MIND MATRIX

GROUP 3

Course Code: BUAN 6320.008

- **Bhavishya Vardhineni**
  bxv240005@utdallas.edu
- **Sarayu Sadineni**
  sxs240116@utdallas.edu
- **Arbbaz Alif Abdul Shafeeq**
  axa240104@utdallas.edu
- **Gurram Pramod Sai Reddy**
  gsr230001@utdallas.edu
- **Nimanshu Nirmal Jain**
  nxj240003@utdallas.edu

Youtube Video Link: https://youtu.be/0IwmxQ3b-OU

Database Layer

A Transformative Learning Management System (LMS)

MIND MATRIX

INDEX

PROJECT OVERVIEW

In today's fast-paced digital era, traditional methods of education are no longer sufficient to keep learners engaged and motivated. Mind Matrix is a next generation Learning Management System (LMS) that aims to disrupt conventional online learning with a personalized, engaging, and gamified approach.

This platform integrates interactive courses with AI-driven personalization to tailor learning experiences to individual users' preferences, abilities, and pace. The objective is to create an educational ecosystem that not only imparts knowledge but also sustains interest and encourages application.

Vision Statement:

To transform education by creating an LMS that actively engages learners, adapts to their needs, and ensures the effective application of knowledge.

Project Objectives

1. **Primary Objective:**
   To develop a comprehensive, scalable database architecture that supports the operations, analytics, and personalization capabilities of the Mind Matrix LMS.

2. **Secondary Objectives:**

   - **Database Design:** Create an efficient schema to manage user profiles, course materials, progress tracking, and analytics.
   - **Advanced SQL Development:** Design complex queries to provide actionable insights into user behavior, performance, and trends.
   - **Automation:** Leverage stored procedures and triggers to automate repetitive database operations, ensuring reliability and efficiency.
   - **Documentation:** Produce detailed documentation to facilitate the integration of the database with future UI and logic layers.
   - **Future-Ready Design:** Enable seamless scalability for a growing user base and integration with AI modules.

Scope and Deliverables

In-Scope:

- **Database Layer Development:** The primary focus will be on designing and implementing the database backend for Mind Matrix.
- **SQL Query Development:** Including the creation of complex queries, stored procedures, and triggers for advanced functionality.
- **Documentation:** Deliver a detailed report comprising E-R diagrams, database schema explanations, query results, and rationale.

Out-of-Scope:

- User Interface (UI) and application logic layers.
- Live deployment and hosting.
- AI algorithm implementation and front-end integrations.

Key Deliverables:

- **Database Schema:** Fully developed and optimized relational database structure.
- **Automated Workflows:** Procedures and triggers to handle common tasks like progress updates, notifications, and report generation.
- **Query Library:** A repository of tested queries for analytics and reporting.
- **Project Documentation:** Comprehensive documentation, including database design, rationale, and usage.

Assumptions

1. Resources for database design and development, including hardware and software tools, will be consistently available.
2. Educational requirements and project objectives will not significantly change during the development cycle.
3. End-users will access the system via a web-based interface, and data access will be managed through SQL queries.

MILESTONES AND TIMELINE

Milestone	Description	Completion Date
Project Definition	Finalization of project scope, requirements, and charter.	September 25, 2024
Research and Analysis	Detailed research into LMS requirements, database tools, and best practices.	October 2, 2024
Schema Development	Design and implementation of the database schema.	October 9, 2024
Query Design	Development of complex queries for operational and analytical needs.	October 23, 2024
Automation Integration	Creation of stored procedures and triggers to automate backend tasks.	October 30, 2024
Testing and debugging	Comprehensive testing of database functionalities.	November 6, 2024
Final Presentation	Project Presentation and demonstration of database.	November 22, 2024
Video Shoot	Creating a fun and engaging skit to highlight the product's value and impact creatively.	November 23, 2024
Final Report Submission	Submission of the final report detailing the database design, queries, automation, and project outcomes.	December 7, 2024

PROJECT WRITE-UP

Introduction

Mind Matrix is envisioned as a transformative Learning Management System (LMS) tailored for today's dynamic educational needs. While most LMS platforms focus on delivering content, Mind Matrix reimagines learning as an engaging, personalized journey. Through its innovative features, such as AI-driven personalization, gamified challenges, and robust analytics, the platform promises to deliver an unparalleled user experience.

The focus of this phase of the project is the development of the **database layer**, the foundation upon which all other features and functionalities are built. This write-up narrates the entire journey of designing Mind Matrix's database system while providing a roadmap for integrating the user interface (UI) and logical layers in future phases.

Target Audience

Mind Matrix is built to cater to a diverse group of users, each with unique needs and expectations:

- **Students**: Seeking interactive and adaptable learning tools to enhance their skills.
- **Educators**: Interested in monitoring learner performance and delivering tailored teaching materials.
- **Corporate Training Departments**: Aiming to upskill employees in an engaging and effective manner.
- **Parents**: Monitoring the progress and achievements of their children on the platform.

By addressing a wide spectrum of users, Mind Matrix establishes itself as a versatile platform suitable for both academic and corporate learning environments.

Scope

The scope of the project is tightly focused on creating a database system that supports the key functionalities of the platform. The database is designed to serve as a strong, scalable backbone that ensures efficient handling of critical processes, including:

1. **User Profiles**:
    - Storing information about learners, instructors, and administrators.
    - Tracking progress, preferences, and achievements.
2. **Course Management**:
    - Managing details about courses, modules, and learning materials.
    - Supporting multimedia content storage and retrieval.
3. **Gamification and Personalization**:
    - Maintaining data for gamified elements such as points, badges, and leaderboards.
    - Storing user behavior data to allow AI algorithms to tailor the learning experience.
4. **Reporting and Analytics**:
    - Providing actionable insights into learner performance and engagement.
    - Generating detailed reports for instructors and administrators.
5. **System Administration**:
    - Enabling seamless role-based access and secure handling of user credentials.
    - Supporting platform scalability as user demand grows.
6. **Administrative Controls:**
    - Implement role-based access controls for secure operation and data integrity.
    - Allow administrators to monitor and manage platform activity effectively.

Design Philosophy

The design philosophy of the Mind Matrix database reflects the core values of **efficiency**, **adaptability**, and **user engagement**. Every decision in the database design was guided by these principles:

1. **Simplicity and Modularity**:
    - The database schema was kept modular to allow seamless updates and scalability.
    - Core entities, such as users, courses, and progress, are well-defined to ensure clarity.
2. **Future-Readiness**:
    - The design accommodates potential future integrations, such as AI modules for deeper personalization or complex reporting dashboards.
    - It is equipped to support increased data loads as the platform scales.
3. **Security and Privacy**:
    - Role-based access control and encrypted storage protect sensitive user data. The design complies with modern data protection standards, ensuring privacy.

