

XIAMEN UNIVERSITY MALAYSIA



Course Code : SOF202
Course Name : Database
Lecturer : Dr. Subashini A/P Ganapathy
Academic Session : 2025/09
Assessment Title : Lab Report
Submission Due Date : 19 December 2025

Prepared by :

Student ID	Student Name
DSC2409007	Deng Kailong (Leader)
DSC2409009	Du Zhixuan
DSC2409026	Min Yiduo
DSC2409027	Pan Ziliang
DSC2409033	Tao Ouwen

Date Received :

Feedback from Lecturer:	Mark:
-------------------------	-------

Own Work Declaration

I/We acknowledge that my/our work will be examined for plagiarism or any other form of academic misconduct, and that the digital copy of the work may be retained for future comparisons.

I/We confirm that all sources cited in this work have been correctly acknowledged and that I/we fully understand the serious consequences of any intentional or unintentional academic misconduct.

In addition, I/we affirm that this submission does not contain any materials generated by AI tools, including direct copying and pasting of text or paraphrasing. This work is my/our original creation, and it has not been based on any work of other students (past or present), nor has it been previously submitted to any other course or institution.

Signature:

邓楷诚 木智轩 阮一多 潘子良
陶政文

Date: 10 December 2025

Contents

Project Gantt Chart and Milestones	1
Task 1 Database Integrity	2
1.1 Objective and Context	2
1.2 Task 1A: Sample CREATE TABLE Statements with Integrity Constraints	2
1.2.1 Domain Integrity Constraints	2
1.2.2 Entity Integrity Constraints	3
1.2.3 Referential Integrity Constraints	4
1.3 SQL Evidence: Table Creation and Verification	5
1.4 Task 1B: How Integrity Constraints Are Applied and Maintained	5
1.4.1 Maintaining Domain Integrity (DB + Django)	5
1.4.2 Maintaining Entity Integrity	7
1.4.3 Maintaining Referential Integrity	7
1.5 Summary	8
1.6 Schema Diagram (Supplementary)	8
Task 2 SQL	9
2.1 Table Creation	9
2.2 Data Population	11
2.3 Data Retrieval and Manipulation	13
2.3.1 Filtering	13
2.3.2 Aggregate Functions	15
2.3.3 Limit / Sorting	16
2.3.4 Join Operators	18
2.3.5 String / Arithmetic Operations	19
2.3.6 Formatting	20
Task 3 Triggering	22
3.1 Implemented Triggers	22
3.1.1 Booking Guardrails	22
3.1.2 Maintenance XOR Constraints	24
3.1.3 Reservation Overlap and Availability	25
3.1.4 Reservation Equipment Availability	27
3.1.5 Session Enrollment Capacity	30
3.2 Evidence Scenarios	32
3.2.1 Trigger Interaction Diagram	32
3.2.2 Scenario A: Reservation Overlap is Blocked	33
3.2.3 Scenario B: Booking Pending Limit	34

XIAMEN UNIVERSITY MALAYSIA

3.2.4	Scenario C: Maintenance XOR Constraint	35
3.2.5	Scenario D: Equipment Availability	36
3.2.6	Scenario E: Session Enrollment Capacity	37
Task 4 Access Control	38
4.1	Objective and Alignment with the Implemented Schema	38
4.2	Mechanism 1: Role-Based Access Control (RBAC)	39
4.2.1	Role Model and Privilege Boundary	39
4.2.2	SQL Enforcement via GRANT / REVOKE	40
4.2.3	Verification Evidence (SQL)	40
4.2.4	Django Mapping and User Experience	42
4.3	Mechanism 2: Row-Level Access Control (Own-Row / Own-Session)	42
4.3.1	Rationale	42
4.3.2	SQL View for Member Booking Data	43
4.3.3	Django Row-Level Filtering and Update Boundaries	44
4.4	Summary	45
Conclusion	46
Marking Rubric		

SOF202 Database Lab Report

Group Members: Deng Kailong (Leader), Min Yiduo, Pan Ziliang, Tao Ouwen, Du Zhixuan

Milestone 1: Environment Ready (11/25/2025)

Confirmed the previous database schema is error-free and the MySQL environment is configured for all group members.

Milestone 2: Database Populated (12/05/2025)

All tables created with correct integrity constraints (PK/FK) and successfully populated with 6-10 rows of sample data.

Milestone 3: Functionality Verified (12/11/2025)

Advanced SQL queries, triggers, and access control mechanisms are implemented, tested, and screenshots are captured.

Milestone 4: Final Submission (12/18/2025)

The final lab report is compiled, formatted according to guidelines, reviewed, and submitted via Moodle.

Task 1 Database Integrity

1.1 Objective and Context

In a sports arena reservation system, data integrity is not a theoretical concern; it directly determines whether bookings are trustworthy, facilities are not double-booked, equipment inventory remains accurate, and operational decisions can be made based on reliable records. Following the requirements in `docs/description.md`, this task demonstrates how our database design enforces integrity through three complementary constraint families: **domain integrity**, **entity integrity**, and **referential integrity**.

All examples in this section are strictly aligned with the implemented schema exported in `sports_arena.sql`. We intentionally select representative tables that are central to the business workflow: `equipment`, `reservation`, `member`, and `booking`.

1.2 Task 1A: Sample `CREATE TABLE` Statements with Integrity Constraints

1.2.1 Domain Integrity Constraints

Domain integrity ensures that each attribute value stays within a valid domain (type, range, format, and business-validity rules). In our schema, domain integrity is enforced through:

- appropriate data types (e.g., `DATE`, `TIME(6)`, `INT UNSIGNED`),
- `NOT NULL` to prevent incomplete records, and
- `CHECK` constraints to express semantic rules at the database level.

The `equipment` table below demonstrates a numeric domain restriction: `Total_Quantity` is defined as `INT UNSIGNED` and further protected by a `CHECK` constraint, ensuring inventory quantities are non-negative.

```

1 -- Domain integrity example (from sports_arena.sql): non-negative equipment
2 inventory
3 CREATE TABLE `equipment` (
4   `Equipment_ID` int NOT NULL AUTO_INCREMENT,
5   `Equipment_Name` varchar(100) CHARACTER SET utf8mb4 COLLATE utf8mb4_unicode_ci NOT
6   NULL,
7   `Type` varchar(50) CHARACTER SET utf8mb4 COLLATE utf8mb4_unicode_ci NOT NULL,
8   `Status` varchar(20) CHARACTER SET utf8mb4 COLLATE utf8mb4_unicode_ci NOT NULL,
9   `Total_Quantity` int UNSIGNED NOT NULL,
10  PRIMARY KEY (`Equipment_ID`) USING BTREE,
```

```

9  CONSTRAINT `equipment_chk_1` CHECK (`Total_Quantity` >= 0)
10 ) ENGINE = InnoDB AUTO_INCREMENT = 16 CHARACTER SET = utf8mb4 COLLATE =
    utf8mb4_unicode_ci ROW_FORMAT = Dynamic;

```

The `reservation` table demonstrates temporal domain rules: (i) a reservation must have a valid time interval, and (ii) each facility cannot have duplicated reservation start times on the same date. These are captured with `CHECK` and a composite `UNIQUE` index respectively.

```

1 -- Domain integrity example (from sports_arena.sql): valid time interval and slot
   uniqueness
2 CREATE TABLE `reservation` (
3   `Reservation_ID` int NOT NULL AUTO_INCREMENT,
4   `Reservation_Date` date NOT NULL,
5   `Start_Time` time(6) NOT NULL,
6   `End_Time` time(6) NOT NULL,
7   `Facility_ID` int NOT NULL,
8   PRIMARY KEY (`Reservation_ID`) USING BTREE,
9   UNIQUE INDEX `Reservation_Facility_ID_Reservation__55861018_uniq`(`Facility_ID` ASC, `Reservation_Date` ASC, `Start_Time` ASC) USING BTREE,
10  INDEX `Reservation_Facilit_e090a5_idx`(`Facility_ID` ASC, `Reservation_Date` ASC)
     USING BTREE,
11  CONSTRAINT `Reservation_Facility_ID_3e1ddf39_fk_Facility_Facility_ID` FOREIGN KEY
     (`Facility_ID`) REFERENCES `facility` (`Facility_ID`) ON DELETE RESTRICT ON
     UPDATE RESTRICT,
12  CONSTRAINT `chk_reservation_time` CHECK (`Start_Time` < `End_Time`)
13 ) ENGINE = InnoDB AUTO_INCREMENT = 3 CHARACTER SET = utf8mb4 COLLATE =
    utf8mb4_unicode_ci ROW_FORMAT = Dynamic;

```

1.2.2 Entity Integrity Constraints

Entity integrity guarantees that each row is uniquely identifiable and non-ambiguous. In practice, this is achieved through:

- primary keys (`PRIMARY KEY`) that are unique and non-null,
- additional `UNIQUE` constraints for real-world uniqueness rules, and
- careful key design for specialization tables (where a child table key is also a foreign key).

The `member` table below illustrates entity integrity: `Member_ID` is an auto-increment primary key, and `user_id` is uniquely constrained to preserve the one-to-one relationship between a Django account (`auth_user`) and the corresponding member profile.

```

1 -- Entity integrity example (from sports_arena.sql): primary key + unique user
   mapping

```

```

2 CREATE TABLE `member` (
3   `Member_ID` int NOT NULL AUTO_INCREMENT,
4   `First_Name` varchar(50) CHARACTER SET utf8mb4 COLLATE utf8mb4_unicode_ci NOT NULL
      ,
5   `Last_Name` varchar(50) CHARACTER SET utf8mb4 COLLATE utf8mb4_unicode_ci NOT NULL,
6   `Registration_Date` date NOT NULL,
7   `Membership_Status` varchar(20) CHARACTER SET utf8mb4 COLLATE utf8mb4_unicode_ci
      NOT NULL,
8   `user_id` int NULL DEFAULT NULL,
9   PRIMARY KEY (`Member_ID`) USING BTREE,
10  UNIQUE INDEX `user_id`(`user_id` ASC) USING BTREE,
11  CONSTRAINT `Member_user_id_2cb16a29_fk_auth_user_id` FOREIGN KEY (`user_id`)
      REFERENCES `auth_user` (`id`) ON DELETE RESTRICT ON UPDATE RESTRICT
12 ) ENGINE = InnoDB AUTO_INCREMENT = 4 CHARACTER SET = utf8mb4 COLLATE =
      utf8mb4_unicode_ci ROW_FORMAT = Dynamic;

```

1.2.3 Referential Integrity Constraints

Referential integrity ensures that relationships between tables remain valid. In other words, a foreign key must reference an existing primary key, preventing orphan records and broken associations.

The **booking** table implements a specialization of **reservation**: **Reservation_ID** acts as the primary key of **booking** and simultaneously references **reservation(Reservation_ID)**. Additionally, each booking must reference a valid member via **Member_ID**.

```

1 -- Referential integrity example (from sports_arena.sql): booking must link to
2   existing reservation and member
3 CREATE TABLE `booking` (
4   `Reservation_ID` int NOT NULL,
5   `Booking_Status` varchar(20) CHARACTER SET utf8mb4 COLLATE utf8mb4_unicode_ci NOT
      NULL,
6   `Member_ID` int NOT NULL,
7   PRIMARY KEY (`Reservation_ID`) USING BTREE,
8   INDEX `Booking_Member__138c47_idx`(`Member_ID` ASC) USING BTREE,
9   CONSTRAINT `Booking_Member_ID_311d123e_fk_Member_Member_ID`
      FOREIGN KEY (`Member_ID`) REFERENCES `member` (`Member_ID`)
      ON DELETE RESTRICT ON UPDATE RESTRICT,
10  CONSTRAINT `Booking_Reservation_ID_2e41085d_fk_Reservation_Reservation_ID`
      FOREIGN KEY (`Reservation_ID`) REFERENCES `reservation` (`Reservation_ID`)
      ON DELETE RESTRICT ON UPDATE RESTRICT
11 ) ENGINE = InnoDB CHARACTER SET = utf8mb4 COLLATE = utf8mb4_unicode_ci ROW_FORMAT =
      Dynamic;

```

1.3 SQL Evidence: Table Creation and Verification

After deploying the schema, we verified both the execution logic and the stored definitions. Figure 1.1 shows the successful execution of the `CREATE TABLE` statements for key tables, confirming that the DBMS accepted the integrity constraints (Query OK).

