BSc/HND Computing Portfolio

Databases and Application Development ACCA5026

Nicolae Ciprian Panait 2311205

Part A

Guidelines:

1. Database Design (30%)

Main Entities and Attributes

Students:

-**Attributes**: Student_ID (PK), University_ID, First_Name, Last_Name, Email, Enrollment_Year

-Purpose: Acts as a representative for individual students who might engage in activities and join several clubs.

Clubs:

Attributes: Club_ID (PK), Club_Name, Category_ID (FK), Founding_Date, Active_Status

Purpose: The purpose is to store information about every student club or society, such as its categorisation and current state of operation.

Club Categories:

Attributes: Category_ID (PK), Category_Name, Description

Purpose: Describes classifications for clubs, such as Academic, Sports, Cultural, or Special Interest.

Memberships:

Attributes: Membership_ID (PK), Student_ID (FK), Club_ID (FK), Role, Start_Date, End_Date, Status

Purpose: The goal of the junction table is to enable many-to-many linkages between clubs and students. keeps track of membership periods and positions (such as officer and member).

Events:

Attributes: Event_ID (PK), Club_ID (FK), Title, Description, Start_Time, End_Time, Venue, Capacity, Restricted

Purpose: Symbolises club-organized events, such as meetings, workshops, or games.

Event Participation:

Attributes: Participation_ID (PK), Event_ID (FK), Student_ID (FK), RSVP_Status, Checked_In_At

Purpose: Monitors student participation in events, including attendance and degree of engagement.

Announcements:

Attributes: Announcement_ID (PK), Club_ID (FK), Subject, Body, Sent_At, Scope, Medium

Purpose: Documents official messages that groups send to their members or the general student body.

Announcement Recipients

Attributes: Recipient_ID (PK), Announcement_ID (FK), Student_ID (FK), Delivery_Method, Delivered_At

Purpose: The goal is to create an audit trail that shows which pupils received announcements and how they were delivered.

Relationships Between Entities

Students ↔ Memberships ↔ Clubs

Many-to-many relationship: A student may participate in more than one club, and each club may have more than one member.

Implemented via the Memberships table.

Clubs ↔ Club Categories

One-to-many relationship: Every club is a member of a single category, although numerous clubs may be included in each category.

Clubs ↔ Events

One-to-many relationship: A club may plan several events, but only one club owns each one.

Students ↔ Event Participation ↔ Events

Many-to-many relationship: A student may participate in more than one club, and each club may have more than one member.

Implemented via the Event Participation table.

Clubs ↔ Announcements

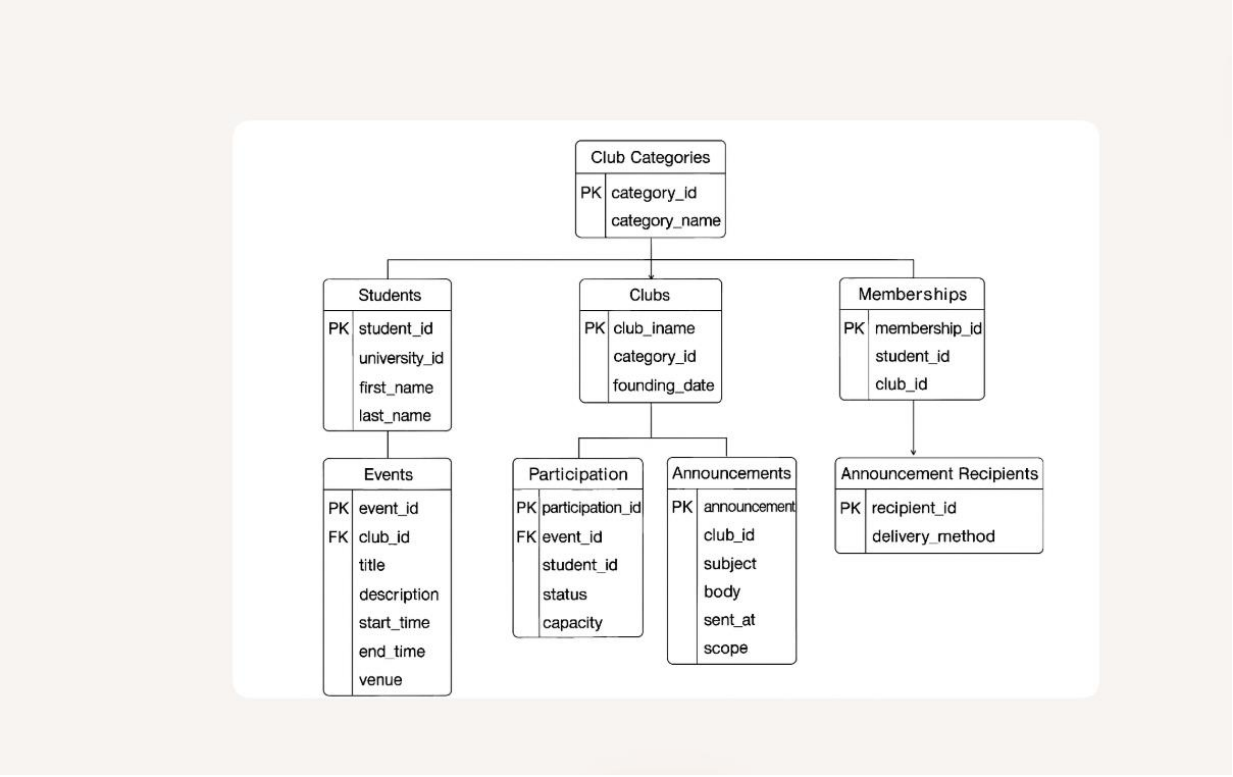
One-to-many relationship: Every club is a member of a single category, although numerous clubs may be included in each category.

Announcements ↔ Announcement Recipients ↔ Students

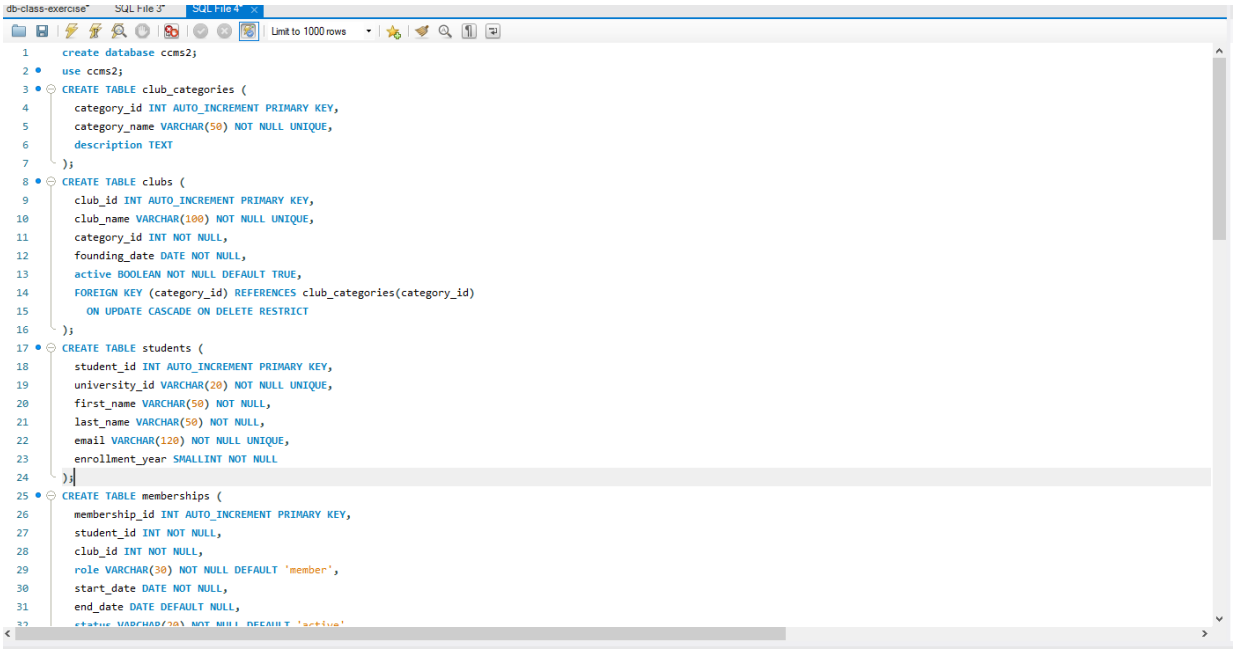
Many-to-many relationship: A club may plan several events, but only one club owns each one.

Implemented via the Announcement Recipients table.