Challenges and Considerations

Every project comes with its unique set of challenges and considerations, and the Mind Matrix LMS is no exception. Identifying potential challenges early helps in creating a robust system capable of overcoming them. These considerations are integral to the success of the project:

1. **Data Integrity and Security:**
    - With user data like personal information, progress reports, and test results stored in the database, maintaining its confidentiality and integrity is essential.
    - Implementing encryption for sensitive data ensures secure storage and transmission.

2. **Scalability of the System:**
    - As the user base grows, the database needs to handle an increasing load of concurrent users accessing courses and submitting assignments.
3. **User Experience:**
    - The responsiveness of the platform heavily depends on efficient database queries.
    - Users should experience seamless course loading, real-time updates on leaderboards, and instant feedback on quizzes, which necessitates optimized database design.
4. **Integration with AI and Machine Learning:**
    - The personalization feature of Mind Matrix depends heavily on AI and machine learning models.
    - Ensuring the database structure supports large-scale data collection and processing for training these models.
5. **Testing and Quality Assurance:**
    - The database must undergo rigorous testing to identify potential weaknesses or inefficiencies.
    - Validation of stored procedures, triggers, and queries against edge cases to ensure reliability.
6. **Cross-Platform Accessibility:**
    - Users may access the platform via PCs, tablets, or smartphones, requiring database support for quick, consistent performance across all devices.
    - Backend queries must be optimized to handle requests from various device types without compromising speed or accuracy.
7. **Continuous Feedback Loop:**
    - The database structure is designed to support analytics and reporting, enabling administrators to receive actionable insights.
    - A system that evolves based on user feedback and changing educational needs.

Value Proposition

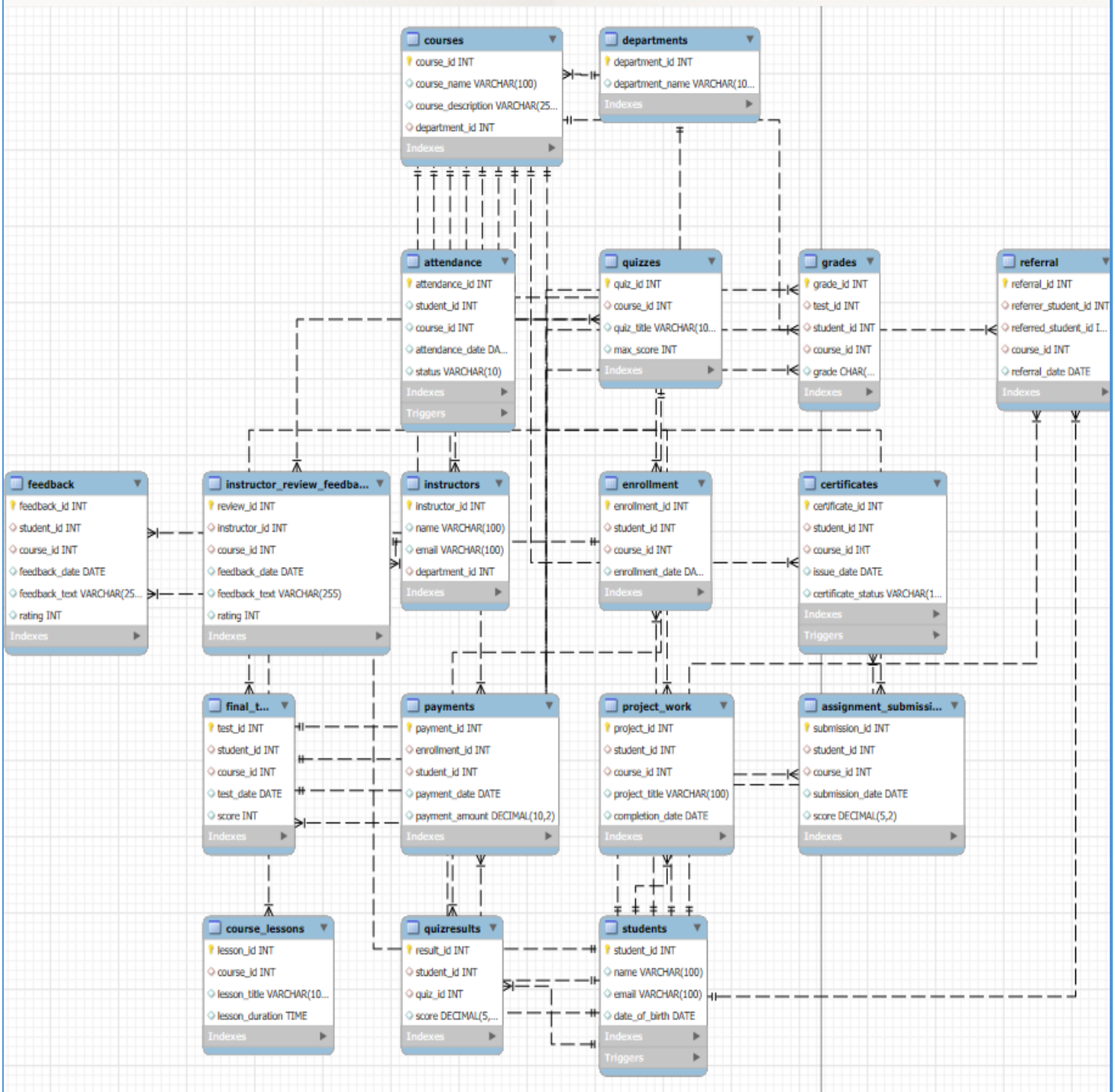
1. **Engagement Through Gamification:**
    - Learning becomes an interactive and enjoyable experience with challenges, leaderboards, and rewards.
    - These gamified elements keep users motivated and encourage consistent participation.
2. **Personalized Learning Paths:**
    - Unlike traditional one-size-fits-all approaches, the platform uses AI to adapt to each learner's pace and style.
    - For instance, a fast learner may receive advanced modules sooner, while others get extra support where needed.
3. **Insightful Analytics:**
    - The platform tracks metrics such as course completion rates, quiz scores, and time spent on tasks.
    - These insights help educators fine-tune course content and identify areas for improvement.
4. **Accessibility:**
    - Designed for global reach, Mind Matrix ensures compatibility across devices and internet conditions.
    - This accessibility makes it a versatile tool for learners in different environments, ensuring education is inclusive.

