = fake.name() if random.random() > 0.1 else "Unknown"

    # Specialization (5% chance of being missing, allowing duplicates)
    specialization = random.choice(['Mathematics', 'Biology', 'Physics', 'Chemistry', 'Zoology']) if random.random() > 0.05 else "Unknown"

    # Hire Date (more recent dates, 20% chance of being missing, default to "2020-01-01")
    hire_date = fake.date_this_decade() if random.random() > 0.2 else "2020-01-01"

    # Mode of Learning (5% chance of being missing, allowing duplicates)
    mode_of_learning = random.choice(['Online', 'Offline']) if random.random() > 0.05 else "Online"

    # Contact Number (15% chance of being missing, allowing duplicates)
    contact_number = fake.phone_number() if random.random() > 0.15 else "Unknown"

    # Email (generated with the function to avoid duplicates)
    email = generate_email(specialization, used_emails)
```

Fig 2.2 Instructors Table Data Generation

Test for Data Integrity: Extensive testing is performed to validate its integrity:

1.NULL Value Check: Verified that critical columns (instructor_name, hire_date, email) have no Null values.
2.Duplicate Entry Check: No duplicate emails found, and Duplicates are observed in contact_num field for “Unknown” entries, which is intentional
3. observed “No constraint violations” in critical columns (mode_of_learning, hire_date)
4. 26 missing Values are present which are non-constraint fields like specialization, contact_num etc.

```
# Check for NULL values in critical columns
cursor.execute('''
    SELECT COUNT(*)
    FROM Instructors
    WHERE instructor_name IS NULL OR hire_date IS NULL OR email IS NULL
''')
null_values = cursor.fetchone()[0]
print(f'Number of records with NULL values in critical columns: {null_values}')

# Check for duplicate emails
cursor.execute('''
    SELECT email, COUNT(*)
    FROM Instructors
    GROUP BY email
    HAVING COUNT(*) > 1
''')
duplicate_emails = cursor.fetchall()
print(f'Duplicate emails: {duplicate_emails}')


# Check for duplicate contact numbers
cursor.execute('''
    SELECT contact_num, COUNT(*)
    FROM Instructors
    GROUP BY contact_num
    HAVING COUNT(*) > 1;
''')
duplicate_contact_numbers = cursor.fetchall()
print("Duplicate Contact Numbers:", duplicate_contact_numbers)

# Check for duplicate specializations
cursor.execute('''
    SELECT specialization, COUNT(*)
    FROM Instructors
    GROUP BY specialization
    HAVING COUNT(*) > 1;
''')
duplicate_specializations = cursor.fetchall()
print("Duplicate Specializations:", duplicate_specializations)
```

Fig 2.3 Tests for Data Integrity in Instructors Table

```
# Check for duplicate modes of learning
cursor.execute('''
    SELECT mode_of_learning, COUNT(*)
    FROM Instructors
    GROUP BY mode_of_learning
    HAVING COUNT(*) > 1;
''')
duplicate_modes = cursor.fetchall()
print("Duplicate Modes of Learning:", duplicate_modes)


# Check for missing or unrealistic entries
cursor.execute('''
    SELECT COUNT(*)
    FROM Instructors
    WHERE instructor_name = "Unknown" OR specialization = "Unknown" OR contact_num = "Unknown"
''')
missing_or_unrealistic = cursor.fetchone()[0]
print(f'Number of records with missing or unrealistic values: {missing_or_unrealistic}')

# Check for total rows in the table
cursor.execute('SELECT COUNT(*) FROM Instructors')
total_rows = cursor.fetchone()[0]
print(f'Total rows in Instructors table: {total_rows}')

# Close the connection
conn.close()
```

```
Number of records with NULL values in critical columns: 0
Duplicate emails: []
Duplicate Contact Numbers: [('Unknown', 11)]
Duplicate Specializations: [('Biology', 15), ('Chemistry', 19), ('Mathematics', 14), ('Physics', 18), ('U
Duplicate Modes of Learning: [('Offline', 45), ('Online', 55)]
Number of records with missing or unrealistic values: 26
Total rows in Instructors table: 100
```

Fig 2.4 Data Integrity Testing Results

3. Courses Table:

The Courses table serves as a central repository for all courses related to intermediate level of education. This table has various field information like

Nominal: course_id, course_name, specialization, instructor_id (foreign key), course_description, prerequisites
Ordinal: duration
Interval: launch_date
Ratio: Price

Key Constraints and Relationships:

Primary Key: course_id behaves as primary key ensuring each course has unique identifier.

Foreign Key: Here instructor_id is a foreign key referencing the instructor table ensuring each course is linked to a valid instructor, maintaining referential integrity. ON restrict constraint ensures that no course is deleted while an instructor is still linked to it.

Check Constraints: The mode of learning is restricted to only online or offline values enforcing integrity of learning mode.

Nullability: columns like price, duration, course_description allow NULL values to represent missing data reflecting real world scenarios.