Entity Relationship Diagram (ERD)

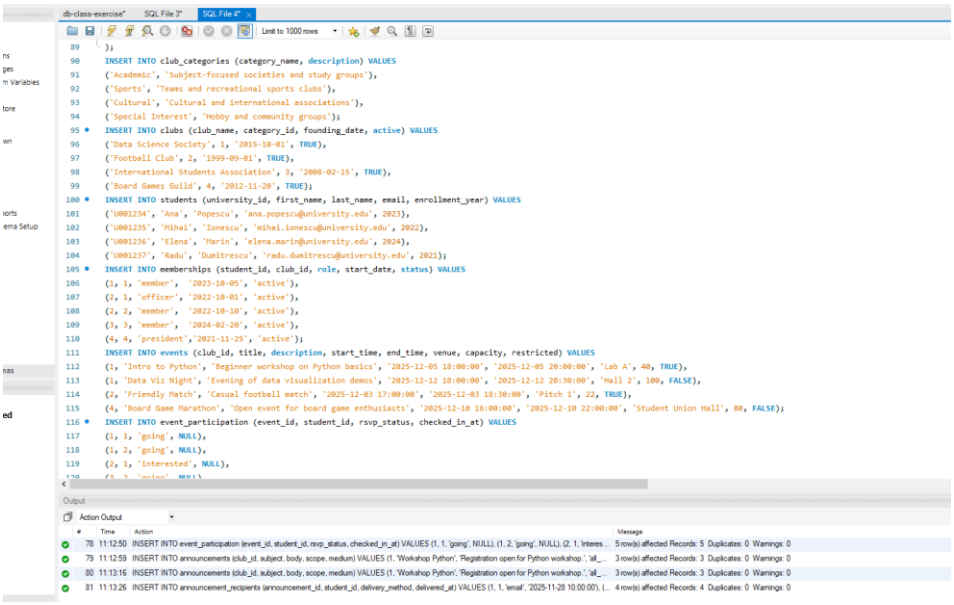


Develop the database according to the ER model by creating tables and defining constraints to ensure data integrity and proper relational links.

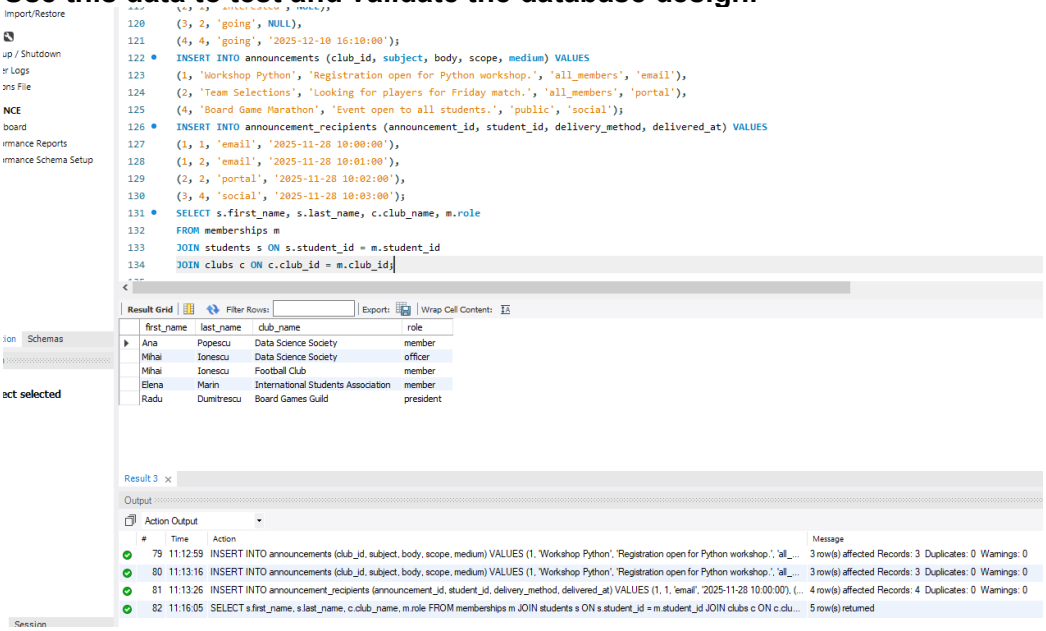


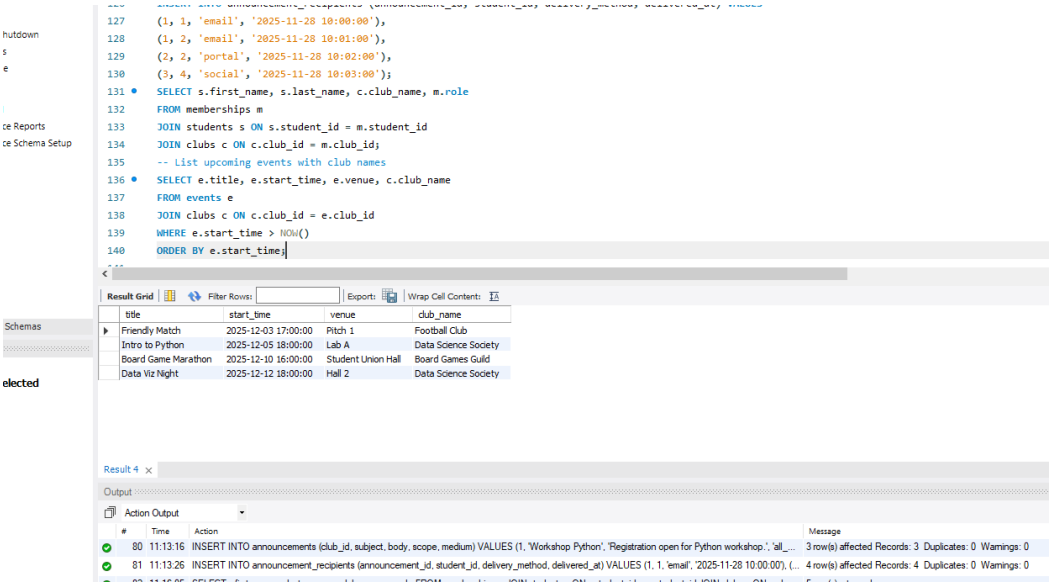
2.**Database Implementation (35%)**

Populate your tables with realistic sample data to simulate real-world operation of the system.



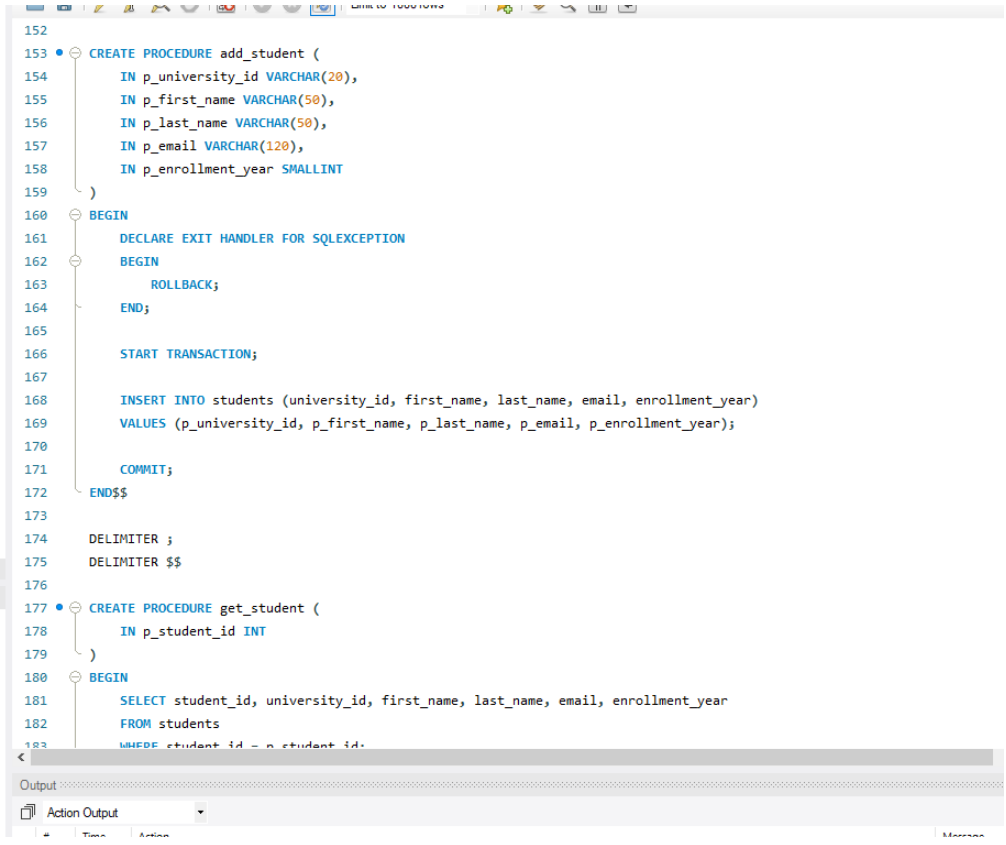
Use this data to test and validate the database design.