TABLES

S.NO	TABLE NAME	PRIMARY KEY	FOREIGN KEYS
1	Students	students_id	
2	Departments	departments_id	
3	Courses	courses_id	department_id -> Departments
4	Instructors	instructors_id	department_id -> Departments
5	Course_Lessons	course_lessons_id	course_id -> Courses
6	Enrollment	enrollment_id	student_id -> Students, course_id -> Courses
7	Payments	payments_id	enrollment_id -> Enrollment, student_id -> Students
8	Attendance	attendance_id	
9	Quizzes	quizzes_id	course_id -> Courses
10	QuizResults	quizresults_id	student_id -> Students, quiz_id -> Quizzes
11	Assignment_Submission	assignment_submission_id	student_id -> Students, course_id -> Courses
12	Project_Work	project_work_id	student_id -> Students, course_id -> Courses
13	Final_Test	final_test_id	student_id -> Students, course_id -> Courses
14	Grades	grades_id	test_id -> Final_Test, student_id -> Final_Test, course_id -> Final_Test
15	Certificates	certificates_id	student_id -> Students, course_id -> Courses
16	Feedback	feedback_id	student_id -> Students, course_id -> Courses
17	Instructor_Review_Feedback	instructor_review_feedback_id	instructor_id -> Instructors, course_id -> Courses

E-R DIAGRAMS AND RELATIONSHIPS

- This ER diagram illustrates the relationships between the SQL tables, highlighting the entities, their attributes, and the cardinality of the associations defined in the database schema.



Key Entities and Relationships:

1. **Courses and Departments:**
    - The courses table includes details about courses (e.g., course_id, course_name, department_id).
    - Each course belongs to a specific department (department_id is a foreign key referencing departments.department_id).
2. **Students:**
    - The students table stores student information (e.g., student_id, name, email, date_of_birth).
    - Relationships:
        - Students can enroll in courses (enrollment table).
        - Students may have referrals (referral table).
3. **Enrollment:**
    - Tracks which students are enrolled in which courses (student_id and course_id are foreign keys).
    - Includes additional details like enrollment date.
4. **Quizzes and Grades:**
    - The quizzes table contains information about quizzes associated with courses (course_id as a foreign key).
    - Students' quiz results are stored in grades, which associates student_id, quiz_id, and grade.
5. **Attendance:**
    - attendance table records attendance information for students in a course.
    - Includes status to indicate presence.
6. **Certificates:**
    - The certificates table contains details of certificates issued to students after completing courses.
    - Relationships:
        - Connected to students and courses.

7. **Payments:**
    - Tracks payments made by students for their enrollment.
    - Relationships:
        - Includes enrollment_id as a foreign key to connect payments to specific enrollments.
8. **Final Tests:**
    - The final_test table records the scores of students for final exams (test_id, student_id, course_id).
9. **Feedback and Instructor Reviews:**
    - The feedback table allows students to leave feedback for courses.
    - instructor_review_feedback table tracks reviews of instructors by students.
10. **Referral System:**
    - The referral table tracks referral information between students.
    - Relationships:
        - Links referrer_student_id and referred_student_id.
11. **Course Lessons:**
    - The course_lessons table lists lessons under a specific course (lesson_id, course_id).
12. **Project Work:**
    - Students can complete project work related to a course, recorded in project_work.
13. **Assignment Submissions:**
    - Tracks assignment submissions by students for specific courses.
14. **Quizzes Results:**
    - The quizzes_results table stores individual quiz results for students (result_id, quiz_id, student_id).

Table-Wise Relationships:

1. **Courses**
    - **Many-to-One**: department_id -> departments.department_id (Each course belongs to one department).
    - **One-to-Many**:
        - course_id -> students.course_id (A course can have many students).
        - course_id -> quizzes.course_id (A course can have multiple quizzes).
        - course_id -> course_lessons.course_id (A course can have multiple lessons).
        - course_id -> project_work.course_id (A course can have multiple projects).
2. **Departments**
    - **One-to-Many**: department_id -> courses.department_id (A department can have many courses).
3. **Students**
    - **One-to-Many**:
        - student_id -> feedback.student_id (A student can provide multiple feedback entries).
        - student_id -> attendance.student_id (A student can have multiple attendance records).
        - student_id -> project_work.student_id (A student can complete multiple projects).
    - **Many-to-Many** (via enrollment table): Students and courses (A student can enroll in multiple courses, and a course can have many students).
4. **Quizzes**
    - **Many-to-One**: course_id -> courses.course_id (Each quiz belongs to one course).
    - **One-to-Many**: quiz_id -> grades.quiz_id (A quiz can have multiple grade entries).

5. **Grades**
    - **Many-to-One**:
        - student_id -> students.student_id (Each grade belongs to one student).
        - quiz_id -> quizzes.quiz_id (Each grade belongs to one quiz).
6. **Attendance**
    - **Many-to-One**:
        - student_id -> students.student_id (Each attendance record is for one student).
        - course_id -> courses.course_id (Each attendance record is for one course).
7. **Certificates**
    - **Many-to-One**:
        - student_id -> students.student_id (Each certificate is issued to one student).
        - course_id -> courses.course_id (Each certificate is related to one course).
8. **Payments**
    - **Many-to-One**:
        - enrollment_id -> enrollment.enrollment_id (Each payment corresponds to one enrollment).
9. **Final Tests**
    - **Many-to-One**:
        - student_id -> students.student_id (Each test result is for one student).
        - course_id -> courses.course_id (Each test result is for one course).

10. **Feedback**
    - **Many-to-One**:
        - student_id -> students.student_id (Each feedback entry belongs to one student).
        - course_id -> courses.course_id (Each feedback entry is for one course).
11. **Instructor Review Feedback**
    - **Many-to-One**:
        - student_id -> students.student_id (Each review feedback is given by one student).
        - instructor_id -> instructors.instructor_id (Each review feedback is for one instructor).
12. **Referrals**
    - **Self Many-to-One**:
        - referrer_student_id -> students.student_id (Each referral is initiated by one student).
        - referred_student_id -> students.student_id (Each referral is for one student).
13. **Course Lessons**
    - **Many-to-One**: course_id -> courses.course_id (Each lesson belongs to one course).
14. **Enrollment**
    - **Many-to-Many**:
        - student_id -> students.student_id (A student can enroll in many courses).
        - course_id -> courses.course_id (A course can have many students).

15. **Project Work**

- **Many-to-One**:
  - student_id -> students.student_id (Each project is completed by one student).
  - course_id -> courses.course_id (Each project is related to one course).

16. **Assignment Submissions**

- **Many-to-One**:
  - student_id -> students.student_id (Each assignment is submitted by one student).
  - course_id -> courses.course_id (Each assignment is related to one course).

17. **Quizzes Results**

- **Many-to-One**:
  - student_id -> students.student_id (Each quiz result is for one student).
  - quiz_id -> quizzes.quiz_id (Each quiz result is for one quiz).