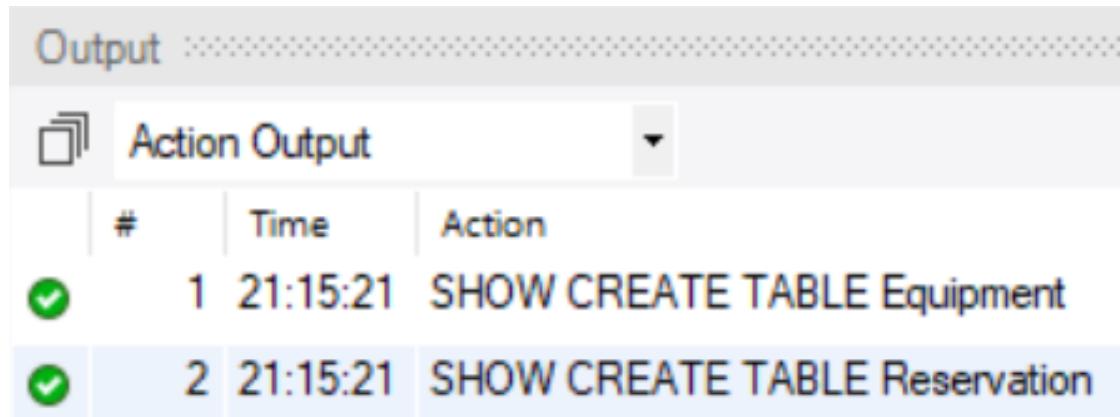


The screenshot shows the MySQL Workbench interface with the 'Action Output' tab selected. It displays two log entries:

#	Time	Action
1	11:50:09	CREATE TABLE IF NOT EXISTS `equipment` (`Equipment_ID` int NOT NULL AUTO_INCREMENT, `Equipment_Name` varc...
2	11:50:09	CREATE TABLE IF NOT EXISTS `reservation` (`Reservation_ID` int NOT NULL AUTO_INCREMENT, `Reservation_Date` da...

Figure 1.1: Execution evidence: Successful `CREATE TABLE` operation (Green Check / Query OK) for `equipment` and `reservation`, strictly corresponding to Task 1A constraints.

Figure 1.2 subsequently confirms the DDL stored in MySQL using `SHOW CREATE TABLE`, verifying that domain (CHECK), entity (PK/Unique), and referential (FK) constraints are persisted as designed.



The screenshot shows the MySQL Workbench interface with the 'Action Output' tab selected. It displays two log entries related to `SHOW CREATE TABLE` statements:

#	Time	Action
1	21:15:21	SHOW CREATE TABLE Equipment
2	21:15:21	SHOW CREATE TABLE Reservation

Figure 1.2: SQL verification: successful `SHOW CREATE TABLE` output for `equipment` and `reservation`.

1.4 Task 1B: How Integrity Constraints Are Applied and Maintained

1.4.1 Maintaining Domain Integrity (DB + Django)

Database enforcement. Domain rules are enforced at the DBMS boundary as the final, non-bypassable safeguard. For example, inserting a negative value into `equipment.Total_Quantity` violates the attribute domain (`INT UNSIGNED`) and is rejected by MySQL. Figure 1.3 shows `Error Code: 1264 (Out of range)`, which provides direct evidence that invalid domain values cannot be persisted.

Output			Message
#	Time	Action	
✖ 1	21:16:22	INSERT INTO Equipment (Equipment_Name, Type, Status, Total_Quantity) VALUES ('Negative Ba...')	Error Code: 1264. Out of range value for column 'Total_Quantity' at row 1

Figure 1.3: Domain integrity enforcement (SQL): inserting a negative `Total_Quantity` is rejected as an out-of-range value (`INT UNSIGNED`).

Application enforcement. While DB constraints guarantee correctness, the application layer should prevent invalid inputs early and provide user-friendly feedback. In our Django implementation, reservation-related domain rules are validated before database submission:

- **Time-slot conflict prevention:** the system checks whether the selected facility and time range overlaps with an existing reservation. If a conflict is detected, the request is blocked with a clear validation message (Figure 1.4).
- **Advance booking limit:** bookings can only be made up to 1 week in advance. Inputs beyond the allowed range are rejected with an explanatory error message (Figure 1.5).

Facility
Facility4 (Main Building-3-304)

Reservation date
2025 / 12 / 22 📅

Bookings can only be made up to 1 week in advance. Latest allowed date: 2025-12-20

Start time
17:00

End time
18:00

Submit request

Figure 1.4: Domain integrity enforcement (Django): reservation conflict detection prevents overlapping time slots at the application layer.

This time slot conflicts with an existing reservation. Please choose a different time.

Facility	Facility5 (Sports Hall B-1-105)
Reservation date	2025 / 12 / 18 <input type="button" value="Calendar"/>
Start time	14:00
End time	17:00
<input type="button" value="Submit request"/>	

Figure 1.5: Domain integrity enforcement (Django): the system enforces a 1-week advance booking policy and reports the latest allowed date.

Together, these validations demonstrate a robust strategy: Django provides early prevention and usability, while MySQL constraints remain the ultimate guardrail against invalid domain values.

1.4.2 Maintaining Entity Integrity

Entity integrity in our database is maintained primarily through `PRIMARY KEY` and `UNIQUE` constraints:

- **Stable identifiers:** auto-increment primary keys such as `Member_ID` and `Reservation_ID` ensure that each entity instance can be referenced unambiguously across the system.
- **Real-world uniqueness rules:** unique indices (e.g., the unique `user_id` mapping in `member`) prevent duplicate profiles for the same login account, preserving a consistent identity model.
- **Slot-level uniqueness:** the composite unique index on `reservation(Facility_ID, Reservation_Date, Start_Time)` prevents duplicated reservation slots and reduces the risk of double booking.

In Django, entity integrity is naturally reinforced by the ORM: each model instance is identified by its primary key, and unique constraints raise immediate errors if violated. This dual-layer alignment keeps identity and uniqueness consistent across the database and application.

1.4.3 Maintaining Referential Integrity

Referential integrity is maintained through foreign key constraints with restrictive actions:

- **No orphan bookings:** the `Reservation_ID` and `Member_ID` columns in `booking` must reference existing rows in `reservation` and `member` respectively.
- **Controlled deletes/updates:** the use of `ON DELETE RESTRICT` and `ON UPDATE RESTRICT` prevents accidental cascades that could silently destroy dependent records or break relationships.

On the Django side, referential integrity is supported by model relationships and form design: users select foreign-key objects from validated querysets, which ensures that referenced entities exist at the time of submission. As a result, relationships remain consistent even under concurrent usage.

1.5 Summary

This task demonstrates that our database integrity strategy is both **correct by design** and **verifiable in practice**. Domain constraints prevent invalid values (e.g., negative quantities and invalid booking dates), entity constraints guarantee unique and stable identities, and referential constraints preserve valid relationships between members, reservations, and bookings. By combining MySQL constraints with Django validation, the system achieves integrity that is rigorous, user-friendly, and aligned with the implemented schema in `sports_arena.sql`.

1.6 Schema Diagram (Supplementary)

Figure 1.6 provides a compact overview of the core relational schema used in this report. It clarifies the supertype–subtype design (`reservation` → `booking/training_session`) and the associative entities (`session_enrollment`, `reservation_equipments`), which supports the integrity discussion above.

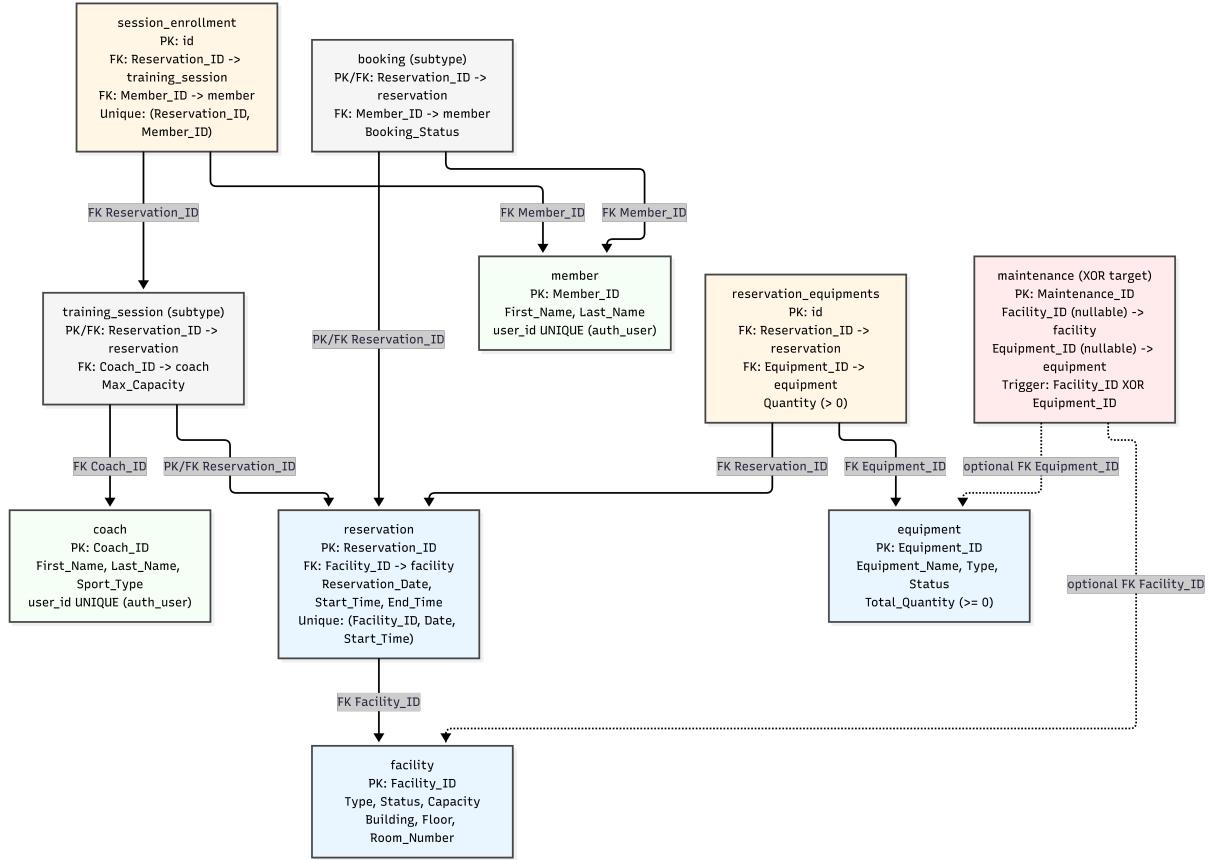


Figure 1.6: Core relational schema overview aligned with `sports_arena.sql`.

Task 2 SQL

This section demonstrates the practical implementation of the University Sports Complex Management System database. It covers the creation of the remaining tables, data population, and various data retrieval operations using SQL.

2.1 Table Creation

Following the initial examples in Task 1, we proceeded to create the remaining tables to complete the database schema. These tables include entities for different member types (Student, Staff, External Visitor), contact information, coaching staff, facility details, and various transaction tables like maintenance records and session enrollments.

The following screenshots demonstrate the successful execution of the `CREATE TABLE` commands for these entities.

Action Output			
#	Time	Action	
1	20:39:38	SHOW CREATE TABLE Student	
2	20:39:38	SHOW CREATE TABLE Staff	
3	20:39:38	SHOW CREATE TABLE External_Visitor	

Figure 2.1: *CREATE TABLE* execution (*student*, *staff*, *external_visitor*)

Action Output			
#	Time	Action	
1	20:40:48	SHOW CREATE TABLE Member_Email	
2	20:40:48	SHOW CREATE TABLE Member_Phone	
3	20:40:48	SHOW CREATE TABLE Coach_Email	
4	20:40:48	SHOW CREATE TABLE Coach_Phone	

Figure 2.2: *CREATE TABLE* execution (*member_email*, *member_phone*, *coach_email*, *coach_phone*)

Action Output		
#	Time	Action
1	13:38:37	SHOW CREATE TABLE Facility
2	13:38:37	SHOW CREATE TABLE Coach
3	13:38:37	SHOW CREATE TABLE Maintenance

Figure 2.3: *CREATE TABLE execution (facility, coach, maintenance)*

Action Output		
#	Time	Action
1	13:39:04	SHOW CREATE TABLE Visitor_Application
2	13:39:05	SHOW CREATE TABLE Training_Session
3	13:39:05	SHOW CREATE TABLE Session_Enrollment
4	13:39:05	SHOW CREATE TABLE Reservation_Equipments

Figure 2.4: *CREATE TABLE execution (visitor_application, training_session, session_enrollment, reservation_equipments)*

2.2 Data Population

To ensure the database is functional for testing and queries, we populated each table with at least 6 rows of data (meeting the required 6–10 rows where practical). In a few cases, additional rows were inserted to support more comprehensive trigger testing scenarios in Task 3. The screenshots below provide evidence of successful data population and a comprehensive row-count summary for all tables in the schema.