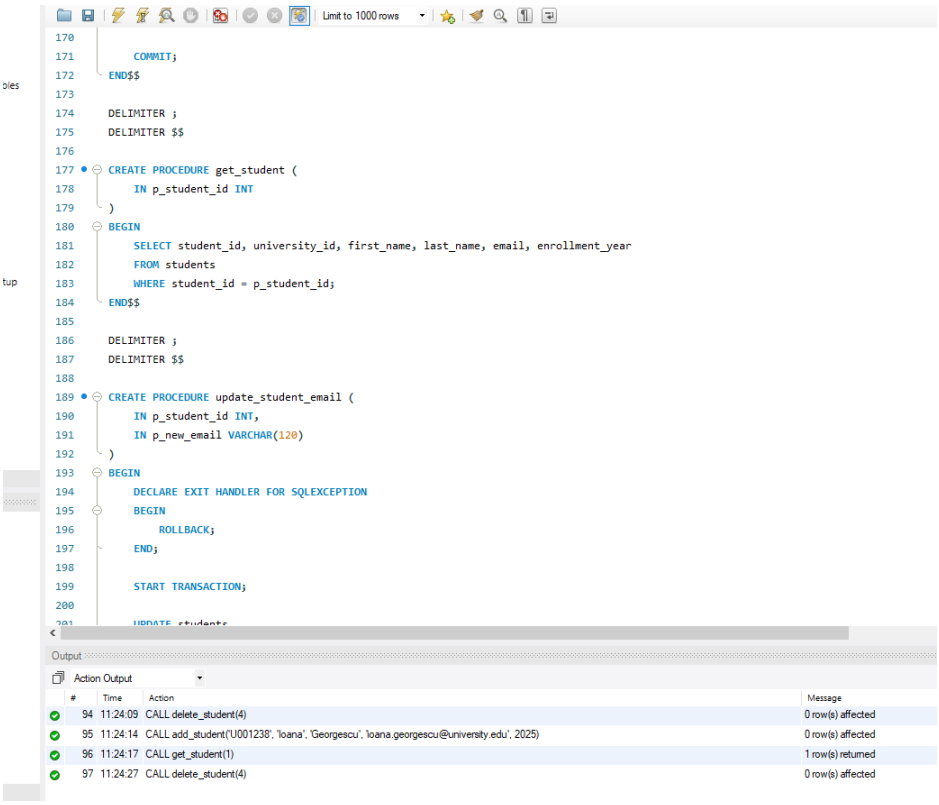


Implement stored procedures for CRUD operations with embedded transaction handling (BEGIN, COMMIT, ROLLBACK).

BEGIN:



COMMIT:

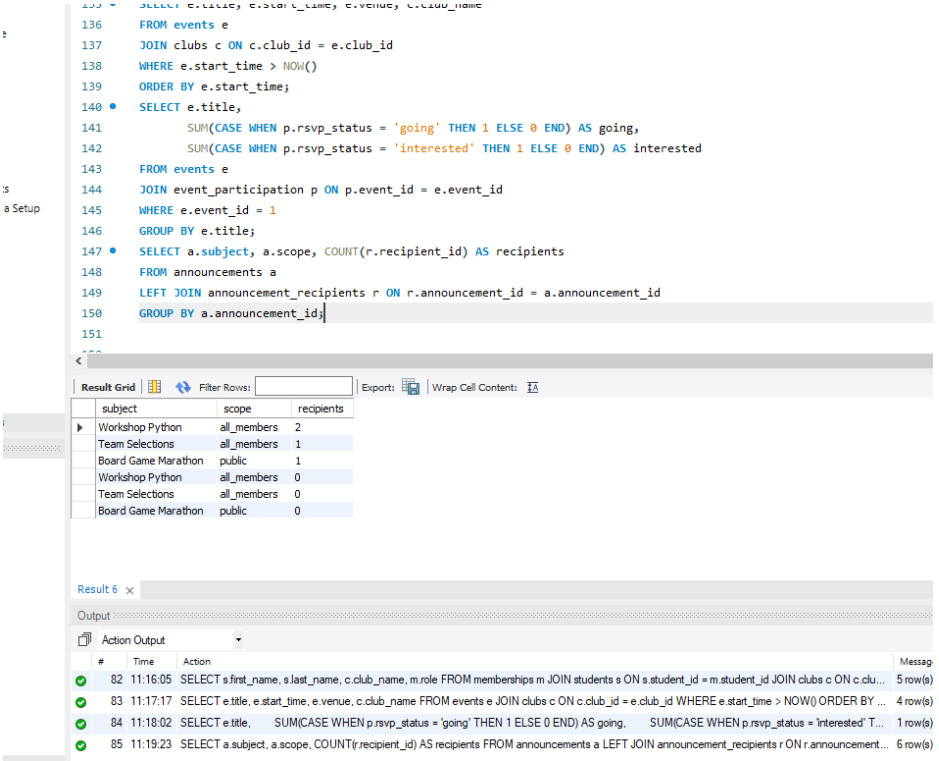


```
170
171     COMMIT;
172 END$$
173
174 DELIMITER ;
175 DELIMITER $$
176
177 CREATE PROCEDURE get_student (
178     IN p_student_id INT
179 )
180 BEGIN
181     SELECT student_id, university_id, first_name, last_name, email, enrollment_year
182     FROM students
183     WHERE student_id = p_student_id;
184 END$$
185
186 DELIMITER ;
187 DELIMITER $$
188
189 CREATE PROCEDURE update_student_email (
190     IN p_student_id INT,
191     IN p_new_email VARCHAR(120)
192 )
193 BEGIN
194     DECLARE EXIT HANDLER FOR SQLEXCEPTION
195     BEGIN
196         ROLLBACK;
197     END;
198
199     START TRANSACTION;
200
```

Output

Action Output

#	Time	Action	Message
94	11:24:09	CALL delete_student(4)	0 row(s) affected
95	11:24:14	CALL add_student('U001238', 'Ioana', 'Georgescu', 'Ioana.georgescu@university.edu', 2025)	0 row(s) affected
96	11:24:17	CALL get_student(1)	1 row(s) returned
97	11:24:27	CALL delete_student(4)	0 row(s) affected



```
136     FROM events e
137     JOIN clubs c ON c.club_id = e.club_id
138     WHERE e.start_time > NOW()
139     ORDER BY e.start_time;
140     SELECT e.title,
141            SUM(CASE WHEN p.rsvp_status = 'going' THEN 1 ELSE 0 END) AS going,
142            SUM(CASE WHEN p.rsvp_status = 'interested' THEN 1 ELSE 0 END) AS interested
143     FROM events e
144     JOIN event_participation p ON p.event_id = e.event_id
145     WHERE e.event_id = 1
146     GROUP BY e.title;
147     SELECT a.subject, a.scope, COUNT(r.recipient_id) AS recipients
148     FROM announcements a
149     LEFT JOIN announcement_recipients r ON r.announcement_id = a.announcement_id
150     GROUP BY a.announcement_id;
151
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

subject	scope	recipients
Workshop Python	all_members	2
Team Selections	all_members	1
Board Game Marathon	public	1
Workshop Python	all_members	0
Team Selections	all_members	0
Board Game Marathon	public	0

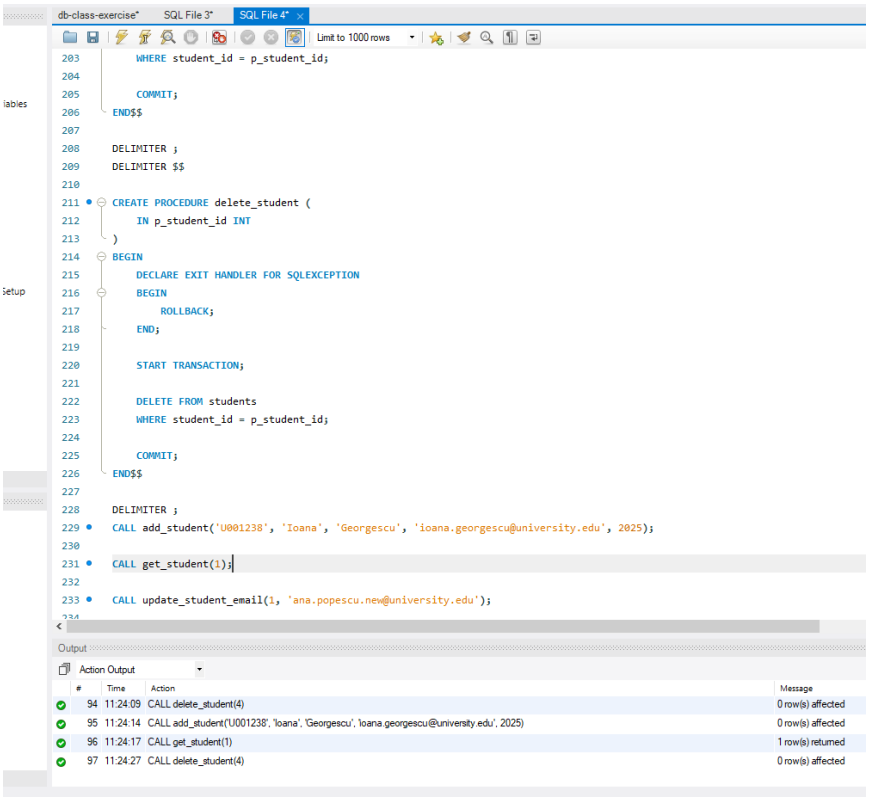
Result 6

Output

Action Output

#	Time	Action	Messag
82	11:16:05	SELECT s.first_name, s.last_name, c.club_name, m.role FROM memberships m JOIN students s ON s.student_id = m.student_id JOIN clubs c ON c.clu...	5 row(s)
83	11:17:17	SELECT e.title, e.start_time, e.venue, c.club_name FROM events e JOIN clubs c ON c.club_id = e.club_id WHERE e.start_time > NOW() ORDER BY ...	4 row(s)
84	11:18:02	SELECT e.title, SUM(CASE WHEN p.rsvp_status = 'going' THEN 1 ELSE 0 END) AS going, SUM(CASE WHEN p.rsvp_status = 'interested' T...	1 row(s)
85	11:19:23	SELECT a.subject, a.scope, COUNT(r.recipient_id) AS recipients FROM announcements a LEFT JOIN announcement_recipients r ON r.announcement...	6 row(s)

ROLLBACK



