Task 5 Access Control

5.1 User Roles and Responsibilities

In our implementation, database roles (DBA, MANAGER, BOOKING_OFFICER, COACH, MEMBER) are mapped to application users stored in the `Staff`, `Member`, `Student`, and `External_Visitor` tables. Role-based access control (RBAC) is widely adopted for assigning permissions based on organizational roles. Modern studies, such as Frank, Buhmann, and Basin (2013), further demonstrate how RBAC structures can be systematically derived and optimized from user–permission relationships.A staff member can, for example, be marked as ”manager” or ”front desk” in the application; at login, the corresponding database role is enabled for that user.

Below we describe each logical role and its responsibilities in the sports complex system.

5.1.1 System Administrator (DBA)

The System Administrator is the technical person who manages the database platform.

Main responsibilities:

- Create and disable database accounts, assign roles, and reset passwords.
- Manage the physical database objects (tables, indexes, views), backups, and recovery.
- Perform data correction only when formally requested and authorised by management.

From a security perspective, the DBA can run any SQL command, but in normal operations the DBA focuses on availability, performance, and integrity rather than on day-to-day business data entry or approvals.

5.1.2 Sports Complex Manager

The Sports Complex Manager is responsible for the overall operation of the sports complex and for higher-level business decisions.

Main responsibilities:

- Maintain core reference data such as `Facility` and `Equipment` (e.g. names, capacities, availability status).
- Define and maintain `Training_Session` records, including assigned coach, schedule, capacity, and prerequisites.

- Review maintenance history and approve completion of maintenance jobs.
- Review conflicts between reservations and ensure that important events (e.g. tournaments) are scheduled correctly.
- Approve or reject `Visitor_Application` records for external visitors.

The Manager therefore needs broad read access over the operational data and the ability to update key business objects. However, the Manager generally does not hard-delete business records (such as `Member` or `Facility`) so that history is preserved; instead, status fields (e.g. `Active/Inactive`) are used.

5.1.3 Front Desk / Booking Officer

Front desk staff handle most of the daily interactions with members and visitors.

Main responsibilities:

- Register new members and assist them in updating their profiles.
- Help members make, modify, and cancel bookings and session enrolments (walk-in or phone requests).
- Record maintenance requests on behalf of members or staff, linking them to the relevant facility/equipment.
- Answer enquiries about facility availability, upcoming sessions, and existing bookings.

Operationally, the Booking Officer needs read/write access to day-to-day transaction tables such as `Booking`, `Session_Enrollment`, and `Maintenance`, but cannot change high-level configuration such as facility capacity or approve external visitor applications. That separation reduces the risk of accidental or unauthorised changes to critical data.

5.1.4 Coach

Coaches are staff members responsible for delivering training sessions.

Main responsibilities:

- View their own `Training_Session` schedule and session details.
- View the list of enrolled members for their sessions (`Session_Enrollment`).
- Record attendance, remarks, or simple outcome indicators for participants in their own sessions.

Coaches do not create or delete training sessions. They also cannot see all member information; their access is restricted to members who are enrolled in sessions that they teach.

5.1.5 Member (Student / Staff / External Visitor)

A Member is any person allowed to use the sports complex: internal students, internal staff, and approved external visitors. All three subtypes share the same high-level behaviour with minor differences in attributes.

Main responsibilities:

- Maintain their own personal details (contact information, emergency contact, etc.).
- Browse the list of facilities, equipment, and training sessions.
- Make, modify, and cancel their own bookings, subject to business rules (time limits, clashes, etc.).
- Enrol in and withdraw from training sessions, again limited to their own records.

Members never directly modify data that belongs to other members or to the sports complex configuration. Access is restricted to "own" data (for example, own bookings and enrolments) using row-level security.

5.2 Access Types for Each Table

We classify access as:

- **R** – Read (SELECT)
- **I** – Insert (CREATE new row)
- **U** – Update (modify existing row)
- **D** – Delete

A star `*` indicates that the access is limited to the user's own rows, enforced by predicates on `Member_ID` (or the equivalent foreign key). A double star `**` indicates access restricted to rows related to the coach's own sessions (for example, enrolments in sessions they teach).

The correctness of such permission assignments is essential, and recent work has shown that access-control policies can be formally analyzed and verified to avoid misconfigurations (Gouglidis, Kagia, & Hu, 2023).

5.2.1 Identity and Contact Tables

These tables store member and coach identities and their contact details.