XIAMEN UNIVERSITY MALAYSIA

Action Output			Message
#	Time	Action	
1	21:04:56	USE sports_arena	0 row(s) affected
2	21:04:56	SET FOREIGN_KEY_CHECKS = 0	0 row(s) affected
3	21:04:56	INSERT IGNORE INTO Member (Member_ID, First_Name, Last_Name, Registration_Date, Membership...) VALUES (4, 'STU004'), (5, 'STU005'), (6, 'STU006')	7 row(s) affected Records: 7 Duplicates: 0 Warnings: 0
4	21:04:56	INSERT IGNORE INTO Student (Member_ID, Student_ID) VALUES (4, 'STU004'), (5, 'STU005'), (6, 'STU006')	3 row(s) affected Records: 3 Duplicates: 0 Warnings: 0
5	21:04:56	INSERT IGNORE INTO Staff (Member_ID, Staff_ID) VALUES (7, 'STA007'), (8, 'STA008')	2 row(s) affected Records: 2 Duplicates: 0 Warnings: 0
6	21:04:56	INSERT IGNORE INTO External_Visitor (Member_ID, IC_Number) VALUES (9, 'IC999009'), (10, 'IC999010')	2 row(s) affected Records: 2 Duplicates: 0 Warnings: 0
7	21:04:56	INSERT IGNORE INTO Member_Email (Member_ID, Email_Address) VALUES (4, 'alice.w@example.com'), (5, 'bob.l@example.com')	7 row(s) affected Records: 7 Duplicates: 0 Warnings: 0
8	21:04:56	INSERT IGNORE INTO Member_Phone (Member_ID, Phone_Number) VALUES (4, '13800000004'), (5, '13800000005'), (6, '13800000006')	5 row(s) affected Records: 5 Duplicates: 0 Warnings: 0
9	21:04:56	INSERT IGNORE INTO Coach (Coach_ID, First_Name, Last_Name, Sport_Type, Level) VALUES (...)	5 row(s) affected Records: 5 Duplicates: 0 Warnings: 0
10	21:04:56	INSERT IGNORE INTO Coach_Email (Coach_ID, Email_Address) VALUES (2, 'sarah.c@example.com'), (3, 'mike.t@example.com')	4 row(s) affected Records: 4 Duplicates: 0 Warnings: 0
11	21:04:56	INSERT IGNORE INTO Coach_Phone (Coach_ID, Phone_Number) VALUES (2, '13900000002'), (3, '13900000003'), (4, '1...')	3 row(s) affected Records: 3 Duplicates: 0 Warnings: 0
12	21:04:56	UPDATE Facility SET Status = 'Available' WHERE Facility_ID IN (1, 2, 3, 4, 5, 6, 10)	5 row(s) affected Rows matched: 7 Changed: 5 Warnings: 0
13	21:04:56	INSERT IGNORE INTO Reservation (Reservation_ID, Facility_ID, Reservation_Date, Start_Time, End...) VALUES (...)	7 row(s) affected Records: 7 Duplicates: 0 Warnings: 0
14	21:04:56	INSERT IGNORE INTO Booking (Reservation_ID, Member_ID, Booking_Status) VALUES (3, 4, 'Confirmed'), (...)	7 row(s) affected Records: 7 Duplicates: 0 Warnings: 0
15	21:04:56	INSERT IGNORE INTO Reservation (Reservation_ID, Facility_ID, Reservation_Date, Start_Time, End...) VALUES (...)	6 row(s) affected Records: 6 Duplicates: 0 Warnings: 0
16	21:04:56	INSERT IGNORE INTO Training_Session (Reservation_ID, Coach_ID, Max_Capacity) VALUES (9, 3, 20), (...)	6 row(s) affected Records: 6 Duplicates: 0 Warnings: 0
17	21:04:56	INSERT IGNORE INTO Session_Enrollment (Reservation_ID, Member_ID) VALUES (9, 4), (9, 5), (10, 6), (10, 7), (10, 8,...)	8 row(s) affected Records: 8 Duplicates: 0 Warnings: 0
18	21:04:57	UPDATE Equipment SET Status = 'Available' WHERE Equipment_ID IN (2, 5, 6, 10, 13)	2 row(s) affected Rows matched: 5 Changed: 2 Warnings: 0
19	21:04:57	INSERT IGNORE INTO Reservation_Equipments (Reservation_ID, Equipment_ID, Quantity) VALUES (3, 2, 2), (...)	5 row(s) affected Records: 5 Duplicates: 0 Warnings: 0
20	21:04:57	INSERT IGNORE INTO Maintenance (Maintenance_ID, Scheduled_Date, Status, Description, Facility...) VALUES (...)	5 row(s) affected Records: 5 Duplicates: 0 Warnings: 0
21	21:04:57	INSERT IGNORE INTO Visitor_Application (Application_ID, First_Name, Last_Name, IC_Number, Application_Status)	5 row(s) affected Records: 5 Duplicates: 0 Warnings: 0
22	21:04:57	INSERT IGNORE INTO Visitor_Application_Email (Application_ID, Email_Address) VALUES (9, 'tom@example.com'), (10, 'jerry...')	2 row(s) affected Records: 2 Duplicates: 0 Warnings: 0
23	21:04:57	INSERT IGNORE INTO Visitor_Application_Phone (Application_ID, Phone_Number) VALUES (9, '99900000009'), (10, '99900...')	2 row(s) affected Records: 2 Duplicates: 0 Warnings: 0

Figure 2.5: Data population execution evidence: representative *INSERT/UPDATE* statements executed successfully across the schema.

	Table_Name	Row_Count
▶	auth_group	2
	auth_user	5
	booking	8
	coach	6
	coach_email	4
	coach_phone	4
	equipment	15
	external_visitor	2
	facility	10
	maintenance	7
	member	10
	member_email	8
	member_phone	7
	reservation	15
	reservation_e...	7
	session_enroll...	8
	staff	3
	student	5
	training_session	7
	visitor_applicat...	13

Figure 2.6: Verification of Data Population: Comprehensive row counts for all tables (via `SELECT COUNT`), confirming dataset volume requirements.

2.3 Data Retrieval and Manipulation

We executed various SQL commands across six categories to demonstrate data retrieval and manipulation capabilities.

2.3.1 Filtering

This category demonstrates the use of `LIKE`, `BETWEEN`, and `IN` clauses to filter results.

Query 1: Filtering by Pattern (`LIKE`)

Find all facilities related to 'Tennis' that are currently available.

Listing 2.1: Filtering with `LIKE`

```

1 SELECT Facility_Name, Type, Status
2 FROM facility
3 WHERE Type LIKE '%Tennis%' AND Status='Available';

```

The screenshot shows a 'Result Grid' window with three columns: Facility_Name, Type, and Status. There are four rows of data:

	Facility_Name	Type	Status
▶	Facility3	Table Tennis Room	Available
	Facility4	Tennis Court	Available
	Facility5	Tennis Court	Available
	Facility6	Tennis Court	Available

Figure 2.7: Output of Query 1**Query 2: Filtering by Range (BETWEEN)**

Retrieve reservations made for the month of December 2025.

Listing 2.2: Filtering with BETWEEN

```

1 SELECT Reservation_ID, Reservation_Date
2 FROM reservation
3 WHERE Reservation_Date BETWEEN '2025-12-01' AND '2025-12-31';

```

The screenshot shows a 'Result Grid' window with two columns: Reservation_ID and Reservation_Date. There are 15 rows of data, each representing a reservation made in December 2025:

	Reservation_ID	Reservation_Date
▶	9	2025-12-14
	13	2025-12-17
	6	2025-12-15
	7	2025-12-16
	1	2025-12-11
	3	2025-12-14
	5	2025-12-15
	11	2025-12-16
	2	2025-12-17
	4	2025-12-14
	8	2025-12-16
	14	2025-12-18
	15	2025-12-17
	10	2025-12-15
	12	2025-12-16

Figure 2.8: Output of Query 2**Query 3: Filtering by Set (IN)**

List bookings that are either 'Pending' or 'Confirmed'.

Listing 2.3: Filtering with IN

```

1 SELECT Reservation_ID, Booking_Status
2 FROM booking
3 WHERE Booking_Status IN ('Pending', 'Confirmed');

```

	Reservation_ID	Booking_Status
▶	1	Confirmed
	3	Confirmed
	4	Pending
	5	Confirmed
	7	Confirmed
	8	Pending
	15	Confirmed

Figure 2.9: Output of Query 3

2.3.2 Aggregate Functions

This category uses functions like COUNT, SUM, and AVG to summarize data.

Query 4: Counting Records (COUNT)

Count the number of facilities for each type.

Listing 2.4: Aggregation with COUNT

```
1 SELECT Type, COUNT(*) AS facility_count
2 FROM facility
3 GROUP BY Type;
```

	Type	facility_count
▶	Swimming Pool	1
	Gym	3
	Table Tennis Room	2
	Tennis Court	4

Figure 2.10: Output of Query 4

Query 5: Summing Values (SUM)

Calculate the total quantity of equipment reserved for each reservation ID.

Listing 2.5: Aggregation with SUM

```
1 SELECT Reservation_ID, SUM(Quantity) AS total_equipment
2 FROM reservation_equipments
3 GROUP BY Reservation_ID;
```

	Reservation_ID	total_equipment
▶	1	5
	2	2
	3	2
	6	1
	9	5
	10	5
	14	4

Figure 2.11: Output of Query 5

Query 6: Averaging Values (AVG)

Calculate the average maximum capacity of training sessions for each coach.

Listing 2.6: Aggregation with AVG

```

1 SELECT Coach_ID, AVG(Max_Capacity) AS avg_capacity
2 FROM training_session
3 GROUP BY Coach_ID;

```

	Coach_ID	avg_capacity
▶	1	10.0000
	2	4.0000
	3	20.0000
	4	30.0000
	5	12.5000
	6	12.0000

Figure 2.12: Output of Query 6**2.3.3 Limit / Sorting**

This category demonstrates sorting results with `ORDER BY` and restricting output with `LIMIT`.

Query 7: Ordering and Limiting

List the top 3 available facilities with the highest capacity.

Listing 2.7: Sorting and Limiting results

```

1 SELECT Facility_Name, Capacity
2 FROM facility
3 WHERE Status='Available'
4 ORDER BY Capacity DESC
5 LIMIT 3;

```

	Facility_Name	Capacity
▶	Facility1	96
	Facility5	69
	Facility6	66

Figure 2.13: Output of Query 7**Query 8: Multi-level Sorting**

Show recent reservations ordered by date and time in descending order.

Listing 2.8: *Multi-column Sorting*

```

1 SELECT Reservation_ID, Reservation_Date, Start_Time
2 FROM reservation
3 ORDER BY Reservation_Date DESC, Start_Time DESC
4 LIMIT 5;

```

	Reservation_ID	Reservation_Date	Start_Time
▶	14	2025-12-18	14:00:00.000000
	2	2025-12-17	12:00:00.000000
	13	2025-12-17	10:00:00.000000
	15	2025-12-17	10:00:00.000000
	7	2025-12-16	15:00:00.000000

Figure 2.14: *Output of Query 8***Query 9: Sorting Applications**

List the 5 most recent visitor applications.

Listing 2.9: *Sorting Visitor Applications*

```

1 SELECT Application_ID, First_Name, Status
2 FROM visitor_application
3 ORDER BY Application_Date DESC
4 LIMIT 5;

```

	Application_ID	First_Name	Status
▶	13	Goofy	Approved
	12	Donald	Pending
	11	Mickey	Rejected
	10	Jerry	Pending
	9	Tom	Approved

Figure 2.15: *Output of Query 9*

2.3.4 Join Operators

This category demonstrates `INNER JOIN`, `LEFT JOIN`, and `RIGHT JOIN` to combine data from multiple tables.

Query 10: Inner Join

Retrieve booking details including member name, facility name, and booking status.

Listing 2.10: Inner Join Example

```

1 SELECT b.Reservation_ID, m.First_Name, f.Facility_Name, b.Booking_Status
2 FROM booking b
3 JOIN member m ON b.Member_ID=m.Member_ID
4 JOIN reservation r ON b.Reservation_ID=r.Reservation_ID
5 JOIN facility f ON r.Facility_ID=f.Facility_ID;

```

	Reservation_ID	First_Name	Facility_Name	Booking_Status
▶	1	M	Facility4	Confirmed
	3	Alice	Facility4	Confirmed
	4	Bob	Facility5	Pending
	5	Charlie	Facility4	Confirmed
	6	Alice	Facility2	Cancelled
	7	Bob	Facility3	Confirmed
	8	David	Facility5	Pending
	15	Eva	Facility6	Confirmed

Figure 2.16: Output of Query 10

Query 11: Left Join

List all training sessions and their coaches (if assigned), facilities, and capacity.