```
# Step 1: Create the Courses table with constraints and foreign key handling
cursor.execute('''
    CREATE TABLE IF NOT EXISTS Courses (
        course_id INTEGER PRIMARY KEY,
        course_name TEXT NOT NULL,
        specialization TEXT NOT NULL,
        instructor_id INTEGER NOT NULL,
        mode_of_learning TEXT CHECK (mode_of_learning IN ('Online', 'Offline')),
        price INTEGER,  -- Some courses may not have a price (null)
        duration INTEGER,  -- Some courses may not have a defined duration (null)
        launch_date DATE NOT NULL,
        course_description TEXT,  -- Some courses may not have a description (null)
        prerequisites TEXT,  -- Some courses may not have prerequisites (null)
        FOREIGN KEY (instructor_id) REFERENCES Instructors(instructor_id) ON DELETE RESTRICT
    )
''')
conn.commit()
```

Fig 3.1 Courses Table Creation

Data Generation: column such as course_description, prerequisites allow missing values and duplicate entries for courses were deliberately introduced to reflect real world scenarios where course name can be reused, or courses may have multiple offerings under the same title.

```
# Step 2: Define random data for course creation
course_names = ['Introduction to Algebra', 'Quantum Mechanics', 'Organic Chemistry','Biology for Beginners', 'Advanced Linear Algebra',
                'Mechanics', 'Data Science Fundamentals', 'Advanced Python', 'Machine Learning Basics']
specializations = ['Mathematics', 'Physics', 'Chemistry', 'Biology', 'Zoology']
modes_of_learning = ['Online', 'Offline']
descriptions = ['Learn the basics of this subject', 'Deep dive into advanced topics', 'Introduction to foundational concepts',
                'In-depth understanding of subject matter', None]  # None represents missing descriptions
prerequisites = ['Basic Mathematics', 'Basic Physics', 'Basic Chemistry', 'None', None]  # None represents missing prerequisites

# Generate random dates
def random_date(start_date, end_date):
    return start_date + timedelta(days=random.randint(0, (end_date - start_date).days))

start_date = datetime(2023, 1, 1)
end_date = datetime(2024, 12, 31)

# Step 3: Generate 100 random courses with realistic missing/duplicate data
course_data = []
unique_course_names = set(course_names)  # Ensuring unique course names
unique_specializations = set(specializations)  # Ensuring unique specializations
unique_modes = set(modes_of_learning)  # Ensuring unique modes of learning
for i in range(1, 101):  # Generate 100 courses
    course_name = random.choice(list(unique_course_names)) if random.random() > 0.07 else "Unknown Course"
    specialization = random.choice(list(unique_specializations)) if random.random() > 0.12 else "Unknown Specialization"
    mode_of_learning = random.choice(list(unique_modes)) if random.random() > 0.08 else "Offline"

    # Randomly generate other data with missing values
    price = random.randint(100, 500) if random.random() > 0.2 else None
    duration = random.choice([20, 30, 40, 50]) if random.random() > 0.3 else None
    launch_date = random_date(start_date, end_date).strftime('%Y-%m-%d')
    course_description = random.choice(descriptions)
```

Fig 3.2 Data Generation for Courses Table

Data Integrity and quality checks:

1.Missing Data: Counts in critical columns (e.g.course_name,specialization,mode_of_learning) were checked ensures left with no missing values.

2.Duplicate Data: Multiple courses with the same name were deliberately included, reflecting real world duplication in course. (e.g. Advance Linear Algebra" appears 9 times in dataset)

3.Foreign key integrity: Invalid instructor_id values were not found ensuring every course is linked to existing instructor.

```
# 1. Check for missing values in critical and non-critical columns
cursor.execute("SELECT COUNT(*) FROM Courses WHERE course_name IS NULL")
missing_course_names = cursor.fetchone()[0]

cursor.execute("SELECT COUNT(*) FROM Courses WHERE specialization IS NULL")
missing_specializations = cursor.fetchone()[0]

cursor.execute("SELECT COUNT(*) FROM Courses WHERE mode_of_learning IS NULL")
missing_modes_of_learning = cursor.fetchone()[0]

# 2. Check for duplicates in non-critical columns (course_name, specialization, mode_of_learning)
cursor.execute("SELECT course_name, COUNT(*) FROM Courses GROUP BY course_name HAVING COUNT(*) > 1")
duplicate_course_names = cursor.fetchall()

cursor.execute("SELECT specialization, COUNT(*) FROM Courses GROUP BY specialization HAVING COUNT(*) > 1")
duplicate_specializations = cursor.fetchall()

cursor.execute("SELECT mode_of_learning, COUNT(*) FROM Courses GROUP BY mode_of_learning HAVING COUNT(*) > 1")
duplicate_modes_of_learning = cursor.fetchall()

# 3. Ensure foreign key integrity (no invalid instructor_id)
cursor.execute("SELECT COUNT(*) FROM Courses WHERE instructor_id NOT IN (SELECT instructor_id FROM Instructors)")
invalid_instructor_ids = cursor.fetchone()[0]

# Display results
print(f"Missing course names: {missing_course_names}")
print(f"Missing specializations: {missing_specializations}")
print(f"Missing modes of learning: {missing_modes_of_learning}")
print(f"Duplicate course names: {duplicate_course_names}")
print(f"Duplicate specializations: {duplicate_specializations}")
print(f"Duplicate modes of learning: {duplicate_modes_of_learning}")
print(f"Invalid instructor_id (foreign key constraint check): {invalid_instructor_ids}")