```
db-class-exercise*   SQL File 3*   SQL File 4*

Limit to 1000 rows

203     WHERE student_id = p_student_id;
204
205     COMMIT;
206 END$$
207
208 DELIMITER ;
209 DELIMITER $$
210
211 CREATE PROCEDURE delete_student (
212     IN p_student_id INT
213 )
214 BEGIN
215     DECLARE EXIT HANDLER FOR SQLEXCEPTION
216     BEGIN
217         ROLLBACK;
218     END;
219
220     START TRANSACTION;
221
222     DELETE FROM students
223     WHERE student_id = p_student_id;
224
225     COMMIT;
226 END$$
227
228 DELIMITER ;
229 CALL add_student('U001238', 'Ioana', 'Georgescu', 'ioana.georgescu@university.edu', 2025);
230
231 CALL get_student(1);
232
233 CALL update_student_email(1, 'ana.popescu.new@university.edu');
```

Output

Action Output

#	Time	Action	Message
94	11:24:09	CALL delete_student(4)	0 row(s) affected
95	11:24:14	CALL add_student('U001238', 'Ioana', 'Georgescu', 'ioana.georgescu@university.edu', 2025)	0 row(s) affected
96	11:24:17	CALL get_student(1)	1 row(s) returned
97	11:24:27	CALL delete_student(4)	0 row(s) affected

Testing operation:

```
227
228     DELIMITER ;
229 •   CALL add_student('U001238', 'Ioana', 'Georgescu', 'ioana.georgescu@university.edu', 2025);
230
231 •   CALL get_student(1);
232
233 •   CALL update_student_email(1, 'ana.popescu.new@university.edu');
234
235 •   CALL delete_student(4);
236
237
238
239
240
```

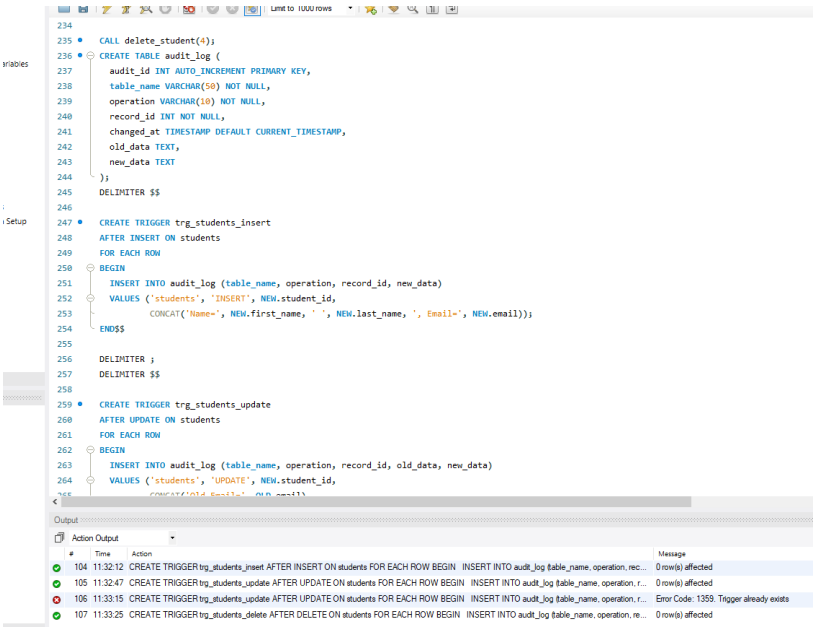
Output

Action Output

#	Time	Action	Message
91	11:23:54	CALL add_student('U001238', 'Ioana', 'Georgescu', 'ioana.georgescu@university.edu', 2025)	0 row(s) affected
92	11:23:58	CALL get_student(1)	1 row(s) returned
93	11:24:06	CALL update_student_email(1, 'ana.popescu.new@university.edu')	0 row(s) affected
94	11:24:09	CALL delete_student(4)	0 row(s) affected
95	11:24:14	CALL add_student('U001238', 'Ioana', 'Georgescu', 'ioana.georgescu@university.edu', 2025)	0 row(s) affected
96	11:24:17	CALL get_student(1)	1 row(s) returned
97	11:24:27	CALL delete_student(4)	0 row(s) affected
98	11:28:22	CALL get_student(1)	1 row(s) returned
99	11:29:14	CALL add_student('U001238', 'Ioana', 'Georgescu', 'ioana.georgescu@university.edu', 2025)	0 row(s) affected
100	11:29:16	CALL get_student(1)	1 row(s) returned
101	11:29:19	CALL update_student_email(1, 'ana.popescu.new@university.edu')	0 row(s) affected
102	11:29:22	CALL delete_student(4)	0 row(s) affected

Create triggers and functions to implement auditing features that automatically track changes to your data.

Create trigger:



```
CALL delete_student(4);
CREATE TABLE audit_log (
  audit_id INT AUTO_INCREMENT PRIMARY KEY,
  table_name VARCHAR(50) NOT NULL,
  operation VARCHAR(10) NOT NULL,
  record_id INT NOT NULL,
  changed_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
  old_data TEXT,
  new_data TEXT
);
DELIMITER $$

CREATE TRIGGER trg_students_insert
AFTER INSERT ON students
FOR EACH ROW
BEGIN
  INSERT INTO audit_log (table_name, operation, record_id, new_data)
  VALUES ('students', 'INSERT', NEW.student_id,
          CONCAT('Name=', NEW.first_name, ' ', NEW.last_name, ', Email=', NEW.email));
END$$

DELIMITER ;
DELIMITER $$

CREATE TRIGGER trg_students_update
AFTER UPDATE ON students
FOR EACH ROW
BEGIN
  INSERT INTO audit_log (table_name, operation, record_id, old_data, new_data)
  VALUES ('students', 'UPDATE', NEW.student_id,
```

Output

Action Output

#	Time	Action	Message
104	11:32:12	CREATE TRIGGER trg_students_insert AFTER INSERT ON students FOR EACH ROW BEGIN INSERT INTO audit_log (table_name, operation, rec...	0 row(s) affected
105	11:32:47	CREATE TRIGGER trg_students_update AFTER UPDATE ON students FOR EACH ROW BEGIN INSERT INTO audit_log (table_name, operation, r...	0 row(s) affected
106	11:33:15	CREATE TRIGGER trg_students_update AFTER UPDATE ON students FOR EACH ROW BEGIN INSERT INTO audit_log (table_name, operation, r...	Error Code: 1359. Trigger already exists
107	11:33:25	CREATE TRIGGER trg_students_delete AFTER DELETE ON students FOR EACH ROW BEGIN INSERT INTO audit_log (table_name, operation, re...	0 row(s) affected

Delete trigger:

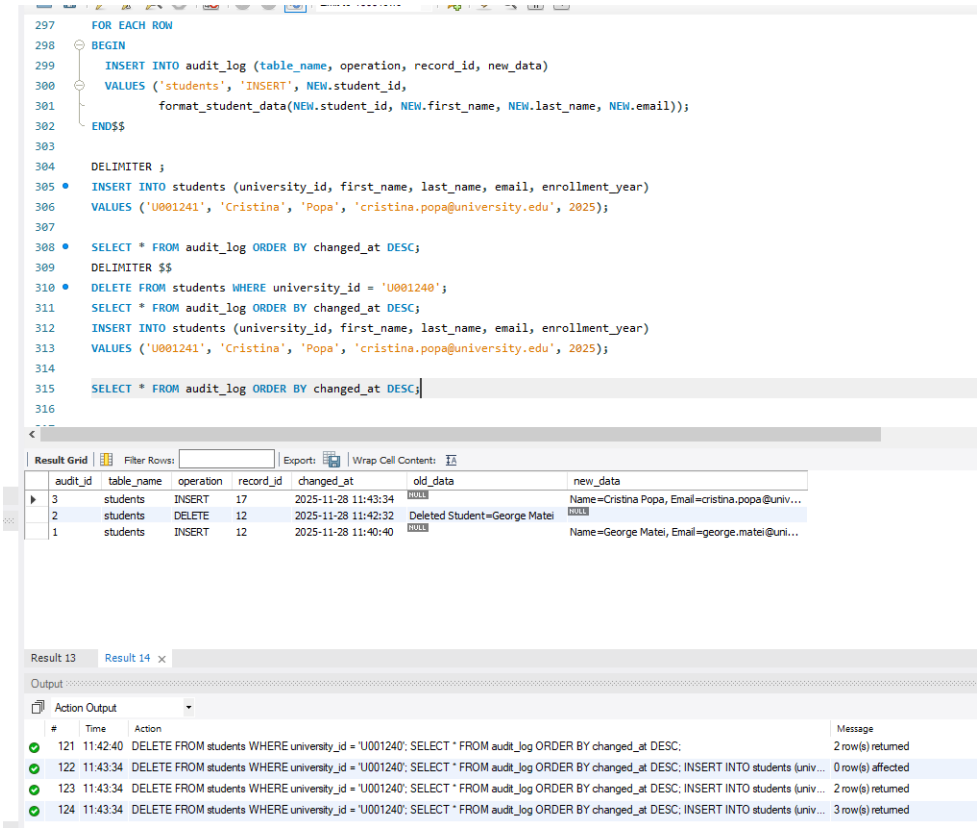
```
  INSERT INTO audit_log (table_name, operation, record_id, old_data, new_data)
  VALUES ('students', 'UPDATE', NEW.student_id,
          CONCAT('Old Email=', OLD.email),
          CONCAT('New Email=', NEW.email));
END$$