Listing 2.11: Left Join Example

```

1 SELECT ts.Reservation_ID, c.First_Name, f.Facility_Name,
2      ts.Max_Capacity
3 FROM training_session ts
4 LEFT JOIN coach c ON ts.Coach_ID=c.Coach_ID
5 LEFT JOIN reservation r ON ts.Reservation_ID=r.Reservation_ID
6 LEFT JOIN facility f ON r.Facility_ID=f.Facility_ID;

```

	Reservation_ID	First_Name	Facility_Name	Max_Capacity
▶	2	C	Facility4	10
	9	Mike	Facility1	20
	10	Serena	Facility10	15
	11	Sarah	Facility4	4
	12	Serena	Facility10	10
	13	LeBron	Facility1	30
	14	Yao	Facility5	12

Figure 2.17: Output of Query 11

Query 12: Right Join

List equipment reservation-assignment rows, demonstrating `RIGHT JOIN` syntax.

Listing 2.12: Right Join Example

```

1 SELECT e.Equipment_Name, re.Reservation_ID, re.Quantity
2 FROM equipment e
3 RIGHT JOIN reservation_equipments re ON e.Equipment_ID=re.Equipment_ID;

```

	Equipment_Name	Reservation_ID	Quantity
▶	Equipment2	1	5
	Equipment6	2	2
	Equipment2	3	2
	Equipment6	6	1
	Equipment5	9	5
	Equipment13	10	5
	Equipment2	14	4

Figure 2.18: Output of Query 12

2.3.5 String / Arithmetic Operations

This category performs calculations and string manipulations on retrieved data.

Query 13: Arithmetic Expression

Calculate the duration of reservations in hours.

Listing 2.13: Arithmetic Operation

```

1 SELECT Reservation_ID,
2       TIMESTAMPDIFF(MINUTE, Start_Time, End_Time)/60 AS duration_hours
3 FROM reservation;

```

	Reservation_ID	duration_hours
▶	1	0.0167
	2	4.0000
	3	1.0000
	4	1.0000
	5	1.0000
	6	2.0000
	7	1.0000
	8	1.0000
	9	2.0000
	10	2.0000
	11	2.0000
	12	2.0000
	13	2.0000
	14	2.0000
	15	2.0000

Figure 2.19: Output of Query 13

Query 14: String Concatenation

Combine first and last names into a single full name column for members.

Listing 2.14: *String Concatenation*

```
1 SELECT CONCAT(First_Name, ' ', Last_Name) AS member_name,
2       Membership_Status
3 FROM member;
```

	member_name	Membership_Status
▶	M 1	Active
	M 2	Active
	Kailong Deng	Active
	Alice Wang	Active
	Bob Lee	Active
	Charlie Chen	Active
	David Zhang	Active
	Eva Wu	Active
	Frank Liu	Pending_Approval
	Grace Xu	Inactive

Figure 2.20: *Output of Query 14***Query 15: Complex Calculation**

Calculate the theoretical remaining quantity of equipment after reservations.

Listing 2.15: *Complex Arithmetic*

```
1 SELECT re.Reservation_ID, re.Quantity, e.Total_Quantity,
2       (e.Total_Quantity - re.Quantity) AS remaining_theoretical
3 FROM reservation_equipments re
4 JOIN equipment e ON re.Equipment_ID=e.Equipment_ID;
```

	Reservation_ID	Quantity	Total_Quantity	remaining_theoretical
▶	1	5	20	15
	2	2	6	4
	3	2	20	18
	6	1	6	5
	9	5	6	1
	10	5	28	23
	14	4	20	16

Figure 2.21: *Output of Query 15***2.3.6 Formatting**

This category focuses on formatting query output using aliases and concatenation for better readability.

Query 16: Column Aliasing

Rename columns for a clear facility location report.

Listing 2.16: Column Aliasing

```
1 SELECT Facility_ID AS ID, Facility_Name AS Name,
2       CONCAT(Building, '-', Floor, '-', Room_Number) AS Location
3 FROM facility;
```

	ID	Name	Location
▶	1	Facility1	Main Building-4-401
	2	Facility2	Main Building-2-202
	3	Facility3	Sports Hall B-4-403
	4	Facility4	Main Building-3-304
	5	Facility5	Sports Hall B-1-105
	6	Facility6	Sports Hall B-1-106
	7	Facility7	Activity Center-2-207
	8	Facility8	Activity Center-3-308
	9	Facility9	Main Building-3-309
	10	Facility10	Sports Hall B-1-110

Figure 2.22: Output of Query 16**Query 17: Date Formatting**

Format reservation dates and create user-friendly booking summaries.

Listing 2.17: Date Formatting and Concatenation

```
1 SELECT b.Reservation_ID AS BookingID,
2       CONCAT(m.First_Name, ' ', m.Last_Name) AS Member,
3       DATE_FORMAT(r.Reservation_Date, '%Y-%m-%d') AS Date
4 FROM booking b
5 JOIN member m ON b.Member_ID=m.Member_ID
6 JOIN reservation r ON b.Reservation_ID=r.Reservation_ID;
```

	BookingID	Member	Date
▶	1	M 1	2025-12-11
	3	Alice Wang	2025-12-14
	6	Alice Wang	2025-12-15
	4	Bob Lee	2025-12-14
	7	Bob Lee	2025-12-16
	5	Charlie Chen	2025-12-15
	8	David Zhang	2025-12-16
	15	Eva Wu	2025-12-17

Figure 2.23: Output of Query 17

Query 18: Session Formatting

Create a formatted schedule for training sessions.

Listing 2.18: *Result Set Formatting*

```

1 SELECT ts.Reservation_ID AS SessionID,
2     CONCAT(c.First_Name, ' ', c.Last_Name) AS CoachName,
3     CONCAT(r.Start_Time, ' - ', r.End_Time) AS TimeSlot
4 FROM training_session ts
5 JOIN coach c ON ts.Coach_ID=c.Coach_ID
6 JOIN reservation r ON ts.Reservation_ID=r.Reservation_ID;

```

	SessionID	CoachName	TimeSlot
▶	2	C 1	12:00:00.000000-16:00:00.000000
	11	Sarah Connor	09:00:00.000000-11:00:00.000000
	9	Mike Tyson	08:00:00.000000-10:00:00.000000
	13	LeBron James	10:00:00.000000-12:00:00.000000
	10	Serena Williams	18:00:00.000000-20:00:00.000000
	12	Serena Williams	14:00:00.000000-16:00:00.000000
	14	Yao Ming	14:00:00.000000-16:00:00.000000

Figure 2.24: *Output of Query 18*

Task 3 Triggering

To ensure data integrity and enforce business rules at the database level, we selected two trigger event types—**INSERT** and **UPDATE**—and implemented **BEFORE** triggers so invalid operations are rejected before any row is committed. All trigger definitions shown in this section are aligned with the deployed schema in `sports_arena.sql`. The evidence scenarios below explicitly show the database state *before* and *after* each attempted operation (success or rejection).

3.1 Implemented Triggers

3.1.1 Booking Guardrails

These triggers ensure a member cannot have more than 2 pending bookings, sessions do not exceed 3 hours, and bookings are made within a valid 7-day window.

Listing 3.1: *Trigger: Booking Before Insert Checks*

```

1 CREATE TRIGGER `booking_bi_guardrails` BEFORE INSERT ON `booking` FOR EACH ROW BEGIN
2     DECLARE v_pending INT DEFAULT 0;
3     DECLARE v_reservation_date DATE;
4     DECLARE v_start TIME;
5     DECLARE v_end TIME;
6     DECLARE v_duration_minutes INT;

```

```

7
8 SELECT COUNT(*) INTO v_pending
9 FROM `booking`
10 WHERE `Member_ID` = NEW.Member_ID
11     AND `Booking_Status` = 'Pending';
12
13 IF NEW.Booking_Status = 'Pending' AND v_pending >= 2 THEN
14     SIGNAL SQLSTATE '45000'
15     SET MESSAGE_TEXT = 'Member already has 2 pending bookings';
16 END IF;
17
18 SELECT `Reservation_Date`, `Start_Time`, `End_Time`
19     INTO v_reservation_date, v_start, v_end
20 FROM `reservation`
21 WHERE `Reservation_ID` = NEW.Reservation_ID;
22
23 IF v_reservation_date IS NULL THEN
24     SIGNAL SQLSTATE '45000'
25     SET MESSAGE_TEXT = 'Reservation not found for booking';
26 END IF;
27
28 SET v_duration_minutes = TIMESTAMPDIFF(MINUTE, v_start, v_end);
29
30 IF v_duration_minutes > 180 THEN
31     SIGNAL SQLSTATE '45000'
32     SET MESSAGE_TEXT = 'Single booking session cannot exceed 3 hours';
33 END IF;
34
35 IF v_reservation_date < CURDATE() THEN
36     SIGNAL SQLSTATE '45000'
37     SET MESSAGE_TEXT = 'Cannot book for past dates';
38 END IF;
39
40 IF v_reservation_date > DATE_ADD(CURDATE(), INTERVAL 7 DAY) THEN
41     SIGNAL SQLSTATE '45000'
42     SET MESSAGE_TEXT = 'Bookings can only be made up to 7 days in advance';
43 END IF;
44 END;

```

Listing 3.2: Trigger: Booking Before Update Checks

```

1 CREATE TRIGGER `booking_bu_guardrails` BEFORE UPDATE ON `booking` FOR EACH ROW BEGIN
2 DECLARE v_pending INT DEFAULT 0;
3 DECLARE v_reservation_date DATE;
4 DECLARE v_start TIME;
5 DECLARE v_end TIME;
6 DECLARE v_duration_minutes INT;

```

```

7
8 SELECT COUNT(*) INTO v_pending
9 FROM `booking`
10 WHERE `Member_ID` = NEW.Member_ID
11     AND `Booking_Status` = 'Pending'
12     AND `Reservation_ID` <> OLD.Reservation_ID;
13
14 IF NEW.Booking_Status = 'Pending' AND v_pending >= 2 THEN
15     SIGNAL SQLSTATE '45000'
16     SET MESSAGE_TEXT = 'Member already has 2 pending bookings';
17 END IF;
18
19 IF NEW.Reservation_ID <> OLD.Reservation_ID THEN
20     SELECT `Reservation_Date`, `Start_Time`, `End_Time`
21     INTO v_reservation_date, v_start, v_end
22     FROM `reservation`
23     WHERE `Reservation_ID` = NEW.Reservation_ID;
24
25     IF v_reservation_date IS NULL THEN
26         SIGNAL SQLSTATE '45000'
27         SET MESSAGE_TEXT = 'Reservation not found for booking';
28     END IF;
29
30     SET v_duration_minutes = TIMESTAMPDIFF(MINUTE, v_start, v_end);
31
32     IF v_duration_minutes > 180 THEN
33         SIGNAL SQLSTATE '45000'
34         SET MESSAGE_TEXT = 'Single booking session cannot exceed 3 hours';
35     END IF;
36
37     IF v_reservation_date < CURDATE() THEN
38         SIGNAL SQLSTATE '45000'
39         SET MESSAGE_TEXT = 'Cannot book for past dates';
40     END IF;
41
42     IF v_reservation_date > DATE_ADD(CURDATE(), INTERVAL 7 DAY) THEN
43         SIGNAL SQLSTATE '45000'
44         SET MESSAGE_TEXT = 'Bookings can only be made up to 7 days in advance';
45     END IF;
46 END IF;
47 END;

```

3.1.2 Maintenance XOR Constraints

These triggers enforce that a maintenance record targets either a facility OR equipment, but not both and not neither.

Listing 3.3: Trigger: Maintenance Before Insert XOR Constraint

```

1 CREATE TRIGGER `maintenance_bi_xor` BEFORE INSERT ON `maintenance` FOR EACH ROW
2 BEGIN
3 IF (NEW.Facility_ID IS NULL AND NEW.Equipment_ID IS NULL)
4     OR (NEW.Facility_ID IS NOT NULL AND NEW.Equipment_ID IS NOT NULL) THEN
5     SIGNAL SQLSTATE '45000'
6     SET MESSAGE_TEXT = 'Maintenance must target either a facility or equipment (
7     exclusively)';
8 END IF;
9 END;

```

Listing 3.4: Trigger: Maintenance Before Update XOR Constraint

```

1 CREATE TRIGGER `maintenance_bu_xor` BEFORE UPDATE ON `maintenance` FOR EACH ROW
2 BEGIN
3 IF (NEW.Facility_ID IS NULL AND NEW.Equipment_ID IS NULL)
4     OR (NEW.Facility_ID IS NOT NULL AND NEW.Equipment_ID IS NOT NULL) THEN
5     SIGNAL SQLSTATE '45000'
6     SET MESSAGE_TEXT = 'Maintenance must target either a facility or equipment (
7     exclusively)';
8 END IF;
9 END;

```

3.1.3 Reservation Overlap and Availability

These triggers prevent overlapping reservations for the same facility and ensure the facility is available and not under maintenance.