# Close the connection
conn.close()
```

```
Missing course names: 0
Missing specializations: 0
Missing modes of learning: 0
Duplicate course names: [('Advanced Linear Algebra', 9), ('Advanced Python', 8), ('Biology for Beginners', 16), ('Da
1), ('Introduction to Algebra', 13), ('Machine Learning Basics', 14), ('Mechanics', 10), ('Organic Chemistry', 6), (
nknown Course', 3)]
Duplicate specializations: [('Biology', 15), ('Chemistry', 27), ('Mathematics', 11), ('Physics', 16), ('Unknown Spec
20)]
Duplicate modes of learning: [('Offline', 52), ('Online', 48)]
Invalid instructor_id (foreign key constraint check): 0
```

Fig 3.3 Data Integrity Test and Results

4.Offline_Branches Table: This table is designed to store information about offline learning branches. It consists of various attributes such as Nominal: Branch_name, location, contact_info, facility_details
Ordinal: class_schedule Ratio: max_students

```
# Create the Offline_Branches table if it doesn't exist
cursor.execute('''
CREATE TABLE IF NOT EXISTS Offline_Branches (
    branch_id INTEGER PRIMARY KEY,
    branch_name TEXT NOT NULL,
    location TEXT NOT NULL,
    contact_info TEXT,
    class_schedule TEXT,
    instructor_id INTEGER,
    max_students INTEGER NOT NULL,
    facility_details TEXT,
    FOREIGN KEY (instructor_id) REFERENCES Instructors (instructor_id)
)
''')
```

Fig 4.1 Offline_Branches Table Creation

Key Constraints and Data integrity

Primary Key: branch_id acts as primary key ensuring each branch has unique identifier.

Foreign Key: instructor_id reference from instructor table to establish relationship such that each branch is associated with valid instructor.

Not Null Constraints: branch_name, location, max_student are not null constraints ensuring crucial fields are filled in every branch.

Missing Data: contact_info, class_schedule, facility_details fields are deliberately left missing values since in real world all information is not available.

Duplicate Data: some fields like branch_name and location might appear due to random generation of data.

```
# Generate 62 random branches
for i in range(1,62 ):
    branch_name = fake.city() + " Branch"  # Corrected method to generate random branch name
    location = fake.city()  # Generate random location name
    contact_info = fake.phone_number()  # Generate random phone number
    class_schedule = generate_random_schedule()  # Generate random class schedule
    instructor_id = random.randint(1, 100)  # Random instructor ID (assuming 100 instructors exist
    max_students = random.randint(30, 100)  # Random max number of students between 30 and 100
    facility_detail = generate_random_facilities()  # Generate random facility details

    # Randomly introduce missing values for realism (10% chance for missing values)
    if random.random() < 0.1:
        contact_info = None
    if random.random() < 0.1:
        class_schedule = None
    if random.random() < 0.1:
        facility_detail = None
```

Fig 4.2 Data Generation -Offline_Branches

Data Integrity Test: A foreign key check was made, ensuring that instructor_id in offline_Branches table match valid entries in Instructors table confirming no invalid foreign key exists.

```
conn = sqlite3.connect('hybrid_learning@in.db')
cursor = conn.cursor()
# Check for invalid foreign keys (instructor_id) in Offline_Branches
cursor.execute('''
    SELECT b.branch_id, b.branch_name, b.instructor_id
    FROM Offline_Branches b
    LEFT JOIN Instructors i ON b.instructor_id = i.instructor_id
    WHERE i.instructor_id IS NULL
''')

invalid_instructors = cursor.fetchall()

if invalid_instructors:
    print("Invalid foreign keys (instructor_id) found:")
    for row in invalid_instructors:
        print(row)
else:
    print("All instructor_ids are valid.")
# Close the connection
conn.close()
```

```
All instructor_ids are valid.
```

Fig 4.3 Data Integrity Test with Results

5.Enrolments Table: This table bridges connection between Users, Courses and Offline_Branches tables and its structure as follows:
Nominal: user_id, course_id, enrolment_type
Ordinal: payment_status, completion_status
Interval: enrolment_date
Ratio: grade

Key Constraints and Relationships:

Primary Key: enrolment_id is unique identifier tracks individual enrolments and prevents insertion of duplicate records.

Foreign Keys: Enrolments table links with Users, Courses, Offline_Branches tables.

User_id: This foreign key ensures enrolment is linked to a valid user in Users table. It enforces that a student must exist in Users table before enrolling in a course. If a record is inserted with a user-id that does not exists in Users table, database will automatically reject it , maintaining the integrity of the relationship between users and enrolments.

course_id: This foreign key ensures each enrolment is associated with valid course in Courses table. This phenomenon works same as of user_id discussed above

branch_id: This foreign key ensures if student enrolled in offline course should always linked with valid branch_id. For online courses this will be NULL.

Composite key:

Unique(user_id,course_id,enrolment_type):
This composite unique constraint enforces the rule that a user can only enrol in a course one for a given enrolment_type(either online or offline)
E.g. If a student already enrolled in an online course they cannot enrol again in same mode(online) until the record is completed or changed. However, a student can enrol in same course in different mode (offline here).