18. **Instructors**

- **Many-to-One**: department_id -> departments.department_id (Each instructor belongs to one department).

COMPLEX QUERIES

QUERY 1: Retrieve Top-Performing Students Based on Cumulative Scores Across All Courses:

- **Goal**: This query identifies the top 10 students based on their cumulative quiz scores across all courses. It uses SUM to aggregate scores and orders the result in descending order.
- **Query**:

```
SELECT s.student_id, s.name, SUM(qr.score) AS total_score
FROM Students s
JOIN QuizResults qr ON s.student_id = qr.student_id
JOIN Quizzes q ON qr.quiz_id = q.quiz_id
GROUP BY s.student_id, s.name
ORDER BY total_score DESC
LIMIT 10;
```

```
1779    -- Query 1: Retrieve Top-Performing Students Based on Cumulative Scores Across All Courses:
1780    -- This query identifies the top 10 students based on their cumulative quiz scores across all courses.
1781    -- It uses SUM to aggregate scores and orders the result in descending order.
1782 •  SELECT s.student_id, s.name, SUM(qr.score) AS total_score
1783    FROM Students s
1784    JOIN QuizResults qr ON s.student_id = qr.student_id
1785    JOIN Quizzes q ON qr.quiz_id = q.quiz_id
1786    GROUP BY s.student_id, s.name
1787    ORDER BY total_score DESC
1788    LIMIT 10;
```

- **Result:**

student_id	name	total_score
4	David Brown	363.50
13	Ishaan Verma	357.00
10	Jack Taylor	357.00
6	Frank Thomas	354.50
22	Saanvi Das	354.50
51	Karan Bhat	354.00
19	Arjun Naidu	353.50
16	Aadhya Rao	353.50
25	Kiaan Menon	351.50
1	Alice Johnson	343.50

QUERY 2: Find the average age of students grouped by the year of birth

- **Goal**: To calculate the average age of students born in each year. This query provides insights into the distribution of students' ages for planning and analysis purposes. It uses the current year (CURDATE) to compute age and groups by birth year.
- **Query**:

  SELECT YEAR(date_of_birth) AS birth_year, AVG(YEAR(CURDATE()) - YEAR(date_of_birth)) AS average_age
  FROM Students
  GROUP BY YEAR(date_of_birth);

```
1790    -- Query 2: Find the Average Age of Students Grouped by Year of Birth:
1791    -- This calculates the average age of students grouped by their year of birth.
1792    -- It uses the current year (CURDATE) to compute age and groups by birth year.
1793 •  SELECT YEAR(date_of_birth) AS birth_year, AVG(YEAR(CURDATE()) - YEAR(date_of_birth)) AS average_age
1794    FROM Students
1795    GROUP BY YEAR(date_of_birth);
```

- **Result:**

	birth_year	average_age
▶	1999	25.0000
	2000	24.0000
	1998	26.0000
	2001	23.0000
	1997	27.0000
	2002	22.0000
	1996	28.0000

QUERY 3: Find Courses with the Highest Attendance Rates:

- **Goal**: This query identifies the top 5 courses with the highest attendance. It counts the attendance records where the status is Present and orders the results by count.
- **Query**:

```
SELECT c.course_name, COUNT(a.attendance_id) AS attendance_count
FROM Courses c
JOIN Attendance a ON c.course_id = a.course_id
WHERE a.status = 'Present'
GROUP BY c.course_name
ORDER BY attendance_count DESC
LIMIT 5;
```

```
1797    -- Query 3: Find Courses with the Highest Attendance Rates:
1798    -- This query identifies the top 5 courses with the highest attendance.
1799    -- It counts the attendance records where the status is Present and orders the results by count.
1800 •  SELECT c.course_name, COUNT(a.attendance_id) AS attendance_count
1801    FROM Courses c
1802    JOIN Attendance a ON c.course_id = a.course_id
1803    WHERE a.status = 'Present'
1804    GROUP BY c.course_name
1805    ORDER BY attendance_count DESC
1806    LIMIT 5;
```

- **Result:**

course_name	attendance_count
Algorithms	4
Database Management	4
Cybersecurity	3
Artificial Intelligence	3
Machine Learning	3

QUERY 4: Fetch the top 5 students who enrolled in the most recent courses

- **Goal**: To identify students who have participated in the latest courses, ranked by the enrollment date. This is useful for targeting recently active students for feedback or follow-up. It is sorted by enrollment date in descending order.
- **Query:**

```
SELECT s.name, c.course_name, e.enrollment_date
FROM Enrollment e
INNER JOIN Students s ON e.student_id = s.student_id
INNER JOIN Courses c ON e.course_id = c.course_id
ORDER BY e.enrollment_date DESC
LIMIT 5;
```

```
1808    -- Query 4: Fetch the Top 5 Students Who Enrolled in the Most Recent Courses:
1809    -- This retrieves the top 5 students who enrolled in the most recently offered courses,
1810    -- sorted by enrollment date in descending order.
1811    SELECT s.name, c.course_name, e.enrollment_date
1812    FROM Enrollment e
1813    INNER JOIN Students s ON e.student_id = s.student_id
1814    INNER JOIN Courses c ON e.course_id = c.course_id
1815    ORDER BY e.enrollment_date DESC
1816    LIMIT 5;
```

- **Result:**

name	course_name	enrollment_date
Shivani Khatri	Sociology Basics	2024-12-18
Myra Singh	Operations Management	2024-12-15
Prisha Singh	Operating Systems	2024-12-05
Rhea Basu	Art History	2024-12-01
Bhavya Sethi	Art History	2024-11-25

QUERY 5: Calculate the total number of students enrolled in each course

- **Goal**: This query counts how many students are enrolled in each course to help assess course popularity and resource allocation needs. Using a LEFT JOIN to include all courses, even those without enrollments.
- **Query**:

```
SELECT c.course_name, COUNT(e.student_id) AS student_count
FROM Courses c
LEFT JOIN Enrollment e ON c.course_id = e.course_id
GROUP BY c.course_name;
```

```
1818    -- Query 5: Calculate the Total Number of Students Enrolled in Each Course:
1819    -- This query calculates the total number of students enrolled in each course
1820    -- using a LEFT JOIN to include all courses, even those without enrollments.
1821    SELECT c.course_name, COUNT(e.student_id) AS student_count
1822    FROM Courses c
1823    LEFT JOIN Enrollment e ON c.course_id = e.course_id
1824    GROUP BY c.course_name;
```