Listing 3.5: Trigger: Reservation Before Insert Checks

```

1 CREATE TRIGGER `reservation_bi_guardrails` BEFORE INSERT ON `reservation` FOR EACH
2 ROW BEGIN
3 DECLARE v_facility_status VARCHAR(20);
4 DECLARE v_conflicts INT DEFAULT 0;
5 DECLARE v_maintenance INT DEFAULT 0;
6
7 SELECT `Status` INTO v_facility_status
8 FROM `facility`
9 WHERE `Facility_ID` = NEW.Facility_ID
10 FOR UPDATE;
11
12 IF v_facility_status IS NULL THEN
13     SIGNAL SQLSTATE '45000'
14     SET MESSAGE_TEXT = 'Facility does not exist for this reservation';
15 END IF;

```

```

16 IF NEW.Start_Time >= NEW.End_Time THEN
17     SIGNAL SQLSTATE '45000'
18     SET MESSAGE_TEXT = 'Reservation start time must be earlier than end time';
19 END IF;
20
21 IF v_facility_status <> 'Available' THEN
22     SIGNAL SQLSTATE '45000'
23     SET MESSAGE_TEXT = 'Facility is not available for reservation';
24 END IF;
25
26 SELECT COUNT(*) INTO v_maintenance
27 FROM `maintenance` m
28 WHERE m.`Facility_ID` = NEW.Facility_ID
29     AND m.`Scheduled_Date` = NEW.Reservation_Date
30     AND m.`Status` IN ('Scheduled', 'In_Progress', 'In Progress');
31
32 IF v_maintenance > 0 THEN
33     SIGNAL SQLSTATE '45000'
34     SET MESSAGE_TEXT = 'Facility has maintenance scheduled for the requested date';
35 END IF;
36
37 SELECT COUNT(*) INTO v_conflicts
38 FROM `reservation` r
39 WHERE r.`Facility_ID` = NEW.Facility_ID
40     AND r.`Reservation_Date` = NEW.Reservation_Date
41     AND NOT (NEW.End_Time <= r.`Start_Time` OR NEW.Start_Time >= r.`End_Time`);
42
43 IF v_conflicts > 0 THEN
44     SIGNAL SQLSTATE '45000'
45     SET MESSAGE_TEXT = 'Reservation overlaps with an existing booking for this
        facility';
46 END IF;
47 END;

```

Listing 3.6: Trigger: Reservation Before Update Checks

```

1 CREATE TRIGGER `reservation_bu_guardrails` BEFORE UPDATE ON `reservation` FOR EACH
    ROW BEGIN
2 DECLARE v_facility_status VARCHAR(20);
3 DECLARE v_conflicts INT DEFAULT 0;
4 DECLARE v_maintenance INT DEFAULT 0;
5
6 SELECT `Status` INTO v_facility_status
7 FROM `facility`
8 WHERE `Facility_ID` = NEW.Facility_ID
9 FOR UPDATE;
10

```

```

11 IF v_facility_status IS NULL THEN
12   SIGNAL SQLSTATE '45000'
13   SET MESSAGE_TEXT = 'Facility does not exist for this reservation';
14 END IF;
15
16 IF NEW.Start_Time >= NEW.End_Time THEN
17   SIGNAL SQLSTATE '45000'
18   SET MESSAGE_TEXT = 'Reservation start time must be earlier than end time';
19 END IF;
20
21 IF v_facility_status <> 'Available' THEN
22   SIGNAL SQLSTATE '45000'
23   SET MESSAGE_TEXT = 'Facility is not available for reservation';
24 END IF;
25
26 SELECT COUNT(*) INTO v_maintenance
27 FROM `maintenance` m
28 WHERE m.`Facility_ID` = NEW.Facility_ID
29   AND m.`Scheduled_Date` = NEW.Reservation_Date
30   AND m.`Status` IN ('Scheduled', 'In_Progress', 'In Progress');
31
32 IF v_maintenance > 0 THEN
33   SIGNAL SQLSTATE '45000'
34   SET MESSAGE_TEXT = 'Facility has maintenance scheduled for the requested date';
35 END IF;
36
37 SELECT COUNT(*) INTO v_conflicts
38 FROM `reservation` r
39 WHERE r.`Facility_ID` = NEW.Facility_ID
40   AND r.`Reservation_Date` = NEW.Reservation_Date
41   AND r.`Reservation_ID` <> OLD.Reservation_ID
42   AND NOT (NEW.End_Time <= r.`Start_Time` OR NEW.Start_Time >= r.`End_Time`);
43
44 IF v_conflicts > 0 THEN
45   SIGNAL SQLSTATE '45000'
46   SET MESSAGE_TEXT = 'Reservation overlaps with an existing booking for this
47   facility';
48 END IF;
49

```

3.1.4 Reservation Equipment Availability

These triggers ensure enough equipment stock is available for a reservation, accounting for other reservations and maintenance.

Listing 3.7: Trigger: *Reservation Equipments Before Insert Checks*

```

1 CREATE TRIGGER `reservation_equipments_bi_guardrails` BEFORE INSERT ON `reservation_equipments` FOR EACH ROW BEGIN
2     DECLARE v_total INT;
3     DECLARE v_reserved INT DEFAULT 0;
4     DECLARE v_available INT;
5     DECLARE v_status VARCHAR(20);
6     DECLARE v_reservation_date DATE;
7     DECLARE v_start TIME;
8     DECLARE v_end TIME;
9     DECLARE v_maintenance INT DEFAULT 0;
10
11    SELECT r.`Reservation_Date`, r.`Start_Time`, r.`End_Time`
12        INTO v_reservation_date, v_start, v_end
13    FROM `reservation` r
14   WHERE r.`Reservation_ID` = NEW.Reservation_ID
15  FOR UPDATE;
16
17  IF v_reservation_date IS NULL THEN
18      SIGNAL SQLSTATE '45000'
19      SET MESSAGE_TEXT = 'Reservation not found for equipment assignment';
20  END IF;
21
22  SELECT e.`Total_Quantity`, e.`Status`
23      INTO v_total, v_status
24  FROM `equipment` e
25 WHERE e.`Equipment_ID` = NEW.Equipment_ID
26  FOR UPDATE;
27
28  IF v_status IS NULL THEN
29      SIGNAL SQLSTATE '45000'
30      SET MESSAGE_TEXT = 'Equipment does not exist';
31  END IF;
32
33  IF v_status <> 'Available' THEN
34      SIGNAL SQLSTATE '45000'
35      SET MESSAGE_TEXT = 'Equipment is not available for reservation';
36  END IF;
37
38  IF NEW.Quantity <= 0 THEN
39      SIGNAL SQLSTATE '45000'
40      SET MESSAGE_TEXT = 'Quantity must be positive';
41  END IF;
42
43  SELECT COUNT(*) INTO v_maintenance
44  FROM `maintenance` m
45 WHERE m.`Equipment_ID` = NEW.Equipment_ID
46     AND m.`Scheduled_Date` = v_reservation_date

```

```

47     AND m.`Status` IN ('Scheduled', 'In_Progress', 'In Progress');
48
49 SELECT COALESCE(SUM(re2.`Quantity`), 0) INTO v_reserved
50 FROM `reservation_equipments` re2
51 JOIN `reservation` r2 ON r2.`Reservation_ID` = re2.`Reservation_ID`
52 WHERE re2.`Equipment_ID` = NEW.Equipment_ID
53     AND r2.`Reservation_Date` = v_reservation_date
54     AND NOT (v_end <= r2.`Start_Time` OR v_start >= r2.`End_Time`);
55
56 SET v_available = v_total - v_reserved - v_maintenance;
57
58 IF NEW.Quantity > v_available THEN
59     SIGNAL SQLSTATE '45000'
60     SET MESSAGE_TEXT = 'Requested equipment exceeds available quantity';
61 END IF;
62 END;

```

Listing 3.8: Trigger: Reservation Equipments Before Update Checks

```

1 CREATE TRIGGER `reservation_equipments_bu_guardrails` BEFORE UPDATE ON `reservation_equipments` FOR EACH ROW BEGIN
2     DECLARE v_total INT;
3     DECLARE v_reserved INT DEFAULT 0;
4     DECLARE v_available INT;
5     DECLARE v_status VARCHAR(20);
6     DECLARE v_reservation_date DATE;
7     DECLARE v_start TIME;
8     DECLARE v_end TIME;
9     DECLARE v_maintenance INT DEFAULT 0;
10
11    SELECT r.`Reservation_Date`, r.`Start_Time`, r.`End_Time`
12        INTO v_reservation_date, v_start, v_end
13    FROM `reservation` r
14   WHERE r.`Reservation_ID` = NEW.Reservation_ID
15  FOR UPDATE;
16
17  IF v_reservation_date IS NULL THEN
18      SIGNAL SQLSTATE '45000'
19      SET MESSAGE_TEXT = 'Reservation not found for equipment assignment';
20  END IF;
21
22  SELECT e.`Total_Quantity`, e.`Status`
23      INTO v_total, v_status
24  FROM `equipment` e
25 WHERE e.`Equipment_ID` = NEW.Equipment_ID
26  FOR UPDATE;
27

```

```

28 IF v_status IS NULL THEN
29     SIGNAL SQLSTATE '45000'
30     SET MESSAGE_TEXT = 'Equipment does not exist';
31 END IF;
32
33 IF v_status <> 'Available' THEN
34     SIGNAL SQLSTATE '45000'
35     SET MESSAGE_TEXT = 'Equipment is not available for reservation';
36 END IF;
37
38 IF NEW.Quantity <= 0 THEN
39     SIGNAL SQLSTATE '45000'
40     SET MESSAGE_TEXT = 'Quantity must be positive';
41 END IF;
42
43 SELECT COUNT(*) INTO v_maintenance
44 FROM `maintenance` m
45 WHERE m.`Equipment_ID` = NEW.Equipment_ID
46     AND m.`Scheduled_Date` = v_reservation_date
47     AND m.`Status` IN ('Scheduled', 'In_Progress', 'In Progress');
48
49 SELECT COALESCE(SUM(re2.`Quantity`), 0) INTO v_reserved
50 FROM `reservation_equipments` re2
51 JOIN `reservation` r2 ON r2.`Reservation_ID` = re2.`Reservation_ID`
52 WHERE re2.`Equipment_ID` = NEW.Equipment_ID
53     AND r2.`Reservation_Date` = v_reservation_date
54     AND NOT (v_end <= r2.`Start_Time` OR v_start >= r2.`End_Time`)
55     AND re2.`id` <> OLD.`id`;
56
57 SET v_available = v_total - v_reserved - v_maintenance;
58
59 IF NEW.Quantity > v_available THEN
60     SIGNAL SQLSTATE '45000'
61     SET MESSAGE_TEXT = 'Requested equipment exceeds available quantity';
62 END IF;
63 END;

```

3.1.5 Session Enrollment Capacity

These triggers prevent enrollment if the training session has reached its maximum capacity.

Listing 3.9: Trigger: Session Enrollment Before Insert Capacity Check

```

1 CREATE TRIGGER `session_enrollment_bi_capacity` BEFORE INSERT ON `session_enrollment`
  ` FOR EACH ROW BEGIN
2 DECLARE v_capacity INT;
3 DECLARE v_current INT DEFAULT 0;

```

```

4
5 SELECT `Max_Capacity` INTO v_capacity
6 FROM `training_session`
7 WHERE `Reservation_ID` = NEW.Reservation_ID
8 FOR UPDATE;
9
10 IF v_capacity IS NULL THEN
11     SIGNAL SQLSTATE '45000'
12     SET MESSAGE_TEXT = 'Training session not found for enrollment';
13 END IF;
14
15 IF v_capacity <= 0 THEN
16     SIGNAL SQLSTATE '45000'
17     SET MESSAGE_TEXT = 'Training session has no available capacity';
18 END IF;
19
20 SELECT COUNT(*) INTO v_current
21 FROM `session_enrollment`
22 WHERE `Reservation_ID` = NEW.Reservation_ID;
23
24 IF v_current >= v_capacity THEN
25     SIGNAL SQLSTATE '45000'
26     SET MESSAGE_TEXT = 'Training session is full';
27 END IF;
28 END;

```

Listing 3.10: Trigger: Session Enrollment Before Update Capacity Check

```

1 CREATE TRIGGER `session_enrollment_bu_capacity` BEFORE UPDATE ON `session_enrollment`
  ` FOR EACH ROW BEGIN
2 DECLARE v_capacity INT;
3 DECLARE v_current INT DEFAULT 0;
4
5 SELECT `Max_Capacity` INTO v_capacity
6 FROM `training_session`
7 WHERE `Reservation_ID` = NEW.Reservation_ID
8 FOR UPDATE;
9
10 IF v_capacity IS NULL THEN
11     SIGNAL SQLSTATE '45000'
12     SET MESSAGE_TEXT = 'Training session not found for enrollment';
13 END IF;
14
15 IF v_capacity <= 0 THEN
16     SIGNAL SQLSTATE '45000'
17     SET MESSAGE_TEXT = 'Training session has no available capacity';
18 END IF;

```

```

19
20 SELECT COUNT(*) INTO v_current
21 FROM `session_enrollment`
22 WHERE `Reservation_ID` = NEW.Reservation_ID
23     AND `id` <> OLD.`id`;
24
25 IF v_current >= v_capacity THEN
26     SIGNAL SQLSTATE '45000'
27     SET MESSAGE_TEXT = 'Training session is full';
28 END IF;
29 END;

```

3.2 Evidence Scenarios

3.2.1 Trigger Interaction Diagram

Figure 3.1 summarizes how the implemented triggers collaborate to enforce business rules in the reservation workflow. It also highlights the shared dependency on base constraints (e.g., foreign keys, checks) and the common rejection behavior (`SIGNAL SQLSTATE '45000'`).