```
# Create the Enrollments table if it doesn't exist
cursor.execute('''
CREATE TABLE IF NOT EXISTS Enrollments (
    enrollment_id INTEGER PRIMARY KEY,
    user_id INTEGER,
    course_id INTEGER,
    enrollment_date TEXT,
    branch_id INTEGER,
    payment_status TEXT,
    completion_status TEXT,
    grade TEXT,
    enrollment_type TEXT,
    FOREIGN KEY (user_id) REFERENCES Users(user_id),
    FOREIGN KEY (course_id) REFERENCES Courses(course_id),
    FOREIGN KEY (branch_id) REFERENCES Offline_Branches(branch_id),
    UNIQUE(user_id, course_id, enrollment_type)  -- Composite unique constraint
)
''')
```

Fig 5.1 Enrolments Table Creation

Data Generation: Using Faker library

Randomised Data: Each field is populated with randomised values like enrolment_date, payment_status (randomly assigned with possibility of being NULL to reflect incomplete payment information and similarly with completion_status field as well.

```
# Generate random data for Enrollments with missing and duplicate data
enrollment_data = []

# Define possible values
payment_statuses = ['Paid', 'Pending', None]  # Added None for missing data
completion_statuses = ['Completed', 'In Progress', None]  # Added None for missing data
grades = ['A', 'B', 'C', 'D', 'F', None]  # Added None for missing data

# Generate 1000 random enrollments with missing and duplicate data
for i in range(1, 1001):
    user_id = random.randint(1, 100)  # Random user from Users table
    course_id = random.randint(1, 100)  # Random course from Courses table
    enrollment_date = fake.date_this_decade() if random.random() > 0.1 else None  # 10% chance of missing enrollment

    # Get course mode (offline/online) from Courses table
    cursor.execute('SELECT mode_of_learning FROM Courses WHERE course_id = ?', (course_id,))
    course_mode = cursor.fetchone()

    if course_mode and course_mode[0] == 'Offline':
        branch_id = random.randint(1, 100)  # Random branch from Offline_Branches
    else:
        branch_id = None  # No branch for online courses

    payment_status = random.choice(payment_statuses)  # Randomly assign 'Paid', 'Pending', or None
    completion_status = random.choice(completion_statuses)  # Randomly assign 'Completed', 'In Progress', or None
    grade = random.choice(grades)  # Randomly assign grades or None
    enrollment_type = 'Offline' if course_mode and course_mode[0] == 'Offline' else 'Online'
```

Fig 5.2 Data Generation-Enrolments

Data Integrity and Tests:

Missing data: Approximately 5-10% of records consist of missing values in payment_status, completion_status, grade since these are non-critical constraints and common scenario where these data maybe incomplete.

Intentional Duplicates: Intentional duplicates like for every 100th record contains duplicates of previous enrolment entry, and every 5th record has payment status is marked as completed and every 15th record has A grade.

```
# Test 1: Check for duplicate enrollments (user_id, course_id, and enrollment_type)
cursor.execute('''
    SELECT user_id, course_id, enrollment_type, COUNT(*)
    FROM Enrollments
    GROUP BY user_id, course_id, enrollment_type
    HAVING COUNT(*) > 1
''')
duplicates = cursor.fetchall()

if duplicates:
    print("Duplicate enrollments found:")
    for duplicate in duplicates:
        print(duplicate)
else:
    print("No duplicate enrollments.")

# Query to count duplicates in payment_status, completion_status, and grade together
cursor.execute('''
SELECT COUNT(*)
FROM (
    SELECT payment_status, completion_status, grade
    FROM Enrollments
    GROUP BY payment_status, completion_status, grade
    HAVING COUNT(*) > 1
) AS duplicate_group
''')
duplicate_count = cursor.fetchone()[0]  # Fetch the count from the query result
print("Number of duplicate groups (based on payment_status, completion_status, and grade):", duplicate_count)


# Test 2: Check for missing values in payment_status, completion_status, and grade
cursor.execute('''
    SELECT * FROM Enrollments
    WHERE payment_status IS NULL OR completion_status IS NULL OR grade IS NULL
''')
missing_values = cursor.fetchall()

if missing_values:
    print("Enrollments has missing values.")
```

```
    if invalid_courses:
        print("Invalid course_id foreign keys in Enrollments:")
        for row in invalid_courses:
            print(row)
    else:
        print("All course_id foreign keys are valid.")

    # Test 5: check for invalid enrollment_type, user_id,course_id references in Enrollments table
    # Check for invalid enrollment_type (should be either 'Online' or 'Offline')
    cursor.execute('''
        SELECT * FROM Enrollments
        WHERE enrollment_type IS NULL OR (enrollment_type != 'Online' AND enrollment_type != 'Offline')
    ''')
    invalid_enrollment_type = cursor.fetchall()

    if invalid_enrollment_type:
        print("Enrollments with invalid enrollment_type:")
        for row in invalid_enrollment_type:
            print(row)
    else:
        print("All enrollment_type values are valid.")