Table	DBA	Manager	Booking Officer	Coach	Member
Member	R/I/U/D	R/I/U	R/I/U	R	R/U*
Student	R/I/U/D	R/I/U	R/I/U	R	R/U*
Staff	R/I/U/D	R/I/U	R/I/U	R	R/U*
External_Visitor	R/I/U/D	R/I/U	R/I/U	R	R/U*
Coach	R/I/U/D	R/I/U	R	R/I/U*	R
Member_Phone	R/I/U/D	R/I/U	R/I/U	R	R/I/U*
Member_Email	R/I/U/D	R/I/U	R/I/U	R	R/I/U*
Coach_Phone	R/I/U/D	R/I/U	R	R/I/U*	R
Coach_Email	R/I/U/D	R/I/U	R	R/I/U*	R

Rationale:

- The DBA has full control for administrative and recovery purposes.
- The Manager can maintain staff and member data but avoids hard deletes in normal practice.
- Booking Officers need to create and update member records when assisting people at the front desk. They have unrestricted read, insert, and update access to member-related tables because their primary job is to help *any* member with registration, profile updates, and booking management. Application-level logic (e.g., audit logs, transaction context) ensures that these operations are performed only in legitimate service scenarios.
- Each Member can read and update only their own profile and contact details, enforced via predicates such as `Member_ID = CURRENT_USER` (or an equivalent context mechanism).
- Coaches maintain only their own identity and contact details in these tables. When they need to contact participants, the application exposes filtered views that show minimal member contact information only for participants enrolled in their sessions, not for all members in the system.

Row-level security for the * and ** restrictions will be implemented using views or fine-grained access predicates (see example below).

5.2.2 Facility, Equipment, and Maintenance Tables

Table	DBA	Manager	Booking Officer	Coach	Member
Facility	R/I/U/D	R/I/U	R	R	R
Equipment	R/I/U/D	R/I/U	R	R	R
Maintenance	R/I/U/D	R/I/U	R/I/U	R	R

Rationale:

- `Facility` and `Equipment` define the core resources of the sports complex. They are therefore maintained by the Manager; other roles have read-only access.
- `Maintenance` records may be created either by the Manager or by the Booking Officer when a member reports a problem. In practice, application logic or column-level controls ensure that only the manager can set a job to completed (e.g. `Status = 'Completed'` and `Completion_Date`), while Booking Officers can update descriptive fields such as problem description or notes.
- Members and Coaches can read maintenance information to understand why a resource is unavailable, but cannot create or alter maintenance records.

5.2.3 Reservation, Booking, and Training Tables

The conceptual model treats `Reservation` as a supertype of `Booking` and `Training_Session`. In practice, end users mainly interact with higher-level entities through application views; direct access to the base `Reservation` table is mostly required for administrative purposes.

Table	DBA	Manager	Booking Officer	Coach	Member
Reservation	R/I/U/D	R/U	R/I/U	R**	R*
Booking	R/I/U/D	R/U	R/I/U	R**	R/I/U*
Training_Session	R/I/U/D	R/I/U/D	R	R/U**	R
Reservation_Equipments	R/I/U/D	R/I/U	R/I/U	R**	R*
Session_Enrollment	R/I/U/D	R/U	R/I/U	R/U**	R/I/U*

Rationale:

- **Member**:
  - Can create, update, and cancel their own `Booking` and `Session_Enrollment` records (*).

  - Has read-only access to their underlying `Reservation` records, normally via views rather than direct table access.
  - Can view the equipment associated with their own reservations via application views over `Reservation_Equipments`, but cannot directly insert, update, or delete rows in that table.

- **Booking Officer**:
  - Handles creation, modification, and cancellation of `Booking` and `Session_Enrollment` for any member.
  - Can also create or adjust `Reservation` and `Reservation_Equipments` entries to support bookings and training sessions.

- **Manager**:
  - Oversees overall scheduling and therefore needs read and update rights on `Reservation`, `Booking`, and `Session_Enrollment` to resolve conflicts or correct errors.
  - Has full control over `Training_Session` (create, update, and, in exceptional cases, delete), including assigning coaches and setting capacities.

- **Coach**:
  - Can read all relevant reservation and booking details for their own sessions only (**).
  - Can update their own `Training_Session` details where permitted (e.g. remarks, minor time adjustments approved by the Manager) and record attendance or notes in `Session_Enrollment` for participants in their sessions.

In an actual DBMS implementation, most of the member-facing operations would be performed through views such as `Member_Booking` and `Member_Session_Enrollment`, which already filter rows by `Member_ID` and prevent access to other members' data.