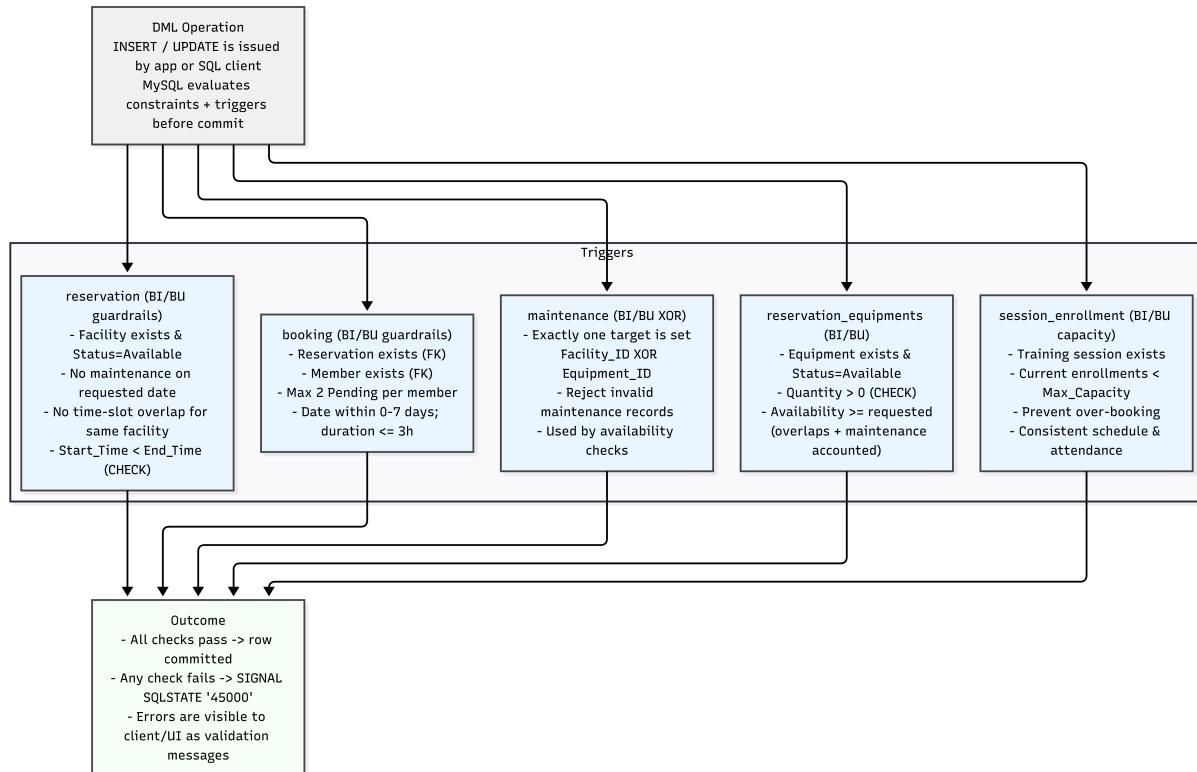


Figure 3.1: Trigger enforcement overview: major trigger points and the resulting accept/reject outcomes during `INSERT/UPDATE`.

3.2.2 Scenario A: Reservation Overlap is Blocked

Before: The target facility has two valid, non-overlapping reservations on the chosen date (Scenario A1). **Action:** Attempt to insert a reservation that overlaps an existing time interval for the same facility. **After:** The trigger blocks the overlapping insert with an error (Scenario A2), and the reservation list remains unchanged (Scenario A3).

Listing 3.11: Scenario A1 SQL: Create two non-overlapping reservations

```

1 SET @res_facility := 4;
2 SET @res_date := DATE_ADD(CURDATE(), INTERVAL 1 DAY);
3 INSERT INTO reservation (Reservation_Date, Start_Time, End_Time, Facility_ID)
4 VALUES (@res_date, '10:00:00', '11:00:00', @res_facility);
5 SET @res_ok_1 := LAST_INSERT_ID();
6 INSERT INTO reservation (Reservation_Date, Start_Time, End_Time, Facility_ID)
7 VALUES (@res_date, '12:00:00', '13:00:00', @res_facility);
8 SET @res_ok_2 := LAST_INSERT_ID();
9 SELECT 'current reservations' AS step, Reservation_ID, Start_Time, End_Time
10 FROM reservation WHERE Reservation_ID IN (@res_ok_1, @res_ok_2) ORDER BY
    Reservation_ID;
```

step	Reservation_ID	Start_Time	End_Time
▶ current reservations	3	10:00:00.000000	11:00:00.000000
▶ current reservations	4	12:00:00.000000	13:00:00.000000

Figure 3.2: Scenario A1: Successful creation of non-overlapping reservations

Then, we attempt an overlapping insert, which should fail.

Listing 3.12: Scenario A2 SQL: Attempt overlapping reservation

```

1 INSERT INTO reservation (Reservation_Date, Start_Time, End_Time, Facility_ID)
2 VALUES (@res_date, '10:30:00', '11:30:00', @res_facility);
```

✖ 881 15:31:51 INSERT INTO reservation (Reservation_Date, Start_... Error Code: 1644. Reservation overlaps with an existing booking for this facility

Figure 3.3: Scenario A2: Overlapping reservation attempt blocked

Confirmation that only the valid rows remain:

Listing 3.13: Scenario A3 SQL: Verify reservations found

```

1 SELECT Reservation_ID, Start_Time, End_Time
2 FROM reservation
3 WHERE Facility_ID = @res_facility AND Reservation_Date = @res_date
4 ORDER BY Reservation_ID;
```

	Reservation_ID	Start_Time	End_Time
▶	3	10:00:00.000000	11:00:00.000000
●	4	12:00:00.000000	13:00:00.000000
•	NULL	NULL	NULL

Figure 3.4: Scenario A3: Verification of existing reservations

3.2.3 Scenario B: Booking Pending Limit

Rule: A member may have at most two Pending bookings. **Before:** Member 1 has fewer than two pending bookings. **Action:** Prepare three future reservation slots (Scenario B1), insert two pending bookings (Scenario B2), then attempt a third pending booking (Scenario B3). **After:** The trigger raises 'Member already has 2 pending bookings' and the database still contains only two pending bookings for the member (Scenario B4).

Listing 3.14: Scenario B1 SQL: Prepare three reservations

```

1 SET @booking_member := 1;
2 SET @booking_date := DATE_ADD(CURDATE(), INTERVAL 2 DAY);
3 INSERT INTO reservation (Reservation_Date, Start_Time, End_Time, Facility_ID)
4 VALUES (@booking_date, '09:00:00', '10:00:00', 5);
5 SET @book_res_1 := LAST_INSERT_ID();
6 INSERT INTO reservation (Reservation_Date, Start_Time, End_Time, Facility_ID)
7 VALUES (@booking_date, '10:10:00', '11:10:00', 5);
8 SET @book_res_2 := LAST_INSERT_ID();
9 INSERT INTO reservation (Reservation_Date, Start_Time, End_Time, Facility_ID)
10 VALUES (@booking_date, '11:20:00', '12:20:00', 5);
11 SET @book_res_3 := LAST_INSERT_ID();
12 SELECT 'prep reservations' AS step, Reservation_ID, Reservation_Date, Start_Time,
    End_Time
13 FROM reservation WHERE Reservation_ID IN (@book_res_1, @book_res_2, @book_res_3)
14 ORDER BY Reservation_ID;
```

	step	Reservation_ID	Reservation_Date	Start_Time	End_Time
▶	prep reservations	3	2025-12-13	09:00:00.000000	10:00:00.000000
●	prep reservations	4	2025-12-13	10:10:00.000000	11:10:00.000000
•	prep reservations	5	2025-12-13	11:20:00.000000	12:20:00.000000
•	prep reservations	NULL	NULL	NULL	NULL

Figure 3.5: Scenario B1: Preparation of reservations

Insert two Pending bookings (allowed) and count:

Listing 3.15: Scenario B2 SQL: Insert two pending bookings

```

1 INSERT INTO booking (Reservation_ID, Booking_Status, Member_ID)
2 VALUES (@book_res_1, 'Pending', @booking_member);
3 INSERT INTO booking (Reservation_ID, Booking_Status, Member_ID)
4 VALUES (@book_res_2, 'Pending', @booking_member);
5 SELECT 'after two pendings' AS step, COUNT(*) AS pending_count
6 FROM booking WHERE Member_ID = @booking_member AND Booking_Status = 'Pending';

```

step	pending_count
after two pendings	2

Figure 3.6: Scenario B2: Two pending bookings successfully created

Attempt third Pending (should fail) and verify list:

Listing 3.16: Scenario B3 SQL: Attempt third pending booking

```

1 INSERT INTO booking (Reservation_ID, Booking_Status, Member_ID)
2 VALUES (@book_res_3, 'Pending', @booking_member);
3 SELECT Reservation_ID, Booking_Status
4 FROM booking WHERE Member_ID = @booking_member ORDER BY Reservation_ID;

```

✖ 1300 16:01:04 INSERT INTO booking (Reservation_ID, Booking_Status, Me... Error Code: 1644. Member already has 2 pending bookings

Figure 3.7: Scenario B3: Third pending booking blocked

	Reservation_ID	Booking_Status
▶	1	Confirmed
	3	Pending
	4	Pending
■	NULL	NULL

Figure 3.8: Scenario B4: Verification of booking list

3.2.4 Scenario C: Maintenance XOR Constraint

Rule: Maintenance must target either a facility *or* equipment (exclusive-or), never both and never neither. **Before:** No invalid maintenance row exists for the tested combinations.

Action: Attempt two invalid inserts (neither target / both targets), then insert one

valid maintenance record. **After:** The invalid inserts are rejected with the trigger error (Scenario C1), while the valid insert succeeds and is queryable (Scenario C2).

Listing 3.17: Scenario C SQL: Maintenance Insert Tests

```

1 INSERT INTO maintenance (Scheduled_Date, Completion_Date, Status, Description,
    Equipment_ID, Facility_ID)
2 VALUES (DATE_ADD(CURDATE(), INTERVAL 4 DAY), NULL, 'Scheduled', 'Invalid: none',
    NULL, NULL); -- expect error
3
4 INSERT INTO maintenance (Scheduled_Date, Completion_Date, Status, Description,
    Equipment_ID, Facility_ID)
5 VALUES (DATE_ADD(CURDATE(), INTERVAL 4 DAY), NULL, 'Scheduled', 'Invalid: both', 2,
    4); -- expect error
6
7 INSERT INTO maintenance (Scheduled_Date, Completion_Date, Status, Description,
    Equipment_ID, Facility_ID)
8 VALUES (DATE_ADD(CURDATE(), INTERVAL 4 DAY), NULL, 'Scheduled', 'Valid facility
    maintenance', NULL, 4);
9 SELECT 'valid maintenance inserted' AS step, Maintenance_ID, Equipment_ID,
    Facility_ID, Scheduled_Date, Status
10 FROM maintenance ORDER BY Maintenance_ID DESC LIMIT 1;

```

✖ 1650 16:02:54 INSERT INTO maintenance (Scheduled_Date, Completion_D... Error Code: 1644. Maintenance must target either a facility or equipment (exclusively)
 ✖ 1651 16:02:54 INSERT INTO maintenance (Scheduled_Date, Completion_D... Error Code: 1644. Maintenance must target either a facility or equipment (exclusively)

Figure 3.9: Scenario C1: Invalid maintenance keys blocked

step	Maintenance_ID	Equipment_ID	Facility_ID	Scheduled_Date	Status
valid maintenance inserted	3	NULL	4	2025-12-15	Scheduled

Figure 3.10: Scenario C2: Valid maintenance entry inserted

3.2.5 Scenario D: Equipment Availability

Rule: Reserved quantity cannot exceed available equipment stock (total quantity minus overlapping reservations and maintenance). **Before:** Equipment stock is recorded in `equipment.Total_Quantity` and is currently available (Scenario D1). **Action:** Assign a valid quantity to the first reservation (Scenario D2), then attempt to over-allocate for an overlapping reservation (Scenario D3). **After:** The over-allocation is rejected with 'Requested equipment exceeds available quantity' (Scenario D3), preserving inventory correctness.