    # Check for invalid user_id references in Enrollments
    cursor.execute('''
    SELECT e.enrollment_id, e.user_id
    FROM Enrollments e
    LEFT JOIN Users u ON e.user_id = u.user_id
    WHERE u.user_id IS NULL
    ''')
    invalid_users = cursor.fetchall()
    if invalid_users:
        print("Invalid user_id found in Enrollments table:")
        for row in invalid_users:
            print(row)
    else:
        print("All user_ids are valid in Enrollments.")

    # Check for invalid course_id references in Enrollments
    cursor.execute('''
    SELECT e.enrollment_id, e.course_id
    FROM Enrollments e
    LEFT JOIN Courses c ON e.course_id = c.course_id
    WHERE c.course_id IS NULL
    ''')
    invalid_courses = cursor.fetchall()
    if invalid_courses:
        print("Invalid course_id found in Enrollments table:")
        for row in invalid_courses:
            print(row)
    else:
        print("All course_ids are valid in Enrollments.")

# Close the connection
conn.close()
```

```
No duplicate enrollments.
Number of duplicate groups (based on payment_status, completion_status, and grade): 54
Enrollments has missing values.
All user_id foreign keys are valid.
All course_id foreign keys are valid.
All enrollment_type values are valid.
All user_ids are valid in Enrollments.
```

Fig 5.3 Data Integrity Test and Results

6.Reviews table:

This table is designed to capture feedback from users about courses they enrolled in. It contributes to evaluation and improvement of course quality. It comprises of review id, review_date, rating, comment, response, course_id (foreign key), user_id (foreign key)

Key Data constraints:

Primary Key: Ensures each review has a unique identifier
Foreign key (Relationship with other tables):
Users table where it uses user_id as reference ensures that only enrolled users can submit reviews.
Courses table: It references with course_id allowing relationship of aggregation of reviews for each course, enabling instructors and administrators to access course quality
Enriching relationship: Here Enrolment table indirectly complements the Reviews table such that users who leave reviews are expected to have completed a course or on actively enrolled.

```
# Create the Reviews table if it doesn't exist
cursor.execute('''
CREATE TABLE IF NOT EXISTS Reviews (
    review_id INTEGER PRIMARY KEY,
    user_id INTEGER,
    course_id INTEGER,
    review_date TEXT,
    rating INTEGER,
    comment TEXT,
    response TEXT,
    FOREIGN KEY (user_id) REFERENCES Users(user_id),
    FOREIGN KEY (course_id) REFERENCES Courses(course_id)
)
''')
```

Fig 6.1 Reviews Table Creation

Data Generation:

Ratings: Values from 1 to 5 or set NULL (missing values) allows statistical analysis of course performance. Comments are aligned with ratings feedback.

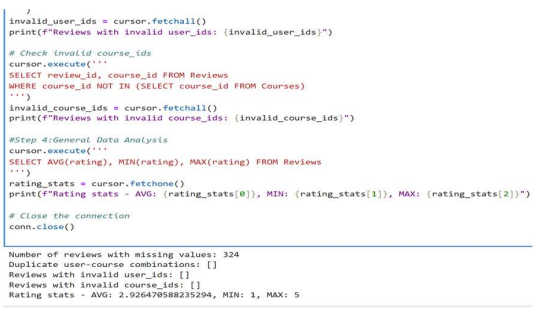
```
# Clear existing data to avoid conflicts
cursor.execute("DELETE FROM Reviews")
conn.commit()

# Define possible values for reviews
ratings = [1, 2, 3, 4, 5, None]  # Ratings between 1 and 5, or None for missing data
responses = [None, "Thank you for your feedback!", "We appreciate your comments.",
             "Your feedback is valuable!", "We'll work on it."]

# Define comment categories
positive_comments = [
    "The course was very informative and well-organized.",
    "I really enjoyed the interactive elements of this course.",
    "Great content and knowledgeable instructors!",
    "This course exceeded my expectations.",
    "Highly recommend for anyone interested in this subject."
]

neutral_comments = [
    "The course was okay but could use more examples.",
    "Decent content, but the pace was a bit slow.",
    "It was fine, but nothing exceptional.",
    "The course met the basic expectations.",
    "Not bad, but not great either."
]
```

```
# Function to generate a comment based on rating
def generate_comment(rating):
    if rating is None:
        return None  # Missing comment
    elif rating >= 4:
        return random.choice(positive_comments)
    elif rating == 3:
        return random.choice(neutral_comments)
    else:
        return random.choice(negative_comments)

# Generate unique random data for Reviews
review_data = []
unique_combinations = set()  # Track unique (user_id, course_id) combinations

for review_id in range(1, 1001):  # Generate 1000 reviews
    user_id = random.randint(1, 100)  # Random user from Users table
    course_id = random.randint(1, 100)  # Random course from Courses table

    # Ensure no duplicate (user_id, course_id) pairs
    while (user_id, course_id) in unique_combinations:
        user_id = random.randint(1, 100)
        course_id = random.randint(1, 100)

    unique_combinations.add((user_id, course_id))  # Track unique pairs

    review_date = fake.date_this_decade()  # Random date within the last decade
    rating = random.choice(ratings)  # Random rating
    comment = generate_comment(rating)  # Generate a realistic comment based on rating
    response = random.choice(responses)  # Random response
```

Fig 6.2 Data Generation- Reviews

Data Integrity Validation:

A range of 324 records of Missing date is found in values of rating, comment, response. No duplicates for user_id, course_id combinations are found to maintain data quality and established value foreign key integrity with user_id and course_id constraints. Also observed average rating of 2.93 with 1 as minimum and 5 as maximum rating.