5.2.4 Visitor Application Management

Table	DBA	Manager	Booking Officer	Coach	Member
Visitor_Application	R/I/U/D	R/I/U	R/I	R	–
Visitor_Application_Phone	R/I/U/D	R/I/U	R/I	R	–
Visitor_Application_Email	R/I/U/D	R/I/U	R/I	R	–

Rationale:

- Prospective visitors (not yet members) submit applications, typically through the front desk or an online form. Booking Officers can create and insert `Visitor_Application` records on behalf of walk-in applicants.
- The Manager reviews applications, updates their status (e.g. `Pending`, `Approved`, `Rejected`), and records approval details. Upon approval, the Manager creates corresponding `Member` and `External_Visitor` records and updates the `Created_Member_ID` field for traceability.
- Members do not have access to `Visitor_Application` tables because these records represent people who are not yet in the system. Once an application is approved and a `Member` record is created, the applicant gains member privileges.
- Hard deletes are avoided in normal operations to keep an audit trail of all applications.

5.3 Example SQL Implementation (Roles, Grants, and Views)

Below is a simplified example of how this access control model could be implemented in SQL. Exact syntax may vary between DBMSs.

```sql
-- Create roles
CREATE ROLE dba_role;              -- mapped to the real DBA account
CREATE ROLE manager_role;          -- sports complex manager
CREATE ROLE booking_officer_role;  -- front desk staff
CREATE ROLE coach_role;            -- coaches
CREATE ROLE member_role;           -- regular members

-- Example: grant privileges on Booking
GRANT SELECT, INSERT, UPDATE, DELETE ON Booking TO dba_role;
GRANT SELECT, UPDATE ON Booking TO manager_role;
GRANT SELECT, INSERT, UPDATE ON Booking TO booking_officer_role;

-- Member should only see and modify their own bookings.
-- We create a view that filters on MEMBER_ID and only expose the view to the
   member_role.

CREATE VIEW Member_Booking AS
SELECT b.*
FROM Booking b
JOIN Member m ON b.Member_ID = m.Member_ID
WHERE m.Member_ID = CURRENT_USER;  -- or another context function depending on DBMS

GRANT SELECT, INSERT, UPDATE ON Member_Booking TO member_role;
```

```sql
REVOKE ALL ON Booking FROM member_role;

-- Similarly, we can define a view for coaches to see enrolments only in their own
  sessions.

CREATE VIEW Coach_Session_Enrollment AS
SELECT se.*
FROM Session_Enrollment se
JOIN Training_Session ts ON se.Reservation_ID = ts.Reservation_ID
JOIN Coach c ON ts.Coach_ID = c.Coach_ID
WHERE c.Coach_ID = CURRENT_USER;  -- or another context function depending on DBMS

GRANT SELECT, UPDATE ON Coach_Session_Enrollment TO coach_role;
REVOKE ALL ON Session_Enrollment FROM coach_role;
```

In a real deployment we would also use schemas, additional views, or fine-grained access control mechanisms (depending on the DBMS) to enforce the * and ** row-level restrictions consistently across all tables.

5.4 Security Principles Demonstrated

This design implements several core data security principles:

1. **Least privilege**
   Each role receives only the permissions necessary for its duties. For example, members can modify only their own bookings and profiles; coaches can see only the enrolments for sessions they teach; booking officers cannot change facility capacity or approve applications.

2. **Separation of duties**
   Sensitive actions such as approving external visitor registrations or marking maintenance as completed are reserved for the Manager. Front desk staff can collect and record data but cannot finalise approvals or override important business rules.

3. **Data privacy**
   Personal data is protected via row-level restrictions and dedicated views. Members see only their own identity, contact details, bookings, and applications. Coaches see limited information about participants in their sessions, and not about all members in the system.

4. **Auditability and integrity**
   Instead of deleting key business records, we generally mark them as cancelled, inactive, or completed using status attributes. Physical deletion is reserved for the DBA in exceptional cases (e.g. test data or legal requirements). This keeps a reliable

audit trail of all important operations (bookings, maintenance, applications, etc.).

Overall, this role-based access control (RBAC) design is consistent with our EERD and supports the daily workflow of the sports complex while providing a reasonable level of security, privacy, and maintainability.

Task 6 Database Integrity

This section explains how entity integrity, referential integrity, and domain integrity are enforced in our logical design. We link each type of integrity to the tables and attributes generated from the EERD.