Listing 3.18: Scenario D1 SQL: Setup and Stock Check

```

1 SET @equip_date := DATE_ADD(CURDATE(), INTERVAL 2 DAY);
2 INSERT INTO reservation (Reservation_Date, Start_Time, End_Time, Facility_ID)

```

```

3 VALUES (@equip_date, '14:00:00', '15:00:00', 8);
4 SET @equip_res_1 := LAST_INSERT_ID();
5 INSERT INTO reservation (Reservation_Date, Start_Time, End_Time, Facility_ID)
6 VALUES (@equip_date, '14:30:00', '15:30:00', 5);
7 SET @equip_res_2 := LAST_INSERT_ID();
8 SELECT 'equipment stock snapshot' AS step, Equipment_ID, Equipment_Name, Status,
      Total_Quantity
9 FROM equipment WHERE Equipment_ID = 2;

```

step	Equipment_ID	Equipment_Name	Status	Total_Quantity
equipment stock snapshot	2	Equipment2	Available	20

Figure 3.11: Scenario D1: Equipment stock snapshot

Valid assign, then over-allocate (should fail):

Listing 3.19: Scenario D2 SQL: Assignment Tests

```

1 INSERT INTO reservation_equipments (Quantity, Equipment_ID, Reservation_ID)
2 VALUES (5, 2, @equip_res_1);
3 SET @equip_row := LAST_INSERT_ID();
4 SELECT 'after valid assign' AS step, id, Equipment_ID, Reservation_ID, Quantity
5 FROM reservation_equipments WHERE id = @equip_row;
6
7 INSERT INTO reservation_equipments (Quantity, Equipment_ID, Reservation_ID)
8 VALUES (16, 2, @equip_res_2); -- expect error

```

step	id	Equipment_ID	Reservation_ID	Quantity
after valid assign	1	2	3	5

Figure 3.12: Scenario D2: Valid equipment assignment

⌚ 2011 16:05:05 INSERT INTO reservation_equipments (Quantity, Equipment_ID, Reservation_ID) Error Code: 1644. Requested equipment exceeds available quantity

Figure 3.13: Scenario D3: Over-allocation blocked

3.2.6 Scenario E: Session Enrollment Capacity

Rule: A training session cannot exceed its `Max_Capacity`. **Before:** A training session is created with `Max_Capacity = 1` (Scenario E1). **Action:** Insert the first enrollment (Scenario E2), then attempt a second enrollment. **After:** The second enrollment is rejected with '`Training session is full`' (Scenario E3), and the session enrollment count remains within capacity.

Listing 3.20: Scenario E SQL: Session Capacity Tests

```

1 SET @session_date := DATE_ADD(CURDATE(), INTERVAL 3 DAY);
2 INSERT INTO reservation (Reservation_Date, Start_Time, End_Time, Facility_ID)
3 VALUES (@session_date, '16:00:00', '17:00:00', 4);
4 SET @session_res_full := LAST_INSERT_ID();
5 INSERT INTO training_session (Reservation_ID, Max_Capacity, Coach_ID)
6 VALUES (@session_res_full, 1, 1);
7 SELECT 'session created' AS step, Reservation_ID, Max_Capacity, Coach_ID
8 FROM training_session WHERE Reservation_ID = @session_res_full;
9
10 INSERT INTO session_enrollment (Member_ID, Reservation_ID)
11 VALUES (1, @session_res_full);
12 SELECT 'after first enrollment' AS step, id, Member_ID, Reservation_ID
13 FROM session_enrollment WHERE Reservation_ID = @session_res_full;
14
15 INSERT INTO session_enrollment (Member_ID, Reservation_ID)
16 VALUES (2, @session_res_full); -- expect error

```

Result Grid				
	step	Reservation_ID	Max_Capacity	Coach_ID
▶	session created	3	1	1

Figure 3.14: Scenario E1: Session creation

Result Grid				
	step	id	Member_ID	Reservation_ID
▶	after first enrollment	1	1	3

Figure 3.15: Scenario E2: First enrollment successful

✖ 2367 16:07:34 INSERT INTO session_enrollment (Member_ID, Reservation_ID) ... Error Code: 1644. Training session is full

Figure 3.16: Scenario E3: Second enrollment blocked due to capacity

Task 4 Access Control

4.1 Objective and Alignment with the Implemented Schema

This task requires the implementation and verification of **two (2)** access control mechanisms, supported by clear evidence (screenshots) demonstrating users' ability (or inability) to *view, update, and delete* data. Our implementation is strictly aligned with the exported database schema in `sports_arena.sql` and the role design previously proposed in the assignment.

In this system, booking-related data is modeled around the following tables (all names and attributes follow `sports\arena.sql`):

- `reservation` stores the reservable time slot (`Reservation_Date`, `Start_Time`, `End_Time`) and the target facility (`Facility_ID`).
- `booking` is a specialization of `reservation` for members, where `Reservation_ID` is both the primary key and a foreign key to `reservation`, and `Booking_Status` captures lifecycle states (e.g., *Pending*, *Confirmed*, *Cancelled*).
- `training_session` is another specialization of `reservation`, linked to `coach` by `Coach_ID` and constrained by `Max_Capacity`.
- `session_enrollment` records member enrollments in training sessions (`Member_ID` + `Reservation_ID`).

Given the multi-role nature of the sports arena, access control must ensure: (i) confidentiality of member-related data, (ii) integrity of bookings and schedules, and (iii) least-privilege operations for each role. To achieve this, we implement **RBAC** (Role-Based Access Control) and **row-level access control (own-row/own-session)** in a defense-in-depth manner (database + Django application layer).

4.2 Mechanism 1: Role-Based Access Control (RBAC)

4.2.1 Role Model and Privilege Boundary

RBAC assigns permissions based on business responsibilities instead of individual identities. The system defines five operational roles consistent with the project implementation (Django `Group`) and the database security model: `dba_role`, `manager_role`, `booking_officer_role`, `coach_role`, and `member_role`.

To keep the evidence concise, the SQL verification screenshots in this task focus on two representative roles (`coach_role` and `member_role`). The remaining roles are included in the privilege boundary matrix to document the full intended least-privilege model.

Table 4.1 summarizes the key **DML** boundaries on the booking-related tables. The principle of *least privilege* is enforced by granting only the minimal operations required to complete legitimate tasks, while explicitly preventing destructive or cross-domain actions (e.g., coaches deleting members).

Role	booking	reservation	training_session	member
dba_role	S/I/U/D	S/I/U/D	S/I/U/D	S/I/U/D
manager_role	S/U	S/U	S	S
booking_officer_role	S/I/U	S/I/U	S	S
coach_role	—	—	S	—
member_role	— (via view)	— (via view)	—	—

Table 4.1: RBAC privilege boundary (*S*=SELECT, *I*=INSERT, *U*=UPDATE, *D*=DELETE). Members are forced to access booking data through restricted views (Mechanism 2), rather than the base tables.

4.2.2 SQL Enforcement via GRANT / REVOKE

At the DBMS layer, we enforce RBAC using MySQL authorization. The following snippet illustrates the core idea: coaches are permitted to read training sessions, while members are denied direct access to sensitive base tables such as `booking`. In our testing, we used dedicated database accounts to represent the `coach_role` and `member_role` sessions.

```

1 -- Example RBAC (simplified for demonstration; aligned to sports_arena.sql table
2   names)
3 CREATE ROLE coach_role;
4 CREATE ROLE member_role;
5
6 -- Coach: can read training session information (e.g., capacity and schedule)
7 GRANT SELECT ON sports_arena.training_session TO coach_role;
8
9 -- Coach: must not manipulate core identity data (member table)
10 REVOKE DELETE ON sports_arena.member FROM coach_role;
11
12 -- Member: cannot directly access the base booking table
13 REVOKE SELECT ON sports_arena.booking FROM member_role;
```

4.2.3 Verification Evidence (SQL)

First, we verified the active permissions to ensure the role configuration is effective. Figure 4.1 displays the `SHOW GRANTS` output, confirming the specific privileges assigned to the user accounts.

Result Grid Filter Rows: <input type="text"/> Export: Wrap Cell Content:	
Grants for root@localhost	
▶ GRANT SELECT, INSERT, UPDATE, DELETE, CREATE, DROP, RELOAD, SHUTDOWN, PROCESS, FILE, REFERENCES, INDEX...	
GRANT APPLICATION_PASSWORD_ADMIN,AUDIT_ABORT_EXEMPT,AUDIT_ADMIN,AUTHENTICATION_POLICY_ADMIN,B...	
GRANT PROXY ON ` ``@`` TO `root` @`localhost` WITH GRANT OPTION	
Root User	
Result Grid Filter Rows: <input type="text"/> Export: Wrap Cell Content:	
Grants for C@localhost	
▶ GRANT USAGE ON *.* TO `C` @`localhost`	
GRANT SELECT ON `sports_arena`.`coach` TO `C` @`localhost`	
GRANT SELECT ON `sports_arena`.`facility` TO `C` @`localhost`	
GRANT SELECT, UPDATE ON `sports_arena`.`training_session` TO `C` @`localhost`	
GRANT `coach_role` @`%` TO `C` @`localhost`	
Coach User	
Result Grid Filter Rows: <input type="text"/> Export: Wrap Cell Content:	
Grants for M@localhost	
▶ GRANT USAGE ON *.* TO `M` @`localhost`	
GRANT SELECT ON `sports_arena`.`member_booking_view` TO `M` @`localhost`	
GRANT `member_role` @`%` TO `M` @`localhost`	
Member User	

Figure 4.1: Permission verification: `SHOW GRANTS` output demonstrating active privileges for `root`, `coach` and `member` roles.

Figure 4.2 demonstrates an allowed operation: a coach account can successfully read rows from `training_session` (showing `Reservation_ID`, `Max_Capacity`, and `Coach_ID`). This supports day-to-day coaching tasks such as checking the sessions they are responsible for.

Result Grid Filter Rows:	
Reservation_ID	Max_Capacity
2	10
NULL	NULL

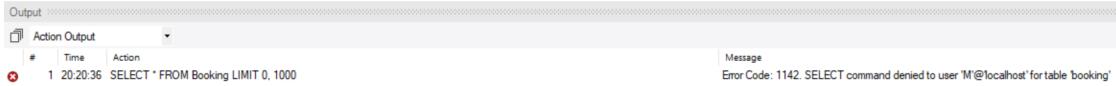
Figure 4.2: RBAC verification (allowed): coach can `SELECT` from `training_session`.

Figure 4.3 demonstrates a blocked operation: the same coach account is explicitly denied from deleting member records. MySQL returns `Error Code: 1142`, which confirms that high-risk actions on identity tables are not available to coaches.

Output		
Action Output	#	Time
1 20:14:42 DELETE FROM Member WHERE Member_ID = 1		
		Message Error Code: 1142. DELETE command denied to user 'C'@'localhost' for table 'member'

Figure 4.3: RBAC verification (denied): coach cannot `DELETE` from `member` (Error Code: 1142).

Similarly, Figure 4.4 shows that a member account is denied direct access to the `booking` base table. This is a critical control, because unrestricted reads on `booking` would expose other members' reservation activities.



The screenshot shows the MySQL Workbench interface with the following details:

- Output** tab selected.
- Action Output** dropdown set to `Time`.
- Query results table:

#	Time	Action
1	20:20:36	SELECT * FROM Booking LIMIT 0, 1000
- Message** panel: Error Code: 1142. SELECT command denied to user 'M'@'localhost' for table 'booking'

Figure 4.4: RBAC verification (denied): member cannot `SELECT` from the `booking` base table (Error Code: 1142).

4.2.4 Django Mapping and User Experience

At the application layer, the same RBAC model is mapped to Django `Group` memberships. Views are protected by role checks (e.g., only booking officers/managers can manage all bookings; members can only access member-facing booking pages). This avoids the common pitfall of “front-end hiding only” by ensuring authorization is enforced server-side.

Figure 4.5 shows a privileged operational view where an administrative role can list and manage bookings across multiple members (the *Member* column is visible and the *Manage* action is available). In contrast, members do not receive this interface and cannot access it directly (see the 403 evidence in Mechanism 2).