```
    ''')
invalid_user_ids = cursor.fetchall()
print(f"Reviews with invalid user_ids: {invalid_user_ids}")

# Check invalid course_ids
cursor.execute('''
SELECT review_id, course_id FROM Reviews
WHERE course_id NOT IN (SELECT course_id FROM Courses)
''')
invalid_course_ids = cursor.fetchall()
print(f"Reviews with invalid course_ids: {invalid_course_ids}")

#Step 4:General Data Analysis
cursor.execute('''
SELECT AVG(rating), MIN(rating), MAX(rating) FROM Reviews
''')
rating_stats = cursor.fetchone()
print(f"Rating stats - AVG: {rating_stats[0]}, MIN: {rating_stats[1]}, MAX: {rating_stats[2]}")

# Close the connection
conn.close()
```

```
Number of reviews with missing values: 324
Duplicate user-course combinations: []
Reviews with invalid user_ids: []
Reviews with invalid course_ids: []
Rating stats - AVG: 2.926470588235294, MIN: 1, MAX: 5
```

Fig 6.3 Data Integrity Testing

7.Payments table:

Payments table records financial transactions, linking users and enrolments table ensuring data consistency and usability. It comprises of field Nominal- payment_id, payment_method, payment_status, user_id (foreign key), enrolment_id (foreign key)
Ratio: amount, Interval: payment_date
Key Constraints and Relationship with other tables:

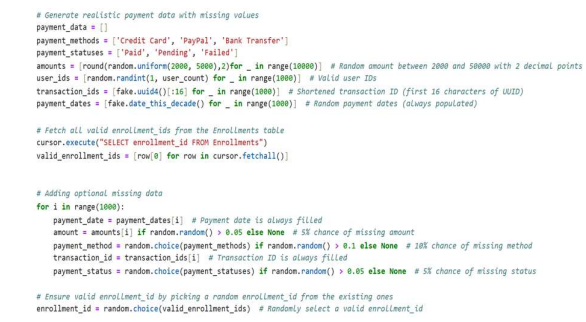
Primary Key: payment_id ensures unique payment_id for each record.
Foreign key (Relationships):
Users: user_id foreign key links each payment to a user in Users table.
Enrolments: enrolment_id foreign key connects payments to specific course enrolments.Courses table is indirectly connected through enrolment table to ensure each payment is associated with specific courses.


```
# Create Payments table with constraints
cursor.execute('''
CREATE TABLE IF NOT EXISTS Payments (
    payment_id INTEGER PRIMARY KEY,
    user_id INTEGER NOT NULL,
    enrollment_id INTEGER NOT NULL,
    payment_date TEXT NOT NULL,
    amount REAL CHECK (amount >= 0),
    payment_method TEXT CHECK (payment_method IN ('Credit Card', 'PayPal', 'Bank Transfer')),
    transaction_id TEXT NOT NULL,
    payment_status TEXT CHECK (payment_status IN ('Paid', 'Pending', 'Failed')),
    FOREIGN KEY (user_id) REFERENCES Users(user_id),
    FOREIGN KEY (enrollment_id) REFERENCES Enrollments(enrollment_id)
)
''')
```

Fig 7.1 Payments Table Creation

Data Generation: using faker library

Missing values were inserted in non-critical fields like payment_method, payment_status, with proportion of 5-10% missing data Duplicates are not added in this table for data integrity issue and having realistic data

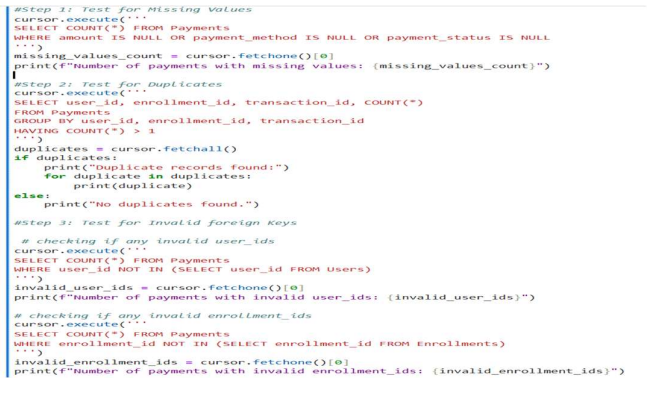


```
# Generate realistic payment data with missing values
payment_data = []
payment_methods = ['Credit Card', 'PayPal', 'Bank Transfer']
payment_statuses = ['Paid', 'Pending', 'Failed']
amounts = [round(random.uniform(2000, 5000),2)for _ in range(10000)]  # Random amount between 2000 and 50000 with 2 decimal points
user_ids = [random.randint(1, user_count) for _ in range(1000)]  # Valid user IDs
transaction_ids = [fake.uuid4()[:16] for _ in range(1000)]  # Shortened transaction ID (first 16 characters of UUID)
payment_dates = [fake.date_this_decade() for _ in range(1000)]  # Random payment dates (always populated)