- **Result:**

course_name	student_count
Data Structures	3
Algorithms	3
Operating Systems	3
Artificial Intelligence	3
Machine Learning	3
Networking Basics	3
Database Management	3
Web Development	3
Cybersecurity	3
Cloud Computing	3
Introduction to Philoso...	3
World History	3
Creative Writing	3
Sociology Basics	3
Art History	3
Marketing Fundamentals	3
Financial Accounting	3
Human Resource Man...	3
Business Analytics	3
Operations Management	3
Engineering Mechanics	2
Thermodynamics	2
Electrical Circuits	2
Fluid Mechanics	2
Control Systems	2

QUERY 6: Identify students who have not enrolled in any course

- **Goal**: To find students who have not yet enrolled in any course, helping target them for promotional campaigns or onboarding assistance.
- **Query**:

SELECT s.name, s.email
FROM Students s
LEFT JOIN Enrollment e ON s.student_id = e.student_id
WHERE e.student_id IS NULL;

```
1826    -- Query 6: Identify Students Who Have Not Enrolled in Any Course:
1827    -- This identifies students who have not enrolled in any course
1828    -- by looking for NULL values in the Enrollment table.
1829 •  SELECT s.name, s.email
1830    FROM Students s
1831    LEFT JOIN Enrollment e ON s.student_id = e.student_id
1832    WHERE e.student_id IS NULL;
```

- **Results:** Lists students without any enrollments along with their email addresses i.e. 0

	name	email

QUERY 7: Find the Courses Where Enrollment is Greater Than 2:

- **Goal**: This query lists courses where more than 2 students are enrolled. The HAVING clause filters aggregated results.
- **Query**:

SELECT c.course_name, COUNT(e.student_id) AS enrollment_count
FROM Courses c
INNER JOIN Enrollment e ON c.course_id = e.course_id
GROUP BY c.course_name
HAVING enrollment_count > 2;

```
1834    -- Query 7: Find the Courses Where Enrollment is Greater Than 2:
1835    -- This query lists courses where more than 2 students are enrolled.
1836    -- The HAVING clause filters aggregated results.
1837 •  SELECT c.course_name, COUNT(e.student_id) AS enrollment_count
1838    FROM Courses c
1839    INNER JOIN Enrollment e ON c.course_id = e.course_id
1840    GROUP BY c.course_name
1841    HAVING enrollment_count > 2;
```

- **Result:**

course_name	enrollment_count
Data Structures	3
Algorithms	3
Operating Systems	3
Artificial Intelligence	3
Machine Learning	3
Networking Basics	3
Database Management	3
Web Development	3
Cybersecurity	3
Cloud Computing	3
Introduction to Philoso...	3
World History	3

QUERY 8: Calculate Total Revenue Generated by Each Department:

- **Goal**: It's to calculate the **total revenue generated by each department** in an educational institution or similar system. This is achieved by summing up all payments made for enrollments in courses offered by each department.
- **Query**:

```
SELECT d.department_name, SUM(p.payment_amount) AS total_revenue
FROM Departments d
JOIN Courses c ON d.department_id = c.department_id
JOIN Enrollment e ON c.course_id = e.course_id
JOIN Payments p ON e.enrollment_id = p.enrollment_id
GROUP BY d.department_name
ORDER BY total_revenue DESC;
```

```
1843    -- Query 8: Calculate Total Revenue Generated by Each Department:
1844    -- This calculates the total revenue generated by each department by summing up payments
1845    -- linked through courses and enrollments.
1846    SELECT d.department_name, SUM(p.payment_amount) AS total_revenue
1847    FROM Departments d
1848    JOIN Courses c ON d.department_id = c.department_id
1849    JOIN Enrollment e ON c.course_id = e.course_id
1850    JOIN Payments p ON e.enrollment_id = p.enrollment_id
1851    GROUP BY d.department_name
1852    ORDER BY total_revenue DESC;
```

- **Result:**

department_name	total_revenue
Humanities	1194.00
Information Technology	1162.00
Business Administration	1156.00
Computer Science	1149.00
Engineering	811.00

QUERY 9: Analyze Enrollment Trends Over Time:

- **Goal**: It is to **analyze enrollment trends over time** by calculating the total number of enrollments on each specific date. This information helps understand how enrollments fluctuate over time and can be used for planning and decision-making.
- **Query:**

SELECT DATE(e.enrollment_date) AS enrollment_date,
COUNT(e.enrollment_id) AS total_enrollments
FROM Enrollment e
GROUP BY DATE(e.enrollment_date)
ORDER BY enrollment_date;

```
1854    -- Query 9: Analyze Enrollment Trends Over Time:
1855    -- This analyzes trends in enrollments by counting the number of enrollments for each date.
1856 •  SELECT DATE(e.enrollment_date) AS enrollment_date, COUNT(e.enrollment_id) AS total_enrollments
1857    FROM Enrollment e
1858    GROUP BY DATE(e.enrollment_date)
1859    ORDER BY enrollment date;
```

- **Result:**

enrollment_date	total_enrollments
2024-01-05	1
2024-01-12	1
2024-01-15	2
2024-01-20	1
2024-01-22	1
2024-01-25	1
2024-02-10	1
2024-02-12	1
2024-02-15	1
2024-02-28	2
2024-03-05	2
2024-03-10	1

QUERY 10: Aggregate Average Feedback Ratings for Each Instructor:

- **Goal:** Its to **calculate the average feedback rating for each instructor** based on student feedback and rank the instructors by their average rating in descending order. This helps identify instructors with the highest student satisfaction levels.
- **Query:**

```
SELECT i.name AS instructor_name, AVG(irf.rating) AS average_rating
FROM Instructors i
JOIN Instructor_Review_Feedback irf ON i.instructor_id = irf.instructor_id
GROUP BY i.name
ORDER BY average_rating DESC;
```

```
1861    -- Query 10: Aggregate Average Feedback Ratings for Each Instructor:
1862    -- This calculates the average feedback ratings for each instructor based on student feedback
1863    -- and orders results by the highest rating.
1864    SELECT i.name AS instructor_name, AVG(irf.rating) AS average_rating
1865    FROM Instructors i
1866    JOIN Instructor_Review_Feedback irf ON i.instructor_id = irf.instructor_id
1867    GROUP BY i.name
1868    ORDER BY average_rating DESC;
```


- **Result:**

instructor_name	average_rating
Mr. Frank Wilson	5.0000
Ms. Ivy Adams	5.0000
Dr. Leo Turner	5.0000
Prof. Paul Green	5.0000
Dr. Uma Taylor	5.0000
Dr. Xavier Thomas	5.0000
Dr. Zoe King	5.0000
Dr. Alice Johnson	4.6667
Dr. Emma Thomas	4.6667
Mr. Henry Scott	4.6667
Dr. Karen Hill	4.6667
Dr. Nancy Wright	4.6667