Bookings						
ID	Member	Facility	Date	Time	Status	Action
#1	M 1	Facility4	Dec. 11, 2025	12:03 p.m. - 12:04 p.m.	Cancelled	<button>Manage</button>
#3	M 1	Facility4	Dec. 16, 2025	noon - 3 p.m.	Pending	<button>Manage</button>
#4	M 1	Facility5	Dec. 18, 2025	2 p.m. - 5 p.m.	Confirmed	<button>Manage</button>
#5	M 3	Facility5	Dec. 15, 2025	3 p.m. - 4 p.m.	Pending	<button>Manage</button>
#6	M 3	Facility8	Dec. 15, 2025	7 a.m. - 10 a.m.	Pending	<button>Manage</button>

Figure 4.5: Django RBAC evidence: an administrative role can view bookings across members and access management actions.

4.3 Mechanism 2: Row-Level Access Control (Own-Row / Own-Session)

4.3.1 Rationale

RBAC alone is not sufficient when users with the same role should not access each other's records. For example, all members share the `member_role`, yet each member must be restricted to *their own* bookings. Therefore, we implement row-level access control to enforce **data ownership** and **minimal exposure** within the same role.

We enforce row-level control through two complementary techniques:

- **Database layer (view-based filtering):** revoke access to sensitive base tables and grant access only through a restricted view.
- **Application layer (queryset filtering + 403 blocking):** Django filters records by the authenticated user, and blocks any unauthorized access attempts with a 403 response.

4.3.2 SQL View for Member Booking Data

To ensure members cannot bypass row-level filtering, we expose a restricted view that returns only the booking information the member is allowed to see. The view joins `booking`, `reservation`, and `facility` to provide a user-friendly result consistent with the system UI (date and facility name are not stored in `booking` directly).

```

1 -- View-based row-level access control (aligned to sports_arena.sql)
2 -- The WHERE clause represents "current member only" logic.
3 CREATE OR REPLACE VIEW v_member_booking_detail AS
4 SELECT
5   b.Reservation_ID,
6   b.Booking_Status,
7   b.Member_ID,
8   r.Reservation_Date,
9   f.Facility_Name
10 FROM booking b
11 JOIN reservation r ON r.Reservation_ID = b.Reservation_ID
12 JOIN facility f ON f.Facility_ID = r.Facility_ID
13 WHERE b.Member_ID = 1;  -- demo: current member (e.g., Member_ID=1)
14
15 GRANT SELECT ON v_member_booking_detail TO member_role;

```

Figure 4.6 confirms that, after revoking base-table access (Figure 4.4), the member can still retrieve necessary booking details through the view, without exposing other members' data.

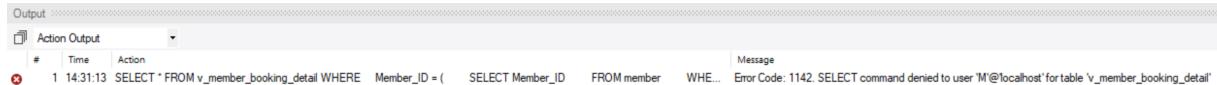
The screenshot shows a database query results grid with the following columns: Reservation_ID, Booking_Status, Member_ID, Reservation_Date, and Facility_Name. The data in the grid is as follows:

	Reservation_ID	Booking_Status	Member_ID	Reservation_Date	Facility_Name
▶	1	Confirmed	1	2025-12-11	Facility4

Figure 4.6: Row-level control (allowed via view): member queries the restricted booking view and receives only permitted rows.

To further validate the effectiveness of the Row-Level Security, we conducted a negative test where Member 1 attempts to explicitly retrieve records belonging to another member

(e.g., Member 2). As demonstrated in Figure 4.7, the result is an empty set (or equivalent denial), confirming that the view successfully filters out unauthorized data even when specifically requested.



```
Output
Action Output
# Time Action
1 14:31:13 SELECT * FROM v_member_booking_detail WHERE Member_ID = (
    SELECT Member_ID
    FROM member
    WHERE Member_ID = 2
)
Message
Error Code: 1142. SELECT command denied to user 'M'@'localhost' for table 'v_member_booking_detail'
```

Figure 4.7: Isolation verification: Member 1 attempts to access Member 2’s booking data through the view, resulting in error (Access Denied).

4.3.3 Django Row-Level Filteringing and Update Boundaries

In the Django application, row-level access control is enforced by filtering the booking list by the logged-in member. Practically, this means:

- A member sees only bookings where `booking.Member_ID` matches the member profile linked to `request.user` (via `member.user_id` in the schema).
- Members are allowed to perform **safe updates** on their own records, such as cancelling their own bookings (implemented as updating `Booking_Status` to *Cancelled* rather than issuing destructive deletes).
- Any attempt to access another role’s page or another member’s record is blocked server-side and returns HTTP 403.

Figure 4.8 shows that **Member 1** can view their own bookings and is provided with the *Cancel* action for eligible items, which demonstrates controlled **UPDATE** capability within an ownership boundary.

Bookings					New Booking
Facility	Date	Time	Status	Action	
Facility4	Dec. 11, 2025	12:03 p.m. - 12:04 p.m.	Cancelled		
Facility4	Dec. 16, 2025	noon - 3 p.m.	Pending	Cancel	
Facility5	Dec. 18, 2025	2 p.m. - 5 p.m.	Confirmed	Cancel	

Figure 4.8: Row-level control (member view): Member 1 can see only their own booking records and perform a bounded update (Cancel).

Figure 4.9 demonstrates the isolation effect: **Member 2** does not see Member 1’s bookings; the page correctly returns “No bookings found.” This is direct evidence that record visibility is constrained by user identity rather than by UI convention.

Bookings					New Booking
Facility	Date	Time	Status	Action	
No bookings found.					

Figure 4.9: Row-level control (member isolation): Member 2 cannot see Member 1's booking records; the list is empty for unrelated data.

Finally, Figure 4.10 shows the system behavior when a user attempts to access a resource outside their authorized role boundary: the request is rejected with a clear 403 page. This verifies that authorization is enforced by the server (not merely by hiding buttons), and that the system provides a secure and user-friendly failure mode.

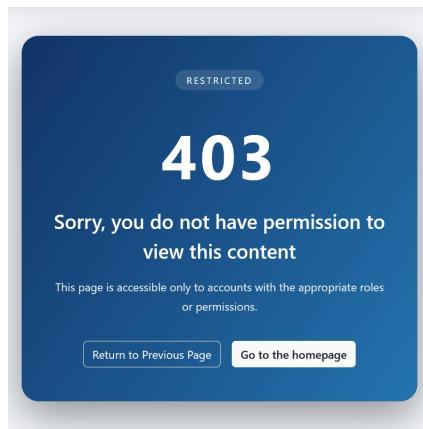


Figure 4.10: Authorization enforcement: unauthorized access is blocked with HTTP 403 and a dedicated error page.

4.4 Summary

This task implements and validates two complementary access control mechanisms:

- **RBAC:** separates privileges across operational roles and prevents unauthorized DML actions (e.g., coach cannot delete members; member cannot directly read the booking base table).
- **Row-level access control:** restricts users to “own data” via view-based filtering (DB) and queryset constraints (Django), and blocks unauthorized access with HTTP 403.

Together, these controls enforce least privilege, prevent data leakage, and protect the integrity of bookings and schedules in a practical, verifiable manner.

Conclusion

This report has detailed the design, implementation, and verification of a robust University Sports Complex Management System. Our solution effectively bridges the gap between theoretical database principles and practical application development.

Summary of Achievements

- **Data Integrity:** We established a solid foundation using Domain, Entity, and Referential integrity constraints. As demonstrated in Task 1, these constraints act as the non-negotiable guardians of data quality, preventing invalid states (e.g., negative inventory or conflicting timestamps) even if the application layer fails.
- **SQL Proficiency vs. ORM:** Task 2 demonstrated the comprehensive implementation of the schema and complex data retrieval needs. While Django handles standard CRUD operations, our raw SQL queries (including Joins, Aggregates, and Subqueries) proved essential for generating advanced analytical reports that go beyond basic object retrieval.
- **Business Logic Enforcement:** In Task 3, we utilized SQL triggers to implement complex rules such as dynamic booking limits (max 2 pending bookings) and cross-table validation (equipment availability). This ensures that business rules are consistent regardless of the access method.
- **Secure Access Control:** Task 4 showcased a defense-in-depth security strategy. By combining Role-Based Access Control (RBAC) with Row-Level Security views, we ensured that users like Coaches and Members operate within strictly defined boundaries, protecting sensitive data while maintaining usability.
- **Application Integration:** The integration with Django demonstrates the synergy between a powerful backend DBMS and a modern web framework. While Django provides a user-friendly interface and preliminary validation (e.g., form checks), the MySQL database serves as the ultimate source of truth and enforcement.

In conclusion, the system meets all specified requirements for data integrity, security, and functionality. The "Database First" design philosophy ensures that the system is reliable, scalable, and maintainable, capable of supporting the dynamic operations of a busy sports complex.

APPENDIX 1
MARKING RUBRICS

Component Title	Lab Report (Grouping)					Percentage (%)	24%
Criteria	Score and Descriptors					Weight	Marks
	Excellent (9.0 – 10.0)	Good (7.0 – 8.5)	Average (5.0 – 6.5)	Need Improvement (3.0 – 4.5)	Poor (0 – 2.5)		
Task 1: Database Integrity (20 Marks)	All CREATE TABLE commands are accurate; all integrity constraints correctly applied. Screenshots complete, clear, and well-organized. Explanations are thorough and show strong understanding.	Most tables and constraints are correctly implemented; minor errors. Screenshots provided and generally clear. Explanations show good understanding but lack depth.	Some tables correctly created; constraints may be incomplete or partially wrong. Screenshots provided. Explanations are basic.	Many constraints incorrect or missing. Screenshots unclear or incomplete. Explanations lack clarity and accuracy.	Incomplete or inaccurate SQL statements. Little or no screenshots. Weak or missing explanations.	20	
Task 2: SQL Implementation (30 Marks)	<ul style="list-style-type: none"> All remaining CREATE TABLE commands are fully correct and clearly shown with screenshots. All tables populated with 6–10 valid rows each. All SQL categories include 3 correct queries each, 	<ul style="list-style-type: none"> CREATE TABLE commands mostly correct with minor errors. Tables filled properly with 6–10 rows each. Most SQL categories include 3 correct queries (some minor mistakes allowed). 	<ul style="list-style-type: none"> Some CREATE TABLE commands incomplete or inaccurate. Tables partially populated (some missing rows or incorrect data). SQL categories include fewer than 3 correct queries or have notable 	<ul style="list-style-type: none"> CREATE TABLE commands contain multiple mistakes or missing screenshots. Many tables not properly populated. Several SQL categories missing or 	<ul style="list-style-type: none"> Missing or incorrect CREATE TABLE commands. Very little or no data inserted. SQL queries missing, incorrect, or not executed. Screenshots mostly missing. 	30	

	<ul style="list-style-type: none"> executed without errors. Screenshots of queries and outputs are complete, clear, and well-organized. Demonstrates excellent understanding of SQL, filtering, joins, arithmetic expressions, and formatting. 	<ul style="list-style-type: none"> Screenshots are provided but may lack full clarity or formatting. Shows good understanding but misses depth in a few areas. 	<ul style="list-style-type: none"> syntax/output issues. Screenshots provided but not consistent or clear. Understanding is present but basic. 	<ul style="list-style-type: none"> incorrectly executed. Screenshots unclear, incomplete, or missing outputs. Limited understanding of SQL syntax and application. 	<ul style="list-style-type: none"> Shows minimal or no understanding of SQL concepts. 	
Task 3: Triggering (15 Marks)	Both triggers fully correct and functional. BEFORE/AFTER results clearly demonstrated with proper screenshots. No syntax errors.	Two triggers implemented with minor mistakes. Evidence provided for BEFORE/AFTER. Screenshots mostly clear.	One trigger corrects, the second partially implemented. Limited or basic BEFORE/AFTER evidence.	Triggers contain multiple errors; screenshots unclear or incomplete. Limited understanding shown.	Triggers missing, non-functional, or incorrect. No valid evidence provided.	15
Task 4: Access Control (15 Marks)	Two access control mechanisms implemented correctly. Permissions tested and verified with clear screenshots. Demonstrates strong security understanding.	Both mechanisms implemented with minor issues. Screenshots provided and mostly clear. Verification mostly correct	One mechanism corrects, the other partially correct. Screenshots basic or incomplete.	Incorrect or incomplete access control commands. Screenshots unclear or missing verification	Access control incorrectly implemented or not attempted. No valid evidence.	15