# Fetch all valid enrollment_ids from the Enrollments table
cursor.execute("SELECT enrollment_id FROM Enrollments")
valid_enrollment_ids = [row[0] for row in cursor.fetchall()]


# Adding optional missing data
for i in range(1000):
    payment_date = payment_dates[i]  # Payment date is always filled
    amount = amounts[i] if random.random() > 0.05 else None  # 5% chance of missing amount
    payment_method = random.choice(payment_methods) if random.random() > 0.1 else None  # 10% chance of missing method
    transaction_id = transaction_ids[i]  # Transaction ID is always filled
    payment_status = random.choice(payment_statuses) if random.random() > 0.05 else None  # 5% chance of missing status

# Ensure valid enrollment_id by picking a random enrollment_id from the existing ones
enrollment_id = random.choice(valid_enrollment_ids)  # Randomly select a valid enrollment_id
```

Fig 7.2 Data Generation-Payments

Data Integrity Testing:

Missing values has been detected across non constraints columns. No duplicate records were found for since unique transaction ids were ensured. No invalid foreign keys were found ensuring data integrity



```
#Step 1: Test for Missing Values
cursor.execute('''
SELECT COUNT(*) FROM Payments
WHERE amount IS NULL OR payment_method IS NULL OR payment_status IS NULL
''')
missing_values_count = cursor.fetchone()[0]
print(f"Number of payments with missing values: {missing_values_count}")

#Step 2: Test for Duplicates
cursor.execute('''
SELECT user_id, enrollment_id, transaction_id, COUNT(*)
FROM Payments
GROUP BY user_id, enrollment_id, transaction_id
HAVING COUNT(*) > 1
''')
duplicates = cursor.fetchall()
if duplicates:
    print("Duplicate records found:")
    for duplicate in duplicates:
        print(duplicate)
else:
    print("No duplicates found.")

#Step 3: Test for Invalid Foreign Keys

 # checking if any invalid user_ids
cursor.execute('''
SELECT COUNT(*) FROM Payments
WHERE user_id NOT IN (SELECT user_id FROM Users)
''')
invalid_user_ids = cursor.fetchone()[0]
print(f"Number of payments with invalid user_ids: {invalid_user_ids}")

# checking if any invalid enrollment_ids
cursor.execute('''
SELECT COUNT(*) FROM Payments
WHERE enrollment_id NOT IN (SELECT enrollment_id FROM Enrollments)
''')
invalid_enrollment_ids = cursor.fetchone()[0]
print(f"Number of payments with invalid enrollment_ids: {invalid_enrollment_ids}")
```

```
# checking if any invalid amounts
cursor.execute('''
SELECT COUNT(*) FROM Payments
WHERE amount < 0
''')
invalid_amounts = cursor.fetchone()[0]
print(f"Number of payments with invalid amount: {invalid_amounts}")

# checking if any invalid payment_methods
cursor.execute('''
SELECT COUNT(*) FROM Payments
WHERE payment_method NOT IN ('Credit Card', 'PayPal', 'Bank Transfer')
''')
invalid_payment_methods = cursor.fetchone()[0]
print(f"Number of payments with invalid payment_method: {invalid_payment_methods}")

#checking if any invalid payment_status
cursor.execute('''
SELECT COUNT(*) FROM Payments
WHERE payment_status NOT IN ('Paid', 'Pending', 'Failed')
''')
invalid_payment_status = cursor.fetchone()[0]
print(f"Number of payments with invalid payment_status: {invalid_payment_status}")
```

```
Number of payments with missing values: 3
No duplicates found.
Number of payments with invalid user_ids: 0
Number of payments with invalid enrollment_ids: 0
Number of payments with invalid amount: 0
Number of payments with invalid payment_method: 0
Number of payments with invalid payment_status: 0
```

Fig 7.3 Data Integrity Testing

Ethical and Data Privacy Considerations across all 7 tables:

Privacy: Sensitive user data such as emails and contact numbers are securely stored. Financial data is kept minimal ensuring no direct storage of sensitive payment information.

Anonymization: Review comments are anonymized to protect user identity, and personal data is not shared outside the necessary scope for course management.

Data Integrity and security are maintained, and main thing is no personal data has used since we used faker python's library to generate data indicating no real-world data has been copied or downloaded.

Scenarios:

1.Retrieve Total Payments by Users for specific course: This query combines Payments, Enrolments, Users and Courses table ensuring proper usage of foreign key constraints.

```
SELECT
    u.name AS user_name,
    c.course_name,
    SUM(p.amount) AS total_paid
FROM
    Payments p
JOIN
    Enrollments e ON p.enrollment_id = e.enrollment_id
JOIN
    Users u ON p.user_id = u.user_id
JOIN
    Courses c ON e.course_id = c.course_id
GROUP BY
    u.user_id, c.course_id;
```

	user_name	course_name	total_paid
1	Sandra Cruz	Mechanics	4401.09
2	Susan Charles	Organic Chemistry	2649.43
3	Devon Taylor	Advanced Python	4181.39
4	Pamela Green	Machine Learning Basics	4246.59
5	Diana Smith	Mechanics	3860.17
6	Corey Mcdowell	Quantum Mechanics	2932.97
7	Barbara Morales	Machine Learning Basics	2198.79
8	Brandon Garcia	Mechanics	3129.06
9	Brenda Mclaughlin	Data Science Fundamentals	4247.0

2. List All Reviews with User and Course Details:Retrieve all reviews with users name,course name,rating and comment by using foreign key relationships