STORED PROCEDURES

Stored Procedure 1: Retrieve Payment History for a Student

- **Explanation:** The GetStudentPayments procedure retrieves the payment history for a specific student, detailing the payment ID, date, amount, and associated course name. It accepts a studentId as input and uses joins between the Payments, Enrollment, and Courses tables to link payments to the courses the student enrolled in. The results are filtered by the given student ID and ordered by payment date in descending order, displaying the most recent payments first. This procedure is useful for students to view their payment details and for administrators to review a student's financial records efficiently.
- **Query:**

```
DROP PROCEDURE IF EXISTS GetStudentPayments;
DELIMITER //
CREATE PROCEDURE GetStudentPayments(IN studentId INT)
BEGIN
  SELECT
    p.payment_id,
    p.payment_date,
    p.payment_amount,
    c.course_name
  FROM Payments p
  JOIN Enrollment e ON p.enrollment_id = e.enrollment_id
  JOIN Courses c ON e.course_id = c.course_id
  WHERE p.student_id = studentId
  ORDER BY p.payment_date DESC;
END;
//
DELIMITER ;
```

Test Procedure 1: Retrieve Payment History for a Student

- This test retrieves payment details for a specific student, including the payment ID, date, amount, and course name.

```
CALL GetStudentPayments(1);
```

```sql
-- Procedure 1: Retrieve Payment History for a Student
DROP PROCEDURE IF EXISTS GetStudentPayments;
DELIMITER //
CREATE PROCEDURE GetStudentPayments(IN studentId INT)
BEGIN
    SELECT
        p.payment_id,
        p.payment_date,
        p.payment_amount,
        c.course_name
    FROM Payments p
    JOIN Enrollment e ON p.enrollment_id = e.enrollment_id
    JOIN Courses c ON e.course_id = c.course_id
    WHERE p.student_id = studentId
    ORDER BY p.payment_date DESC;
END;
//
DELIMITER ;

-- Test Procedure 1: Retrieve Payment History for a Student
-- This test retrieves payment details for a specific student, including the payment ID, date, amount, and course name.
CALL GetStudentPayments(1);
```

- **Result:**

payment_id	payment_date	payment_amount	course_name
40001	2024-01-18	85.00	Data Structures

Stored Procedure 2: Get Feedback Summary for a Course

- **Explanation**: The GetCourseFeedback procedure generates a summary of feedback for a specific course, providing insights into its overall performance and student satisfaction. It accepts a courseId as input and retrieves the course name, average feedback rating, total number of feedback entries, and all feedback comments concatenated into a single string separated by semicolons. Using a join between the Courses and Feedback tables, it filters results based on the specified course ID and groups them by the course name. This procedure is valuable for understanding course quality, enabling administrators or instructors to review consolidated feedback efficiently.

- **Query**:

```
DELIMITER //
CREATE PROCEDURE EnrollStudent(IN student_id INT, IN course_id
INT, IN enroll_date DATE)
BEGIN
  INSERT INTO Enrollment (student_id, course_id, enrollment_date)
VALUES (student_id, course_id, enroll_date);
END //
DELIMITER ;
```

```
1901    -- Procedure 2: Get Feedback Summary for a Course
1902    DROP PROCEDURE IF EXISTS GetCourseFeedback;
1903    DELIMITER //
1904    CREATE PROCEDURE GetCourseFeedback(IN courseId INT)
1905    BEGIN
1906        SELECT
1907            c.course_name,
1908            AVG(f.rating) AS average_rating,
1909            COUNT(f.feedback_id) AS total_feedback,
1910            GROUP_CONCAT(f.feedback_text SEPARATOR '; ') AS all_feedback
1911        FROM Courses c
1912        JOIN Feedback f ON c.course_id = f.course_id
1913        WHERE c.course_id = courseId
1914        GROUP BY c.course_name;
1915    END;
1916    //
1917    DELIMITER ;
1918
1919    -- Test Procedure 2: Get Feedback Summary for a Course
1920    -- This test retrieves feedback summary for a specific course, including the average rating, total feedback count, and concatenated feedback comments.
1921    CALL GetCourseFeedback(1011);
1922
```

- Results:

course_name	average_rating	total_feedback	all_feedback
Introduction to Philosophy	4.3333	3	Philosophy concepts made interesting.; I enjoy...

Stored Procedure 3: Analyze Referral Data for a Course

- **Explanation**: The GetReferralSummary procedure provides an analysis of referral data for a specific course, detailing the names of the referrer and referred students, along with the referral date. It accepts a courseId as input and uses joins between the Referral and Students tables to link referrals with student details. The results are filtered by the specified course ID and ordered by referral date in descending order, highlighting the most recent referrals first. This procedure is useful for tracking and analyzing referral activities, enabling insights into how students promote courses to their peers.

- **Query**:

```
-- Procedure 3: Analyze Referral Data for a Course
DROP PROCEDURE IF EXISTS GetReferralSummary;
DELIMITER //
CREATE PROCEDURE GetReferralSummary(IN courseId INT)
BEGIN
  SELECT
    s1.name AS referrer_name,
    s2.name AS referred_name,
    r.referral_date
  FROM Referral r
  JOIN Students s1 ON r.referrer_student_id = s1.student_id
  JOIN Students s2 ON r.referred_student_id = s2.student_id
  WHERE r.course_id = courseId
  ORDER BY r.referral_date DESC;
END;
//
DELIMITER ;
```

Test Procedure 3: Analyze Referral Data for a Course

- This test retrieves referral data for a specific course, listing referrer and referred student names along with the referral date.

```
CALL GetReferralSummary(1015);
```

```sql
-- Procedure 3: Analyze Referral Data for a Course
DROP PROCEDURE IF EXISTS GetReferralSummary;
DELIMITER //
CREATE PROCEDURE GetReferralSummary(IN courseId INT)
BEGIN
    SELECT
        s1.name AS referrer_name,
        s2.name AS referred_name,
        r.referral_date
    FROM Referral r
    JOIN Students s1 ON r.referrer_student_id = s1.student_id
    JOIN Students s2 ON r.referred_student_id = s2.student_id
    WHERE r.course_id = courseId
    ORDER BY r.referral_date DESC;
END;
//
DELIMITER ;

-- Test Procedure 3: Analyze Referral Data for a Course
-- This test retrieves referral data for a specific course, listing referrer and referred student names along with the referral date.
CALL GetReferralSummary(1015);
```

- **Result:**

	referrer_name	referred_name	referral_date
▶	Bhavya Sethi	Vihaan Mehta	2025-06-01
	Vihaan Mehta	Bhavya Sethi	2024-09-25