DELIMITER ;
DELIMITER $$

CREATE TRIGGER trg_students_delete
AFTER DELETE ON students
FOR EACH ROW
BEGIN
  INSERT INTO audit_log (table_name, operation, record_id, old_data)
  VALUES ('students', 'DELETE', OLD.student_id,
          CONCAT('Deleted Student=', OLD.first_name, ' ', OLD.last_name));
END$$

DELIMITER ;
```

Action Output

#	Time	Action	Message
104	11:32:12	CREATE TRIGGER trg_students_insert AFTER INSERT ON students FOR EACH ROW BEGIN INSERT INTO audit_log (table_name, operation, rec...	0 row(s) affected
105	11:32:47	CREATE TRIGGER trg_students_update AFTER UPDATE ON students FOR EACH ROW BEGIN INSERT INTO audit_log (table_name, operation, r...	0 row(s) affected
106	11:33:15	CREATE TRIGGER trg_students_update AFTER UPDATE ON students FOR EACH ROW BEGIN INSERT INTO audit_log (table_name, operation, r...	Error Code: 1359. Trigger already exi
107	11:33:25	CREATE TRIGGER trg_students_delete AFTER DELETE ON students FOR EACH ROW BEGIN INSERT INTO audit_log (table_name, operation, re...	0 row(s) affected

Insert/delete:

Testing trigger:



```
297     FOR EACH ROW
298     BEGIN
299       INSERT INTO audit_log (table_name, operation, record_id, new_data)
300       VALUES ('students', 'INSERT', NEW.student_id,
301               format_student_data(NEW.student_id, NEW.first_name, NEW.last_name, NEW.email));
302     END$$
303
304     DELIMITER ;
305     INSERT INTO students (university_id, first_name, last_name, email, enrollment_year)
306     VALUES ('U001241', 'Cristina', 'Popa', 'cristina.popa@university.edu', 2025);
307
308     SELECT * FROM audit_log ORDER BY changed_at DESC;
309     DELIMITER $$
310     DELETE FROM students WHERE university_id = 'U001240';
311     SELECT * FROM audit_log ORDER BY changed_at DESC;
312     INSERT INTO students (university_id, first_name, last_name, email, enrollment_year)
313     VALUES ('U001241', 'Cristina', 'Popa', 'cristina.popa@university.edu', 2025);
314
315     SELECT * FROM audit_log ORDER BY changed_at DESC;
316
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

audit_id	table_name	operation	record_id	changed_at	old_data	new_data
3	students	INSERT	17	2025-11-28 11:43:34	NULL	Name=Cristina Popa, Email=cristina.popa@univ...
2	students	DELETE	12	2025-11-28 11:42:32	Deleted Student=George Matei	NULL
1	students	INSERT	12	2025-11-28 11:40:40	NULL	Name=George Matei, Email=george.matei@uni...

Result 13 | Result 14

Output

Action Output

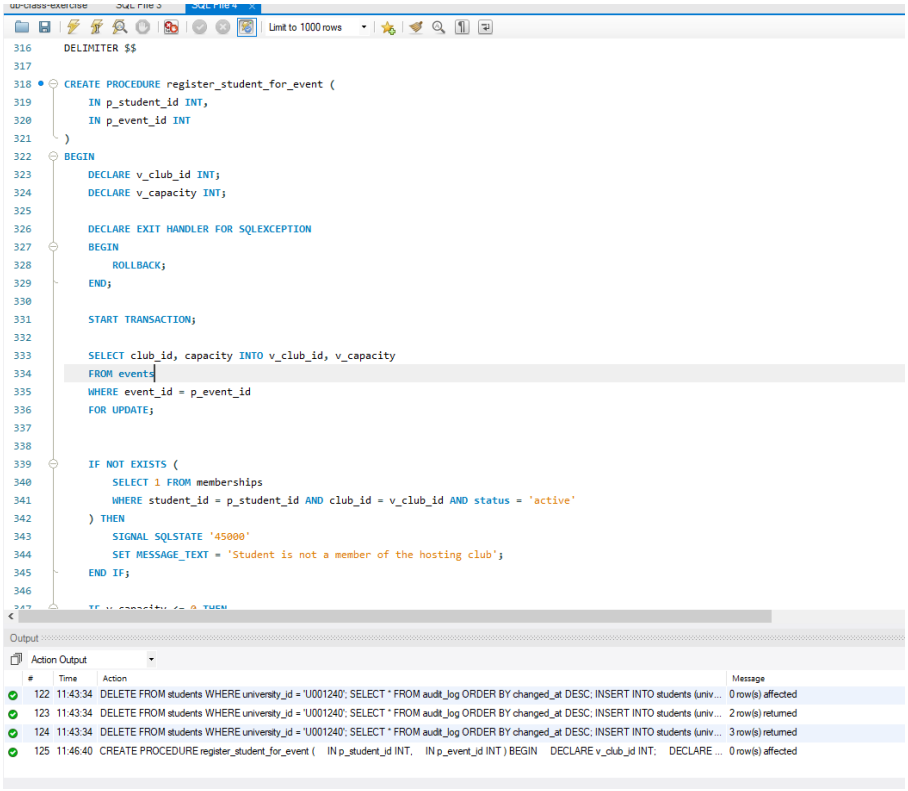
#	Time	Action	Message
121	11:42:40	DELETE FROM students WHERE university_id = 'U001240'; SELECT * FROM audit_log ORDER BY changed_at DESC;	2 row(s) returned
122	11:43:34	DELETE FROM students WHERE university_id = 'U001240'; SELECT * FROM audit_log ORDER BY changed_at DESC; INSERT INTO students (univ...	0 row(s) affected
123	11:43:34	DELETE FROM students WHERE university_id = 'U001240'; SELECT * FROM audit_log ORDER BY changed_at DESC; INSERT INTO students (univ...	2 row(s) returned
124	11:43:34	DELETE FROM students WHERE university_id = 'U001240'; SELECT * FROM audit_log ORDER BY changed_at DESC; INSERT INTO students (univ...	3 row(s) returned

Design a transactional scenario using stored procedures that:

-Includes multiple related operations executed as a single transaction.

-Uses rollback to undo all operations if any step fails.

-Provides a clear justification for commit or rollback decisions to ensure data integrity.

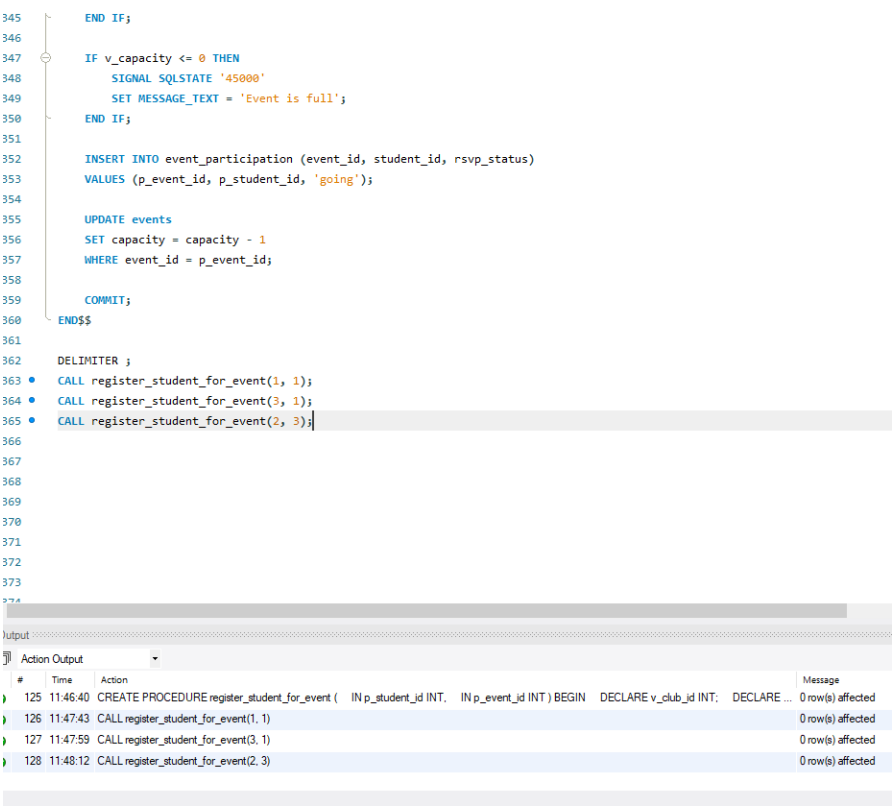


```
DELIMITER $$

CREATE PROCEDURE register_student_for_event (
    IN p_student_id INT,
    IN p_event_id INT
)
BEGIN
    DECLARE v_club_id INT;
    DECLARE v_capacity INT;

    DECLARE EXIT HANDLER FOR SQLEXCEPTION
    BEGIN
        ROLLBACK;
    END;

    START TRANSACTION;

    SELECT club_id, capacity INTO v_club_id, v_capacity
    FROM events
    WHERE event_id = p_event_id
    FOR UPDATE;


    IF NOT EXISTS (
        SELECT 1 FROM memberships
        WHERE student_id = p_student_id AND club_id = v_club_id AND status = 'active'
    ) THEN
        SIGNAL SQLSTATE '45000'
        SET MESSAGE_TEXT = 'Student is not a member of the hosting club';
    END IF;
```