6.1 Entity Integrity

Entity integrity ensures that every row in a table can be uniquely identified and that primary key attributes are never `NULL`.

6.1.1 Strong Entities and Primary Keys

Each strong entity in the EERD becomes a base table with a single, non-null primary key:

- `Member(Member_ID, ...)` with `Member_ID INT PRIMARY KEY NOT NULL`
- `Coach(Coach_ID, ...)` with `Coach_ID INT PRIMARY KEY NOT NULL`
- `Facility(Facility_ID, ...)` with `Facility_ID INT PRIMARY KEY NOT NULL`
- `Equipment(Equipment_ID, ...)` with `Equipment_ID INT PRIMARY KEY NOT NULL`
- `Reservation(Reservation_ID, ...)` with `Reservation_ID INT PRIMARY KEY NOT NULL`
- `Maintenance(Maintenance_ID, ...)` with `Maintenance_ID INT PRIMARY KEY NOT NULL`
- `Visitor_Application(Application_ID, ...)` with `Application_ID INT PRIMARY KEY NOT NULL`

These primary keys uniquely identify each row and are defined as `NOT NULL`, so no row can exist without an identifier.

6.1.2 Supertype–Subtype Structure and Key Inheritance

`Member` is a supertype with three disjoint subtypes: `Student`, `Staff`, and `External_-Visitor`. Each subtype table uses the same primary key as `Member` to preserve the 1:1 relationship between a member and its subtype specialization:

```sql
CREATE TABLE Member (
    Member_ID INT PRIMARY KEY,
    Membership_Status VARCHAR(20) NOT NULL,
    -- other attributes...
);

CREATE TABLE Student (
    Member_ID  INT PRIMARY KEY,
    Student_ID VARCHAR(20) NOT NULL UNIQUE,
    -- other attributes...
    FOREIGN KEY (Member_ID) REFERENCES Member(Member_ID)
        ON DELETE CASCADE
);

CREATE TABLE Staff (
    Member_ID INT PRIMARY KEY,
    Staff_ID  VARCHAR(20) NOT NULL UNIQUE,
    -- other attributes...
    FOREIGN KEY (Member_ID) REFERENCES Member(Member_ID)
        ON DELETE CASCADE
);

CREATE TABLE External_Visitor (
    Member_ID  INT PRIMARY KEY,
    IC_Number  VARCHAR(30) NOT NULL UNIQUE,
    -- other attributes...
    FOREIGN KEY (Member_ID) REFERENCES Member(Member_ID)
        ON DELETE CASCADE
);
```

Here, entity integrity is maintained because:

- Each subtype row has exactly one `Member_ID` that is non-null and unique.
- `Student_ID`, `Staff_ID`, and `IC_Number` are declared `UNIQUE`, so they also behave as candidate keys within their respective tables, preventing two students from sharing the same student ID, etc.

6.1.3 Weak Entities and Composite Primary Keys

Some tables represent relationship entities that depend on other entities and use composite primary keys to ensure uniqueness:

```sql
CREATE TABLE Reservation_Equipments (
    Reservation_ID INT NOT NULL,
    Equipment_ID   INT NOT NULL,
    Quantity       INT NOT NULL,
    PRIMARY KEY (Reservation_ID, Equipment_ID)
);

CREATE TABLE Session_Enrollment (
    Reservation_ID INT NOT NULL,
    Member_ID      INT NOT NULL,
    PRIMARY KEY (Reservation_ID, Member_ID)
);

CREATE TABLE Member_Phone (
    Member_ID     INT NOT NULL,
    Phone_Number  VARCHAR(20) NOT NULL,
    PRIMARY KEY (Member_ID, Phone_Number)
);
```

- In `Reservation_Equipments`, the composite primary key prevents the same equipment from being listed twice for the same reservation.
- In `Session_Enrollment`, it prevents the same member from being enrolled multiple times in the same training session.
- In `Member_Phone`, it prevents duplicate phone numbers for the same member.

In all cases, the composite key columns are defined as `NOT NULL`, maintaining entity integrity for these relationship entities.

6.2 Referential Integrity

Referential integrity ensures that foreign key values always refer to existing rows in the parent table and that deletions/updates do not create orphan records. The formal reasoning behind referential integrity and its implications across relational schemas has been extensively analyzed in recent work (Kenig & Suciu, 2022).