```
SELECT
    r.review_id,
    r.rating,
    r.comment,
    r.review_date,
    u.name AS user_name,
    c.course_name
FROM
    Reviews r
JOIN
    Users u ON r.user_id = u.user_id
JOIN
    Courses c ON r.course_id = c.course_id
ORDER BY
    r.review_date DESC;
```

	review_id	rating	comment	review_date	user_name	course_name
1	311	3	Decent content, but the …	2024-11-14	Phillip King	Introduction to Algebra
2	697	3	Not bad, but not great …	2024-11-13	John Burton	Advanced Python
3	343	NULL	NULL	2024-11-12	Amber Brown	Machine Learning Basics
4	394	2	The course lacked depth …	2024-11-09	Theodore Hill	Machine Learning Basics
5	110	NULL	NULL	2024-11-08	Jordan Kelley	Biology for Beginners
6	868	1	I found the instructor's…	2024-11-06	Robert Brown	Introduction to Algebra
7	668	1	I found the instructor's…	2024-11-05	Justin Bailey	Organic Chemistry
8	733	3	Not bad, but not great …	2024-11-02	Robert Rios …	Introduction to Algebra
9	586	2	Not worth the price I …	2024-10-31	Anna Potts	Introduction to Algebra

3. Total Payments by students for Offline couses in a specific branch:

```
SELECT
    u.name AS student_name,
    c.course_name,
    i.instructor_name AS instructor_name,
    b.branch_name,
    SUM(p.amount) AS total_paid
FROM
    Payments p
JOIN
    Enrollments e USING(enrollment_id)  -- Implicitly handles the join to Enrollments
JOIN
    Users u USING(user_id)  -- Implicitly handles the join to Users
JOIN
    Courses c USING(course_id)  -- Implicitly handles the join to Courses
JOIN
    Instructors i USING(instructor_id)  -- Implicitly handles the join to Instructors
JOIN
    Offline_Branches b USING(branch_id)  -- Implicitly handles the join to Branches
WHERE
    c.mode_of_learning = 'Offline'  -- Only offline courses

GROUP BY
    u.user_id, c.course_id, i.instructor_id, b.branch_id
ORDER BY
    total_paid DESC;
```

	student_name	course_name	instructor_name	branch_name	total_paid
1	Devon Taylor	Advanced Python	Chelsea Heath	Wheelerland Branch	4181.39

4.Analysing Course Enrolment and Payments with Active Transactions: This query aggregates the number of students enrolled and total payments made for each course. It focuses on courses with valid payments filtering out any courses where payment amounts are zero or failed.

```
    c.course_name,
    COUNT(e.enrollment_id) AS student_count,
    COALESCE(SUM(p.amount), 0) AS total_payments
FROM
    Courses c
LEFT JOIN
    Enrollments e ON c.course_id = e.course_id
LEFT JOIN
    Payments p ON e.enrollment_id = p.enrollment_id
WHERE
    p.amount > 0  -- Only count payments greater than 0
GROUP BY
    c.course_id
ORDER BY
    student_count DESC;
```

	course_name	student_count	total_payments
1	Mechanics	2	6989.23
2	Data Science Fundamentals	1	4247.0
3	Mechanics	1	4401.09
4	Advanced Linear Algebra	1	4674.26
5	Machine Learning Basics	1	2198.79
6	Organic Chemistry	1	2649.43
7	Quantum Mechanics	1	2932.97
8	Machine Learning Basics	1	4246.59
9	Advanced Python	1	4181.39

Schema Links:

Payments Table

Key Relationships:

- user_id: Foreign key linking to Users(user_id)
- enrolment_id: Foreign key linking to Enrolments(enrolment_id)

Purpose: Tracks payments made by users for their enrolments.

Reviews Table

Key Relationships:

- user_id: Foreign key linking to Users(user_id)
- course_id: Foreign key linking to Courses(course_id)

Purpose: Allows students to provide feedback on the courses they have taken.

Enrolments Table

Key Relationships:

- user_id: Foreign key linking to Users(user_id)
- course_id: Foreign key linking to Courses(course_id)
- branch_id: Foreign key linking to Offline_Branches(branch_id)
- Composite Key: (user_id, course_id, enrolment_type) ensures uniqueness for each enrolment type for a given student and course.

Purpose: Tracks which users are allowed in specific courses and their branch assignments (for offline users)

Offline_Branches Table

Key Relationships:

- instructor_id: Foreign key linking to Instructors(instructor_id)
- branch_id: Primary key for Offline_Branches table.

Purpose: Defines the branches where offline learning takes place and assigns instructor to these branches.

Courses Table

Key Relationships:

- instructor_id: Foreign key linking to Instructors(instructor_id)
- course_id: Primary Key identifying each course

Purpose: Stores course details and maps them to their respective instructors.

Instructors Table

instructor_id: Primary key uniquely identifies each Instructor

Users Table

user_id: Primary Key uniquely identifies each user(students)

This robust schema ensures data consistency and enforce referential integrity across the database.