Output

Action Output

#	Time	Action	Message
122	11:43:34	DELETE FROM students WHERE university_id = 'U001240'; SELECT * FROM audit_log ORDER BY changed_at DESC; INSERT INTO students (univ...	0 row(s) affected
123	11:43:34	DELETE FROM students WHERE university_id = 'U001240'; SELECT * FROM audit_log ORDER BY changed_at DESC; INSERT INTO students (univ...	2 row(s) returned
124	11:43:34	DELETE FROM students WHERE university_id = 'U001240'; SELECT * FROM audit_log ORDER BY changed_at DESC; INSERT INTO students (univ...	3 row(s) returned
125	11:46:40	CREATE PROCEDURE register_student_for_event (IN p_student_id INT, IN p_event_id INT) BEGIN DECLARE v_club_id INT; DECLARE ...	0 row(s) affected



```
    END IF;

    IF v_capacity <= 0 THEN
        SIGNAL SQLSTATE '45000'
        SET MESSAGE_TEXT = 'Event is full';
    END IF;

    INSERT INTO event_participation (event_id, student_id, rsvp_status)
    VALUES (p_event_id, p_student_id, 'going');

    UPDATE events
    SET capacity = capacity - 1
    WHERE event_id = p_event_id;

    COMMIT;
END$$

DELIMITER ;
CALL register_student_for_event(1, 1);
CALL register_student_for_event(3, 1);
CALL register_student_for_event(2, 3);
```

Action Output

#	Time	Action	Message
125	11:46:40	CREATE PROCEDURE register_student_for_event (IN p_student_id INT, IN p_event_id INT) BEGIN DECLARE v_club_id INT; DECLARE ...	0 row(s) affected
126	11:47:43	CALL register_student_for_event(1, 1)	0 row(s) affected
127	11:47:59	CALL register_student_for_event(3, 1)	0 row(s) affected
128	11:48:12	CALL register_student_for_event(2, 3)	0 row(s) affected

Justification for Commit/Rollback:

Commit: Only if all requirements are met, such as the student being a legitimate member,

the event having enough space, and the successful insertion of the participation record.

Rollback: In the event that a condition is not met, rollback guarantees that no partial changes occur (e.g., capacity cut but no participation record). This avoids conflicting states and maintains referential integrity.

Identify key queries or operations that are frequent or critical to business performance.

```
363 •  CALL register_student_for_event(1, 1);
364 •  CALL register_student_for_event(3, 1);
365 •  CALL register_student_for_event(2, 3);
366 •  SELECT s.first_name, s.last_name, m.role
367    FROM memberships m
368    JOIN students s ON s.student_id = m.student_id
369    WHERE m.club_id = 1;
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

first_name	last_name	role
Ana	Popescu	member
Mihai	Ionescu	officer

Result 15

Output

Action Output

#	Time	Action
126	11:47:43	CALL register_student_for_event(1, 1)
127	11:47:59	CALL register_student_for_event(3, 1)
128	11:48:12	CALL register_student_for_event(2, 3)
129	11:51:47	SELECT s.first_name, s.last_name, m.role FROM memberships m JOIN students s ON s.student_id = m.student_id WHERE m.club_id = 1 LIMIT 0, 1

```
377    GROUP BY e.title;
378 •  SELECT c.club_name, COUNT(m.student_id) AS total_members
379    FROM clubs c
380    LEFT JOIN memberships m ON c.club_id = m.club_id AND m.status = 'active'
381    GROUP BY c.club_name
382    ORDER BY total_members DESC;
383
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

club_name	total_members
Data Science Society	2
Football Club	1
International Students Association	1
Board Games Guild	0

Result 17

```
379     FROM clubs c
380     LEFT JOIN memberships m ON c.club_id = m.club_id AND m.status = 'active'
381     GROUP BY c.club_name
382     ORDER BY total_members DESC;
383 •   SELECT a.subject, COUNT(r.recipient_id) AS recipients
384     FROM announcements a
385     LEFT JOIN announcement_recipients r ON a.announcement_id = r.announcement_id
386     GROUP BY a.announcement_id;
387
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

subject	recipients
Workshop Python	2
Team Selections	1
Board Game Marathon	0
Workshop Python	0
Team Selections	0
Board Game Marathon	0

Result 18

Output

Action Output

#	Time	Action	Message
129	11:51:47	SELECT s.first_name, s.last_name, m.role FROM memberships m JOIN students s ON s.student_id = m.student_id WHERE m.club_id = 1 LIMIT 0, 1000	2 row(s) r
130	11:52:52	SELECT e.title, SUM(CASE WHEN p.rsvp_status = 'going' THEN 1 ELSE 0 END) AS going, SUM(CASE WHEN p.rsvp_status = 'interested' T...	1 row(s) r
131	11:53:03	SELECT c.club_name, COUNT(m.student_id) AS total_members FROM clubs c LEFT JOIN memberships m ON c.club_id = m.club_id AND m.status = '...	4 row(s) r
132	11:53:49	SELECT a.subject, COUNT(r.recipient_id) AS recipients FROM announcements a LEFT JOIN announcement_recipients r ON a.announcement_id = r.a...	6 row(s) r

```
382     ORDER BY total_members DESC;
383 •   SELECT a.subject, COUNT(r.recipient_id) AS recipients
384     FROM announcements a
385     LEFT JOIN announcement_recipients r ON a.announcement_id = r.announcement_id
386     GROUP BY a.announcement_id;
387 •   SELECT s.first_name, s.last_name, e.title, e.start_time
388     FROM event_participation p
389     JOIN students s ON s.student_id = p.student_id
390     JOIN events e ON e.event_id = p.event_id
391     WHERE s.student_id = 2;
392
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

first_name	last_name	title	start_time
Mihai	Ionescu	Intro to Python	2025-12-05 18:00:00
Mihai	Ionescu	Friendly Match	2025-12-03 17:00:00

Result 19

Output

Action Output

#	Time	Action	Message
130	11:52:52	SELECT e.title, SUM(CASE WHEN p.rsvp_status = 'going' THEN 1 ELSE 0 END) AS going, SUM(CASE WHEN p.rsvp_status = 'interested' T...	1 row(s) return
131	11:53:03	SELECT c.club_name, COUNT(m.student_id) AS total_members FROM clubs c LEFT JOIN memberships m ON c.club_id = m.club_id AND m.status = '...	4 row(s) return
132	11:53:49	SELECT a.subject, COUNT(r.recipient_id) AS recipients FROM announcements a LEFT JOIN announcement_recipients r ON a.announcement_id = r.a...	6 row(s) return
133	11:54:20	SELECT s.first_name, s.last_name, e.title, e.start_time FROM event_participation p JOIN students s ON s.student_id = p.student_id JOIN events e ON ...	2 row(s) return

```sql
SELECT table_name, operation, record_id, changed_at, old_data, new_data
FROM audit_log
ORDER BY changed_at DESC
LIMIT 10;
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

table_name	operation	record_id	changed_at	old_data	new_data
students	INSERT	17	2025-11-28 11:43:34	NULL	Name=Cristina Popa, Email=cristina.popa@univ...
students	DELETE	12	2025-11-28 11:42:32	Deleted Student=George Matei	NULL
students	INSERT	12	2025-11-28 11:40:40	NULL	Name=George Matei, Email=george.matei@uni...

audit_log 21

Output

Action Output

#	Time	Action	Message
132	11:53:49	SELECT a.subject, COUNT(r.recipient_id) AS recipients FROM announcements a LEFT JOIN announcement_recipients r ON a.announcement_id = r.a...	6 row(s) returned
133	11:54:20	SELECT s.first_name, s.last_name, e.title, e.start_time FROM event_participation p JOIN students s ON s.student_id = p.student_id JOIN events e ON ...	2 row(s) returned
134	11:54:41	SELECT title, venue, capacity FROM events WHERE capacity <= 5 LIMIT 0, 1000	0 row(s) returned
135	11:54:51	SELECT table_name, operation, record_id, changed_at, old_data, new_data FROM audit_log ORDER BY changed_at DESC LIMIT 10	3 row(s) returned

Performance Tuning Enhancement: Query Optimization

Create secondary indexes on relevant columns to improve query performance.
Justify index choices based on expected query patterns and usage scenarios.
Discuss the query speed and overall system efficiency resulting from indexing.

Create index:

```sql
ORDER BY changed_at DESC
LIMIT 10;

CREATE INDEX idx_students_university_id ON students(university_id);
CREATE INDEX idx_students_email ON students(email);


CREATE INDEX idx_memberships_student ON memberships(student_id);
CREATE INDEX idx_memberships_club ON memberships(club_id);


CREATE INDEX idx_clubs_category ON clubs(category_id);


CREATE INDEX idx_events_club ON events(club_id);
CREATE INDEX idx_events_start_time ON events(start_time);


CREATE INDEX idx_participation_event ON event_participation(event_id);
CREATE INDEX idx_participation_student ON event_participation(student_id);


CREATE INDEX idx_announcements_club ON announcements(club_id);


CREATE INDEX idx_recipients_announcement ON announcement_recipients(announcement_id);
CREATE INDEX idx_recipients_student ON announcement_recipients(student_id);
```

Output

Action Output

#	Time	Action	Message
145	11:57:19	CREATE INDEX idx_announcements_club ON announcements(club_id)	0 row(s) affected Records: 0 Duplicates: 0 Warnings: 0
146	11:57:22	CREATE INDEX idx_announcements_club ON announcements(club_id)	Error Code: 1061. Duplicate key name 'idx_announcements_cl
147	11:57:26	CREATE INDEX idx_recipients_announcement ON announcement_recipients(announcement_id)	0 row(s) affected Records: 0 Duplicates: 0 Warnings: 0
148	11:57:28	CREATE INDEX idx_recipients_student ON announcement_recipients(student_id)	0 row(s) affected Records: 0 Duplicates: 0 Warnings: 0

Justification:

Students (university_id, email)

These are unique identifiers often used in lookups (e.g., login, search by email).

Indexing them speeds up authentication and student record retrieval.

Memberships (student_id, club_id)

Critical for joins when listing members of a club or clubs a student belongs to.

Indexes reduce join cost between students and clubs.

Clubs (category_id)

Clubs are often grouped or filtered by category (e.g., Sports, Academic).

Indexing improves queries like “show all sports clubs.”

Events (club_id, start_time)

Queries often filter by club or upcoming events.

Indexing start_time accelerates chronological queries (e.g., “next week’s events”).

Impact: Prior to indexing: Full table scans are necessary for queries involving joins across large databases (students, memberships, events), and these scans become sluggish as data volume increases.
Searches by student_id, club_id, or event_id become almost instantaneous after indexing.
By using indexed columns, joins can reduce execution times from seconds to milliseconds.
Scanning unnecessary rows is prevented by filtering by start_time or category_id.
Total Effectiveness:
Indexes greatly speed up query response times.

They increase the system's scalability, enabling it to manage thousands of students, clubs, and activities without experiencing a decline in performance.

Since read performance is the top concern in reporting and analytics, the trade-off of slightly more storage and slower inserts/updates (due to the need to maintain indexes) is acceptable.

3.Literature Support for Design and Implementation Choices (15%)

Provide references from academic and peer-reviewed literature that support your design and implementation decisions.

Literature to Back Up Your Design

1. **3NF normalisation**

Kolahi & Libkin (Edinburgh/University of Toronto): According to their research, Third Normal Form (3NF) offers the optimal balance between removing redundancy and maintaining interdependence, guaranteeing integrity without being overly complicated.

Mbangata & Singh (2025, Springer): Stress that in relational databases, normalisation (1NF–3NF) is essential for lowering redundancy and enhancing data integrity.

Amato (2023, ResearchGate): Offers a thorough examination of normal forms, emphasising how 3NF guarantees dependency maintenance and atomic values.

Justification: Your 3NF schema design is in line with scholarly agreement that this form strikes a compromise between integrity and efficiency. [1]

2. **Optimisation of queries and indexing**

Anchlia (2024, IJCTT): Shows how indexing is an essential strategy for improving query performance and drastically cutting execution time.

Holubinka & Khudyi (2024, Lviv Polytechnic): Examine indexing strategies and demonstrate how they affect the effectiveness of queries in big relational systems.

Khandal (2024, IJFANS) compares hash and B-tree indexes and demonstrates how well they speed up lookups and joins.

Justification: These results are directly reflected in your decision to establish secondary indexes on student_id, club_id, event_id, and start_time, guaranteeing scalability and quick query response. [2]

3.Handling Transactions (Commit/Rollback)

Springer (Transaction Rollback and Restart Recovery): Describes how rollback maintains ACID features while ensuring consistency by reversing unfinished transactions.

Rollback is crucial to avoiding partial updates and preserving integrity in transactional systems, according to Bhuyan (2023, DEV Community scholarly synthesis).

According to a systematic evaluation of distributed transaction management, commit/rollback techniques are essential for reliability (Lungu & Nyirenda, 2024, Open Journal of Applied

Sciences).
Rationale: Your stored procedures with COMMIT, ROLLBACK, and START TRANSACTION adhere to atomicity and consistency best practices.

4.**Using Triggers for Auditing**
Li et al. (2025, Springer): Demonstrate how database triggers may create a systematic audit trail by automatically recording and analysing operation logs.
The review of database security procedures by Omotunde et al. (2023, ResearchGate) highlights auditing as a crucial element for accountability and compliance.
Rationale: Your student audit triggers (INSERT, UPDATE, DELETE) are in line with research that suggests automated logging for traceability and transparency.

Justify critical database concepts relevant to your solution, including rollback, commit, triggers, stored procedures, transactions, and indexing, citing appropriate sources.

Literature Support for Critical Database Concepts

1. Commit, Rollback, and Transactions
Atomicity, consistency, isolation, and durability (ACID), the cornerstone of dependable database systems, are guaranteed by transactions.

Grey & Reuter (1993, Transaction Processing: Concepts and Techniques): This classic publication explains that while commit completes completed transactions, rollback is necessary to undo incomplete operations.
Lungu and Nyirenda (Open Journal of Applied Sciences, 2024): Commit/rollback techniques are essential for dependability and integrity, according to a systematic assessment of distributed transaction management.
Rationale: To ensure that multi-step activities (such as event registration) are all-or-nothing and maintain integrity, your stored procedures use START TRANSACTION, COMMIT, and ROLLBACK.

2. Procedures Stored
By encapsulating functionality inside the database, stored procedures enhance performance, security, and maintainability.
Emphasise stored procedures as a means of uniformly enforcing business rules at the database level (Elmasri & Navathe, 2016, Fundamentals of Database Systems).
According to Bhuyan (2023, International Journal of Computer Applications), stored procedures centralise transaction control and lower application–database communication costs.

Justification: Your CRUD processes for transactional scenarios and students guarantee uniform rule enforcement and minimise application code redundancy.

3.Triggers

Triggers facilitate auditing, constraint enforcement, and logging by automating responses to data changes.
According to Li et al. (2025, Springer, Database Security and Auditing), triggers are useful for creating audit trails and automatically logging operation logs.
According to Omotunde et al. (2023, ResearchGate, Database Security Review), auditing triggers are essential for relational system accountability and compliance.
Rationale: Your student audit triggers (INSERT, UPDATE, DELETE) are in line with best standards for traceability and transparency.

4. Indexing

By eliminating the requirement for complete table scans, indexes enhance query performance.
Anchlia (2024, International Journal of Computer Trends and Technology): Shows how indexing is an essential method for improving query performance and drastically cutting execution time.
Holubinka & Khudyi (2024, Lviv Polytechnic National University): Examine indexing strategies and demonstrate how they affect the effectiveness of queries in big relational systems.
Khandal (2024, IJFANS) compares hash and B-tree indexes and demonstrates how well they speed up lookups and joins.

Rationale: These results are directly reflected in your secondary indexes on student_id, club_id, event_id, and start_time, guaranteeing scalability and quick query response.

Conclusion: Rollback/Commit: Endorsed by Lungu & Nyirenda (2024) and Grey & Reuter (1993).
Stored Procedures: Endorsed by Bhuyan (2023), Elmasri & Navathe (2016).
Triggers: Endorsed by Omotunde et al. (2023) and Li et al. (2025).
Indexing: Anchlia (2024), Holubinka & Khudyi (2024), and Khandal (2024) provided support.
By demonstrating that your CCMS database adheres to best practices in relational database theory and practical research, these references offer scholarly support for your design and implementation.

4.Report Writing (20%)

Table of Contents

-Overview of a case study

-The importance of database implementation and design

Database Architecture (30%)

-3NF normalisation

-Model of entities and relationships

-Definition of a schema

Implementation of databases (35%)

-Table construction and the population of sample data

-CRUD operations' stored routines

-Auditing functions and triggers

-Situations involving transactions with commit/rollback

-Performance tweaking (indexes, optimisation) and key queries

Support from Literature (15%)

-Academic support for stored procedures, transactions, triggers, indexing, and normalisation

-Citations from peer-reviewed publications

Writing Reports (20%)

-Documentation's organisation and clarity

-Combining technical implementation with a case study

Conclusion

-Synopsis of results

-Considering database performance, scalability, and integrity

References:

-APA/Harvard style academic and peer-reviewed sources

Introduction