6.2.1 Supertype–Subtype Foreign Keys

Each subtype references `Member` using a foreign key with cascading delete:

```sql
ALTER TABLE Student
ADD CONSTRAINT fk_student_member
FOREIGN KEY (Member_ID)
REFERENCES Member(Member_ID)
ON DELETE CASCADE;

ALTER TABLE Staff
ADD CONSTRAINT fk_staff_member
FOREIGN KEY (Member_ID)
REFERENCES Member(Member_ID)
ON DELETE CASCADE;

ALTER TABLE External_Visitor
ADD CONSTRAINT fk_external_member
FOREIGN KEY (Member_ID)
REFERENCES Member(Member_ID)
ON DELETE CASCADE;
```

- A row in `Student`, `Staff`, or `External_Visitor` cannot exist without a corresponding `Member`.
- When a `Member` is deleted, the related subtype row is automatically removed to avoid orphaned subtype records.

6.2.2 Reservations, Bookings, and Members/Facilities

In the logical design, `Reservation` is a supertype for specific reservation types such as `Booking` and `Training_Session`. Normal facility bookings are stored in `Booking` and linked back to members and facilities as follows:

```sql
CREATE TABLE Reservation (
    Reservation_ID   INT PRIMARY KEY,
    Facility_ID      INT NOT NULL,
    Reservation_Date DATE NOT NULL,
    Start_Time       TIME NOT NULL,
    End_Time         TIME NOT NULL,
    FOREIGN KEY (Facility_ID) REFERENCES Facility(Facility_ID)
);

CREATE TABLE Booking (
    Reservation_ID  INT PRIMARY KEY,
    Member_ID       INT NOT NULL,
    Booking_Status  VARCHAR(20) NOT NULL,
    FOREIGN KEY (Reservation_ID) REFERENCES Reservation(Reservation_ID)
        ON DELETE CASCADE,
    FOREIGN KEY (Member_ID) REFERENCES Member(Member_ID)
);
```

This ensures that:

- Every booking row corresponds to exactly one generic reservation, which already includes the facility and time slot information.
- A booking cannot reference a non-existing member.
- The facility association is managed at the `Reservation` level, ensuring that both `Booking` and `Training_Session` (which also inherits from `Reservation`) can utilize facilities consistently.
- Deleting a reservation cascades to delete its associated booking, preventing orphan bookings.

6.2.3 Training Sessions and Coaches

Training sessions are another specialization of `Reservation` and are taught by exactly one coach:

```
CREATE TABLE Training_Session (
    Reservation_ID INT PRIMARY KEY,
    Coach_ID       INT NOT NULL,
    Max_Capacity   INT NOT NULL,
    FOREIGN KEY (Reservation_ID) REFERENCES Reservation(Reservation_ID)
        ON DELETE CASCADE,
    FOREIGN KEY (Coach_ID) REFERENCES Coach(Coach_ID)
        ON DELETE RESTRICT
);
```

Referential integrity here ensures that:

- A training session cannot exist unless its base `Reservation` exists.
- Each session must be linked to a valid `Coach`.
- The `ON DELETE RESTRICT` on `Coach` prevents deleting a coach while there are still sessions assigned to them, unless handled explicitly by the application or by reassigning the sessions.

6.2.4 Equipment Assignments to Reservations

The `Reservation_Equipments` table connects reservations and equipment:

```
ALTER TABLE Reservation_Equipments
```

```sql
ADD CONSTRAINT fk_res_eq_reservation
FOREIGN KEY (Reservation_ID)
REFERENCES Reservation(Reservation_ID)
ON DELETE CASCADE;

ALTER TABLE Reservation_Equipments
ADD CONSTRAINT fk_res_eq_equipment
FOREIGN KEY (Equipment_ID)
REFERENCES Equipment(Equipment_ID)
ON DELETE RESTRICT;
```

- A row in `Reservation_Equipments` cannot exist without both a valid `Reservation` and a valid `Equipment`.
- If a reservation is deleted, its assigned equipment rows are also deleted.

6.2.5 Session Enrollments

`Session_Enrollment` links members to training sessions:

```sql
ALTER TABLE Session_Enrollment
ADD CONSTRAINT fk_enrollment_reservation
FOREIGN KEY (Reservation_ID)
REFERENCES Training_Session(Reservation_ID)
ON DELETE CASCADE;

ALTER TABLE Session_Enrollment
ADD CONSTRAINT fk_enrollment_member
FOREIGN KEY (Member_ID)
REFERENCES Member(Member_ID)
ON DELETE RESTRICT;
```

This guarantees that:

- Enrollments only exist for valid training sessions and members.
- When a training session is deleted, related enrollments are removed automatically.