FUNCTIONS

Function 1: Calculate Average Quiz Score for a Student

- **Explanation:** The GetAverageQuizScore function calculates the average quiz score for a specific student based on their quiz results. It takes a studentId as input and retrieves the average of all scores from the QuizResults table where the student_id matches the input. The function uses the COALESCE function to return 0 if no quiz scores are found for the student. Designed as deterministic, it ensures consistent results for the same input, making it ideal for individual performance evaluation or integration into broader reporting and analytics systems.
- **Query:**

```
DROP FUNCTION IF EXISTS GetAverageQuizScore;
DELIMITER //
CREATE FUNCTION GetAverageQuizScore(studentId INT)
RETURNS DECIMAL(5, 2)
DETERMINISTIC
BEGIN
  DECLARE avg_score DECIMAL(5, 2);
  SELECT AVG(score) INTO avg_score
  FROM QuizResults
  WHERE student_id = studentId;
  RETURN COALESCE(avg_score, 0);
END;
//
DELIMITER ;
```

Test Function 1: Calculate Average Quiz Score for a Student

- This test calculates the average quiz score for a specific student based on their quiz results.

```
SELECT GetAverageQuizScore(1) AS average_quiz_score;
```

```sql
-- Function 1: Calculate Average Quiz Score for a Student
DROP FUNCTION IF EXISTS GetAverageQuizScore;
DELIMITER //
CREATE FUNCTION GetAverageQuizScore(studentId INT)
RETURNS DECIMAL(5, 2)
DETERMINISTIC
BEGIN
    DECLARE avg_score DECIMAL(5, 2);
    SELECT AVG(score) INTO avg_score
    FROM QuizResults
    WHERE student_id = studentId;
    RETURN COALESCE(avg_score, 0);
END;
//
DELIMITER ;

-- Test Function 1: Calculate Average Quiz Score for a Student
-- This test calculates the average quiz score for a specific student based on their quiz results.
SELECT GetAverageQuizScore(1) AS average_quiz_score;
```

- **Usage Example:**

average_quiz_score
85.88

Function 2: Get Total Revenue Generated by a Course

- **Explanation:** The GetCourseRevenue function calculates the total revenue generated by a specific course based on payments received. It takes a courseId as input and sums up all payment amounts from the Payments table by joining it with the Enrollment table to link payments to the specified course. If no payments are found for the course, it uses the COALESCE function to return 0. Designed as a deterministic function, it ensures consistent results for the same input, making it useful for financial analysis, revenue tracking, or evaluating the profitability of individual courses.

- **Query:**

```
DROP FUNCTION IF EXISTS GetCourseRevenue;
DELIMITER //
CREATE FUNCTION GetCourseRevenue(courseId INT)
RETURNS DECIMAL(10, 2)
DETERMINISTIC
BEGIN
  DECLARE total_revenue DECIMAL(10, 2);
  SELECT SUM(p.payment_amount) INTO total_revenue
  FROM Payments p
  JOIN Enrollment e ON p.enrollment_id = e.enrollment_id
  WHERE e.course_id = courseId;
  RETURN COALESCE(total_revenue, 0);
END;
//
DELIMITER ;
```

Test Function 2: Get Total Revenue Generated by a Course

- This test calculates the total revenue generated by a specific course based on payments received.

```
SELECT GetCourseRevenue(1011) AS total_revenue;
```

```sql
-- Function 2: Get Total Revenue Generated by a Course
DROP FUNCTION IF EXISTS GetCourseRevenue;
DELIMITER //
CREATE FUNCTION GetCourseRevenue(courseId INT)
RETURNS DECIMAL(10, 2)
DETERMINISTIC
BEGIN
    DECLARE total_revenue DECIMAL(10, 2);
    SELECT SUM(p.payment_amount) INTO total_revenue
    FROM Payments p
    JOIN Enrollment e ON p.enrollment_id = e.enrollment_id
    WHERE e.course_id = courseId;
    RETURN COALESCE(total_revenue, 0);
END;
//
DELIMITER ;

-- Test Function 2: Get Total Revenue Generated by a Course
-- This test calculates the total revenue generated by a specific course based on payments received.
SELECT GetCourseRevenue(1011) AS total_revenue;
```

- **Usage Example:**

total_revenue
195.00

Function 3: Calculate Attendance Percentage for a Student in a Course

- **Explanation:** The GetAttendancePercentage function calculates a student's attendance percentage for a specific course. It takes two input parameters, studentId and courseId, and determines the total number of classes for the course and the number of classes the student attended (status = 'Present') from the Attendance table. The attendance percentage is calculated as the ratio of attended classes to total classes, multiplied by 100. If no attendance data is available, the COALESCE function ensures that the function returns 0. This deterministic function is useful for tracking student participation, identifying attendance issues, and supporting academic reporting.
- **Query:**

```
DROP FUNCTION IF EXISTS GetAttendancePercentage;
DELIMITER //
CREATE FUNCTION GetAttendancePercentage(studentId INT, courseId INT)
RETURNS DECIMAL(5, 2)
DETERMINISTIC
BEGIN
  DECLARE total_classes INT;
  DECLARE attended_classes INT;
  DECLARE attendance_percentage DECIMAL(5, 2);

  SELECT COUNT(*) INTO total_classes
  FROM Attendance
  WHERE course_id = courseId;

  SELECT COUNT(*) INTO attended_classes
  FROM Attendance
  WHERE student_id = studentId AND course_id = courseId AND status =
'Present';

  SET attendance_percentage = (attended_classes / total_classes) * 100;
  RETURN COALESCE(attendance_percentage, 0);
END;
//
DELIMITER ;
```

Test Function 3: Calculate Attendance Percentage for a Student in a Course

- This test calculates the attendance percentage of a student for a specific course.

SELECT GetAttendancePercentage(24, 1024) AS attendance_percentage;

```
1997    -- Function 3: Calculate Attendance Percentage for a Student in a Course
1998    DROP FUNCTION IF EXISTS GetAttendancePercentage;
1999    DELIMITER //
2000    CREATE FUNCTION GetAttendancePercentage(studentId INT, courseId INT)
2001    RETURNS DECIMAL(5, 2)
2002    DETERMINISTIC
2003    BEGIN
2004        DECLARE total_classes INT;
2005        DECLARE attended_classes INT;
2006        DECLARE attendance_percentage DECIMAL(5, 2);
2007
2008        SELECT COUNT(*) INTO total_classes
2009        FROM Attendance
2010        WHERE course_id = courseId;
2011
2012        SELECT COUNT(*) INTO attended_classes
2013        FROM Attendance
2014        WHERE student_id = studentId AND course_id = courseId AND status = 'Present';
2015
2016        SET attendance_percentage = (attended_classes / total_classes) * 100;
2017        RETURN COALESCE(attendance_percentage, 0);
2018    END;
2019    //
2020    DELIMITER ;
```

- **Usage Example:**

attendance_percentage
50.00

TRIGGERS

Trigger 1: Automatically log when a new student is added.