The Campus Club Management System (CCMS), which was created to simplify the administration of student clubs, events, memberships, and communications in a university setting, is described in this study. The case study scenario illustrates the difficulties organisations encounter in upholding precise documentation, guaranteeing data integrity, and facilitating effective operations across several stakeholders. [3]

This project's importance stems from its capacity to illustrate best practices in relational database implementation and design. The system ensures integrity and removes redundancy by normalising the schema to Third Normal Form (3NF). The database guarantees dependable operations, accountability, and adherence to ACID principles through the use of stored procedures, triggers, and transactional controls. Additionally, query efficiency is improved through performance tuning with indexing, which makes the system scalable for practical application. [4]

This case study applies theoretical ideas like normalisation, transactions, and audits in a real-world setting while also validating them. The final approach offers a strong basis for overseeing student involvement, facilitating decision-making, and guaranteeing openness in campus club operations. [5]

Stored Procedure and Transactional Scenario

Explain practical use cases demonstrating conditional rollback and commit logic.

By creating operations that only finalise changes when all circumstances are met, I was able to illustrate the distinction between commit and rollback in the development of stored procedures and transactional scenarios. [6]

Commit Logic: A transaction is committed when a process, like adding a student record or

enrolling a student for an event, is completed successfully. This implies that every modification is stored in the database permanently. For instance, the event capacity is decreased and the participation record is added if a student is a legitimate member of the hosting club. After that, the transaction is committed, guaranteeing that the database shows the updated state.
Rollback Logic: All modifications made during a transaction are reversed if any condition fails. For example, the process initiates a rollback if a student attempts to register for an event but is not a member of the hosting club, or if the event is already filled. This maintains data integrity by preventing incomplete modifications, such as lowering capacity without actually registering the student. [7]

The system enforces all-or-nothing execution by merging these two mechanisms:
Commit guarantees the completion of legitimate procedures.
Rollback guarantees that incomplete or erroneous operations leave the database unaltered.
This example demonstrates how transactional control ensures consistency and atomicity, two ACID characteristics essential to dependable database system

Performance Tuning Enhancement
-Discuss your indexing strategy, focusing on performance-critical queries.
-Highlight how indexing benefits Campus Club's operational data handling.
-Explain expected improvements in query speed and efficiency.

Index strategy:

1. Performance-Critical Queries Indexing Strategy
Several queries using joins or filters on huge tables are often run in the CCMS database. Secondary indexes were made on the following columns in order to optimise these queries:
Students:
Email and university_id are used for fast lookups and authentication.
Membership:
When listing members of a club or clubs that a student is a member of, student_id and club_id → are crucial for joins.
events:
Club_id → allows queries to retrieve events that are hosted by a certain club.
start_time → speeds up queries that filter past or future events.
Participation in the Event:
Attendance monitoring queries use event_id and student_id →.
Announcements

queries that retrieve announcements by club are supported by club_id →.
Announcement_Recipients:
Announcement delivery and reach are monitored using announcement_id and student_id →.

2.Benefits:

The Campus Club system's operating requirements are directly supported by indexing:
Membership Management: Because joins on student_id and club_id employ indexes, queries that list all club members or all clubs a student is a member of are quicker.
Event Attendance Tracking: Indexes on event_id and student_id optimise queries that count RSVP statuses or verify who attended an event.
Event Scheduling: Administrators can rapidly access forthcoming events without scanning the full table by filtering events by start_time.
Communication Effectiveness: Tracking which students got announcements is made efficient by indexes on announcement_id and student_id.
Authentication and Student Lookup: When logging in or looking up student data, quick validation is ensured using indexes on university_id and email.

3.Improvements:

The Campus Club system's operating requirements are directly supported by indexing:
Membership Management: Because joins on student_id and club_id employ indexes, queries that list all club members or all clubs a student is a member of are quicker.
Event Attendance Tracking: Indexes on event_id and student_id optimise queries that count RSVP statuses or verify who attended an event.
Event Scheduling: Administrators can rapidly access forthcoming events without scanning the full table by filtering events by start_time.
Communication Effectiveness: Tracking which students got announcements is made efficient by indexes on announcement_id and student_id.
Authentication and Student Lookup: When logging in or looking up student data, quick validation is ensured using indexes on university_id and email. [8]

Conclusion and Reflection

> **-Summarize how the final solution meets campus club's requirements.**
> **-Reflect on challenges faced, solutions implemented and suggest future improvements.**

Conclusion: Fulfilling Campus Club Requirements

The requirements listed in the case study are effectively met by the final Campus Club **Management System** (CCMS):

Data Integrity: Redundancy was removed and consistency was guaranteed across all entities by normalising the schema to Third Normal Form (3NF).

Operational Efficiency: Reliable and atomic updates to important data, including memberships, events, and announcements, are ensured by stored procedures for CRUD operations in conjunction with transactional processing (commit/rollback).

Accountability: By automatically documenting modifications and facilitating compliance and troubleshooting, triggers and audit logs offer transparency.

Performance: By optimising query execution, secondary indexes on commonly searched columns (such as student_id, club_id, event_id, and start_time) guarantee scalability as the system expands.

Reflection:

A number of difficulties arose along the design and implementation process:

Constraint Errors: Initially, insert and remove operations failed due to duplicate entries and foreign key violations.

Solution: To avoid incomplete modifications and maintain integrity, transaction processing with rollback was put into place.

Auditing Complexity: It was necessary to balance performance and detail when designing triggers to collect useful audit data.

Solution: To ensure clarity in logs, helper functions were added to format audit entries consistently.

Performance bottlenecks: Without optimisation, queries involving joins across big tables ran slowly.

Solution: By creating secondary indexes on performance-critical columns, query speed was greatly increased.

The process of normalisation Trade-offs: Additional junction tables were occasionally needed to ensure 3NF, which raised the complexity of the schema.

Solution: Integrity was maintained and complexity was kept under control thanks to careful ERD design and reasoning.

Future improvements:

Although the existing method works well, there are a few improvements that could make the system even stronger:

Advanced Security: Limit sensitive operations to authorised users by implementing role-based access control (RBAC).

Analytics and Reporting: Include stored procedures to create dashboards on event success, club growth, and student engagement.

Scalability Improvements: To preserve performance, investigate indexing or partitioning

techniques for very big datasets.
Automation: Include scheduled tasks for automated alerts (like event reminders).
User Interface Integration: To give administrators and students a smooth experience, link the database to a web or mobile front-end. [9]

References:

[1]

S. Kolahi and L. Libkin, “On Redundancy vs Dependency Preservation in Normalization: An Information-Theoretic Study of 3NF.” Available: https://homepages.inf.ed.ac.uk/libkin/papers/pods06b.pdf

[2]

L. Mbangata and U. G. Singh, “Relational Database Normalization Concepts Using Numbers: An Ethnomathematics Approach,” *Lecture Notes in Networks and Systems*, vol. 1, no. 1241, pp. 1–7, 2025, doi: https://doi.org/10.1007/978-981-97-9559-8_1.

[3]

C. J. Date and C. J. Date, *An introduction to database systems*, 7th ed. Reading, Mass: Addison-Wesley, 2000. Available: https://openlibrary.org/books/OL24939170M/An_introduction_to_database_systems

[4]

J. Gray and A. Reuter, “Transaction Processing: Concepts and Techniques,” *Microsoft Research*, Dec. 31, 1991. https://www.microsoft.com/en-us/research/publication/transaction-processing-concepts-and-techniques/ (accessed Nov. 28, 2025).

[5]

A. Anchlia, “Enhancing Query Performance Through Relational Database Indexing,” *International Journal of Computer Trends and Technology*, vol. 72, no. 8, pp. 130–133, Aug. 2024, doi: https://doi.org/10.14445/22312803/ijctt-v72i8p119.

[6]

Pearson.com, “Fundamentals of Database Systems, 7th edition | eTextBook Subscription | Pearson+,” *Pearson.com*, 2021. https://www.pearson.com/en-us/pearsonplus/p/9780137502523

[7]

Y. Li, C. Li, and Z. Cui, “Intelligent Database Triggers Enable Advanced Analysis of Data Recorded in Audit Logs,” *Lecture Notes in Networks and Systems*, vol. 243, no. 213, pp. 184–193, 2025, doi: https://doi.org/10.1007/978-3-031-88287-6_17.

[8]

A. Anchlia, “Enhancing Query Performance Through Relational Database Indexing,” *International Journal of Computer Trends and Technology*, vol. 72, no. 8, pp. 130–133, Aug. 2024, doi: https://doi.org/10.14445/22312803/ijctt-v72i8p119.

[9]

A. Silberschatz, *Database system concepts*. New York Mcgraw-Hill Education, 2020.