6.2.6 Maintenance and Maintained Resources

Each maintenance record applies either to a facility or to an equipment item. We enforce referential integrity via optional foreign keys:

```sql
CREATE TABLE Maintenance (
    Maintenance_ID INT PRIMARY KEY,
    Equipment_ID   INT NULL,
    Facility_ID    INT NULL,
    Status          VARCHAR(20) NOT NULL,
    -- other attributes...
    -- Note: Each record represents maintenance of ONE equipment instance or
      facility.
    -- Multiple equipment items require multiple maintenance records.
    FOREIGN KEY (Equipment_ID) REFERENCES Equipment(Equipment_ID)
        ON DELETE RESTRICT,
    FOREIGN KEY (Facility_ID) REFERENCES Facility(Facility_ID)
        ON DELETE RESTRICT,
    CHECK (
        (Equipment_ID IS NOT NULL AND Facility_ID IS NULL) OR
        (Equipment_ID IS NULL AND Facility_ID IS NOT NULL)
    )
);
```

- A maintenance record must point to exactly one existing equipment or facility.
- The XOR `CHECK` constraint (discussed again in domain integrity) guarantees that both foreign keys cannot be set simultaneously or both be `NULL`.

6.2.7 Visitor Applications

`Visitor_Application` records are created by prospective visitors who are not yet members of the system:

```sql
CREATE TABLE Visitor_Application (
    Application_ID   INT PRIMARY KEY,
    First_Name       VARCHAR(50) NOT NULL,
    Last_Name        VARCHAR(50) NOT NULL,
    IC_Number        VARCHAR(20) NOT NULL UNIQUE,
    Application_Date DATE NOT NULL,
    Status           VARCHAR(20) NOT NULL,
    Approved_By      INT NULL,
    Approval_Date    DATE NULL,
    Reject_Reason    TEXT NULL,
    Created_Member_ID INT NULL,
    FOREIGN KEY (Approved_By) REFERENCES Staff(Member_ID)
        ON DELETE SET NULL,
    FOREIGN KEY (Created_Member_ID) REFERENCES Member(Member_ID)
        ON DELETE SET NULL
);
```

- The applicant's basic information (`First_Name`, `Last_Name`, `IC_Number`) is stored directly in this table. At the time of application submission, the applicant is not yet a `Member`.

- `IC_Number` is marked as `UNIQUE` to prevent duplicate applications from the same person.
- The `Approved_By` field references `Staff` to track which staff member processed the application. If the staff member is deleted, `Approved_By` is set to `NULL` to preserve the application history.
- `Created_Member_ID` links to the `Member` record created after approval. For pending or rejected applications, this field is `NULL`. Upon approval, the staff creates a new `Member` and `External_Visitor` record, then updates this field to establish the traceability link. If the member record is later deleted, this field is set to `NULL`, but the application record is preserved for audit purposes.

6.2.8 Visitor Application Contact Details

The multi-valued contact information is stored in separate weak entity tables:

```sql
CREATE TABLE Visitor_Application_Phone (
    Application_ID INT NOT NULL,
    Phone_Number   VARCHAR(20) NOT NULL,
    PRIMARY KEY (Application_ID, Phone_Number),
    FOREIGN KEY (Application_ID) REFERENCES Visitor_Application(Application_ID)
        ON DELETE CASCADE
);

CREATE TABLE Visitor_Application_Email (
    Application_ID  INT NOT NULL,
    Email_Address   VARCHAR(100) NOT NULL,
    PRIMARY KEY (Application_ID, Email_Address),
    FOREIGN KEY (Application_ID) REFERENCES Visitor_Application(Application_ID)
        ON DELETE CASCADE
);
```

These tables allow each application to have multiple phone numbers and email addresses. The composite primary keys prevent duplicate entries for the same application. When an application is deleted, its contact details are automatically removed via `ON DELETE CASCADE`.

6.2.9 Member Phone Numbers

Finally, `Member_Phone` enforces referential integrity to `Member`:

```sql
ALTER TABLE Member_Phone
ADD CONSTRAINT fk_phone_member
FOREIGN KEY (Member_ID)
```

```sql
REFERENCES Member(Member_ID)
ON DELETE CASCADE;
```

This prevents phone numbers from being associated with non-existing members and automatically cleans up phone records when a member is deleted.