- **Explanation:**

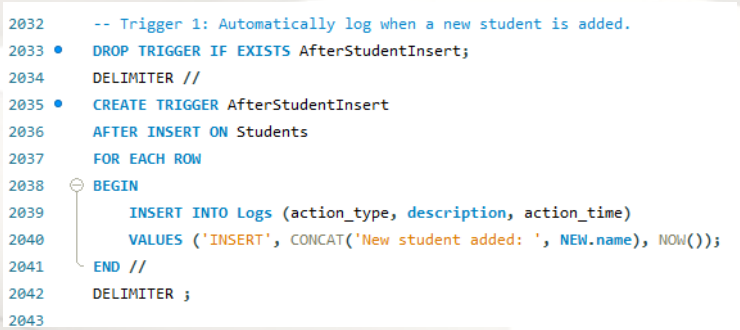
The AfterStudentInsert trigger automatically logs an entry in the Logs table whenever a new student is added to the Students table. This trigger executes **after an INSERT operation** on the Students table and captures relevant details by inserting a log entry with:

- **Action type:** 'INSERT' to indicate the operation.
- **Description:** A concatenated message stating 'New student added:' followed by the name of the new student (from the NEW keyword).
- **Action time:** The current timestamp (NOW()).

This trigger is useful for maintaining an audit trail, tracking database changes, and ensuring accountability for student-related updates.

- **Query:**

```
DROP TRIGGER IF EXISTS AfterStudentInsert;
DELIMITER //
CREATE TRIGGER AfterStudentInsert
AFTER INSERT ON Students
FOR EACH ROW
BEGIN
  INSERT INTO Logs (action_type, description, action_time)
  VALUES ('INSERT', CONCAT('New student added: ', NEW.name), NOW());
END //
DELIMITER ;
```



Trigger 2: Update Attendance Summary

- **Explanation:**

The update_attendance_summary trigger automatically updates the total_attendance count in the Courses table whenever a new attendance record with a 'Present' status is inserted into the Attendance table. This trigger executes **after an INSERT operation** and performs the following steps:

- Checks if the status of the new attendance record (NEW.status) is 'Present'.
- If true, increments the total_attendance field for the corresponding course in the Courses table, identified by the course_id from the new attendance record (NEW.course_id).

This trigger ensures that the attendance summary in the Courses table is automatically and accurately maintained, eliminating the need for manual updates and supporting real-time attendance tracking.

- **Query:**

```
DROP TRIGGER IF EXISTS update_attendance_summary;
DELIMITER //
CREATE TRIGGER update_attendance_summary
AFTER INSERT ON Attendance
FOR EACH ROW
BEGIN
  IF NEW.status = 'Present' THEN
    UPDATE Courses
    SET total_attendance = total_attendance + 1
    WHERE course_id = NEW.course_id;
  END IF;
END;
//
DELIMITER ;
```

```
2044    -- Trigger 2: Update Attendance Summary
2045    DROP TRIGGER IF EXISTS update_attendance_summary;
2046    DELIMITER //
2047    CREATE TRIGGER update_attendance_summary
2048    AFTER INSERT ON Attendance
2049    FOR EACH ROW
2050    BEGIN
2051        IF NEW.status = 'Present' THEN
2052            UPDATE Courses
2053            SET total_attendance = total_attendance + 1
2054            WHERE course_id = NEW.course_id;
2055        END IF;
2056    END;
2057    //
2058    DELIMITER ;
```

Trigger 3: Track Certificate Issuance Log whenever a certificate is issued to a student.

- **Explanation:**

The log_certificate trigger automatically logs certificate issuance details into the AuditLog table whenever a new certificate record is added to the Certificates table. This trigger executes **after an INSERT operation** on the Certificates table and records the following in the AuditLog:

- **Action type:** 'INSERT' to indicate the type of database operation.
- **Table name:** 'Certificates' to specify the affected table.
- **Record ID:** The ID of the newly inserted certificate (NEW.certificate_id).
- **Action time:** The current timestamp (CURRENT_TIMESTAMP) when the trigger is executed.
- **Details:** A concatenated string that includes the student_id, course_id, and certificate_status from the newly inserted certificate record.

This trigger provides a robust mechanism for tracking certificate issuance, ensuring an accurate and auditable history for compliance, transparency, and operational monitoring.

- **Query:**

```
DROP TRIGGER IF EXISTS log_certificate;

CREATE TRIGGER log_certificate

AFTER INSERT ON Certificates

FOR EACH ROW

INSERT INTO AuditLog (action_type, table_name, record_id, action_time, details)

VALUES ('INSERT', 'Certificates', NEW.certificate_id, CURRENT_TIMESTAMP,

    CONCAT('Student ID: ', NEW.student_id, ', Course ID: ', NEW.course_id,

        ', Status: ', NEW.certificate_status));
```

```
2061 •  -- Trigger 3: Track Certificate Issuance Log whenever a certificate is issued to a student.
2062    DROP TRIGGER IF EXISTS log_certificate;
2063 •  CREATE TRIGGER log_certificate
2064    AFTER INSERT ON Certificates
2065    FOR EACH ROW
2066    INSERT INTO AuditLog (action_type, table_name, record_id, action_time, details)
2067    VALUES ('INSERT', 'Certificates', NEW.certificate_id, CURRENT_TIMESTAMP,
2068            CONCAT('Student ID: ', NEW.student_id, ', Course ID: ', NEW.course_id,
2069                   ', Status: ', NEW.certificate_status));
2070
```

CONCLUSION

- The successful completion of this project highlights the importance of a well-structured database system in addressing the data management needs of an academic institution. By following a systematic approach to database design, we developed a scalable, efficient, and secure system that meets the project's objectives.
- The key achievements of this project include the creation of a normalized database schema, implementation of complex queries for actionable insights, and automation of critical operations using stored procedures, functions, and triggers. These features ensure that the database is not only functional but also capable of maintaining data integrity and consistency.
- Moreover, the inclusion of audit mechanisms through triggers enables transparency and accountability, ensuring that all significant database events are logged for future reference. This is crucial for meeting operational and regulatory requirements.
- The project also emphasized collaboration, documentation, and testing, which were essential to the system's robustness and reliability. Through detailed documentation and logical explanations, the system is prepared for future scalability, such as integration with user interfaces or external systems.
- In conclusion, this database system serves as a solid foundation for further development into a comprehensive student management solution. It demonstrates how database design and advanced SQL techniques can address real-world challenges effectively, paving the way for enhanced operational efficiency and better decision-making in academic environments. This project is a significant step towards creating a full-fledged product that meets institutional needs while ensuring adaptability to future requirements.