6.3 Domain Integrity

Domain integrity restricts the values that can be stored in each column through data types, length limits, and `CHECK` constraints.

6.3.1 Data Types and Length Constraints

We choose SQL data types that reflect the meaning of each attribute:

- `DATE`, `TIME`, `DATETIME` for temporal attributes such as reservation start/end time and maintenance dates.
- `INT` for identifiers, quantities, and capacities.
- `VARCHAR(n)` for names, phone numbers, and descriptive fields, with reasonable length limits (e.g., `VARCHAR(50)` for names, `VARCHAR(20)` for phone numbers) to prevent excessively long inputs.

Example:

```sql
CREATE TABLE Facility (
    Facility_ID   INT PRIMARY KEY,
    Facility_Name VARCHAR(100) NOT NULL,
    Capacity      INT NOT NULL,
    Status        VARCHAR(20) NOT NULL
);
```

6.3.2 Enumerated Status Values

Several attributes are restricted to predefined sets of values using `CHECK` constraints:

```sql
ALTER TABLE Member
ADD CONSTRAINT chk_member_status
CHECK (Membership_Status IN ('Active', 'Inactive', 'Pending_Approval'));

ALTER TABLE Booking
ADD CONSTRAINT chk_booking_status
CHECK (Booking_Status IN ('Pending', 'Confirmed', 'Cancelled'));

ALTER TABLE Facility
```

```sql
ADD CONSTRAINT chk_facility_status
CHECK (Status IN ('Available', 'Under_Maintenance', 'Unavailable'));

ALTER TABLE Equipment
ADD CONSTRAINT chk_equipment_status
CHECK (Status IN ('Available', 'In_Use', 'Under_Maintenance', 'Damaged'));

ALTER TABLE Maintenance
ADD CONSTRAINT chk_maintenance_status
CHECK (Status IN ('Scheduled', 'In_Progress', 'Completed', 'Cancelled'));

ALTER TABLE Visitor_Application
ADD CONSTRAINT chk_application_status
CHECK (Status IN ('Pending', 'Approved', 'Rejected'));
```

These constraints ensure that status columns only store meaningful, valid values consistent with the business rules.

6.3.3 Numeric ranges and inventory consistency

Numeric attributes are constrained to realistic ranges:

```sql
ALTER TABLE Facility
ADD CONSTRAINT chk_facility_capacity
CHECK (Capacity > 0);

ALTER TABLE Equipment
ADD CONSTRAINT chk_equipment_quantity
CHECK (Total_Quantity >= 0);

-- Note: Available_Quantity is a derived attribute (not stored in the table),
-- so no constraint is defined for it. It is calculated dynamically as:
-- Total_Quantity - (quantity in active reservations + count of active maintenance
  records)

ALTER TABLE Reservation_Equipments
ADD CONSTRAINT chk_res_eq_quantity
CHECK (Quantity > 0);
```

This prevents negative capacities or quantities, and ensures that available equipment does not exceed the total quantity.

6.3.4 Temporal Consistency

We enforce logical ordering of time-related attributes. For example, a reservation must end after it starts:

```sql
ALTER TABLE Reservation
ADD CONSTRAINT chk_reservation_time
CHECK (Start_Time < End_Time);
```

Optionally, we may also restrict new reservations so that they are not created in the past:

```sql
-- Optional constraint, depending on operational policy
-- Note: Since we use DATE + TIME separation, we check the date component
ALTER TABLE Reservation
ADD CONSTRAINT chk_reservation_date_not_past
CHECK (Reservation_Date >= CURDATE());

-- For same-day reservations, time validation would need to be done
-- at the application layer, as:
-- CHECK ((Reservation_Date > CURDATE()) OR
--         (Reservation_Date = CURDATE() AND Start_Time >= CURTIME()))
-- This complex check is better handled in application logic
```

6.3.5 Business Rule Constraints

Based on the assumptions defined in section 1.5, we enforce the following business rules at the database level where feasible:

Maximum Booking Duration (3 hours) Each member's single booking session cannot exceed three hours. This rule can be enforced directly using a `CHECK` constraint on the `Reservation` table:

```sql
ALTER TABLE Reservation
ADD CONSTRAINT chk_reservation_max_duration
CHECK (TIMESTAMPDIFF(HOUR,
    CONCAT(Reservation_Date, ' ', Start_Time),
    CONCAT(Reservation_Date, ' ', End_Time)
) <= 3);
```

This constraint calculates the time difference between start and end times and ensures it does not exceed 3 hours.

Maximum Advance Booking (1 week) Members may book facilities up to one week in advance. This constraint prevents reservations from being made too far in the future:

```sql
ALTER TABLE Reservation
ADD CONSTRAINT chk_reservation_max_advance
CHECK (Reservation_Date <= DATE_ADD(CURDATE(), INTERVAL 7 DAY));
```

Maximum Pending Bookings per Member The rule ”each member may have at most 2 pending bookings at any given time” requires counting rows across the `Booking` table filtered by `Member_ID` and `Booking_Status = 'Pending'`. Since MySQL CHECK constraints do not support subqueries, this rule **must be enforced at the application layer** or via a `BEFORE INSERT` trigger.

Application-level enforcement example:

```sql
-- Before inserting a new Booking, the application should execute:
SELECT COUNT(*)
FROM Booking
WHERE Member_ID = ? AND Booking_Status = 'Pending';

-- If count >= 2, reject the new booking request
```

Alternatively, a database trigger can enforce this rule:

```sql
DELIMITER $$
CREATE TRIGGER trg_check_pending_bookings
BEFORE INSERT ON Booking
FOR EACH ROW
BEGIN
    DECLARE pending_count INT;

    SELECT COUNT(*) INTO pending_count
    FROM Booking
    WHERE Member_ID = NEW.Member_ID
      AND Booking_Status = 'Pending';

    IF pending_count >= 2 THEN
        SIGNAL SQLSTATE '45000'
        SET MESSAGE_TEXT = 'Member already has 2 pending bookings';
    END IF;
END$$
DELIMITER ;
```

This separation of concerns—simple constraints at the database level, complex counting logic at the application or trigger level—ensures that the system remains maintainable while enforcing all necessary business rules.

In practice, this constraint would typically be enforced at the application layer or only for newly created records, to allow importing historical data if needed.

6.3.6 XOR Constraint for Maintenance

As shown in Section 2.6, we use a `CHECK` constraint in `Maintenance` to ensure that each maintenance record refers to exactly one type of resource (either an equipment item or a facility):

```sql
ALTER TABLE Maintenance
ADD CONSTRAINT chk_maintenance_target
CHECK (
    (Equipment_ID IS NOT NULL AND Facility_ID IS NULL) OR
    (Equipment_ID IS NULL AND Facility_ID IS NOT NULL)
);
```

This enforces the business rule that a maintenance task cannot simultaneously target both a facility and an equipment item, and cannot be left without a target.

6.3.7 Multi-Valued Attributes

The multi-valued attribute `Phone_Number` of `Member` is transformed into a separate table `Member_Phone`. Domain integrity is enforced by:

- Choosing `VARCHAR(20)` for `Phone_Number` to allow different phone formats.
- Using the composite primary key `(Member_ID, Phone_Number)` to avoid duplicates.
- Applying a simple pattern check if needed (implementation-dependent), e.g.:

```sql
-- Example pattern check (syntax depends on the SQL dialect)
ALTER TABLE Member_Phone
ADD CONSTRAINT chk_phone_format
CHECK (Phone_Number <> '');  -- or a more specific pattern using LIKE/REGEXP
```

6.4 Summary

Our design enforces:

- **Entity integrity** by defining appropriate primary keys for all strong and weak entities, using non-null keys and `UNIQUE` constraints for natural candidate keys such as `Student_ID`, `Staff_ID`, and `IC_Number`.
- **Referential integrity** by specifying foreign keys that accurately reflect the relationships in the EERD (including supertype–subtype links, bookings and training sessions derived from reservations, maintenance targets, and enrollment/assignment tables), with suitable actions on delete.

- **Domain integrity** by carefully selecting data types and lengths and by using `CHECK` constraints on enumerated statuses, numeric ranges, time ordering, and XOR conditions.

Together, these constraints ensure that the database state remains consistent with the business rules of the sports facility management system.

Task 7 Database Transaction

For Task 7, we analyze how our **actual system design** (EERD, relational schema, constraints, and access control) works together with the DBMS transaction mechanism to maintain all four ACID properties for key operations such as creating bookings, borrowing equipment, enrolling in training sessions, scheduling maintenance, and approving registrations.

7.1 Atomicity

Atomicity means that a business operation that consists of several database statements is treated as one indivisible transaction: it either completes fully or is entirely rolled back.

In our system, we treat each high-level business action as a single transaction. For example, when a member books a court and borrows equipment at the same time, the following steps logically belong to one transaction:

- Insert a new row into `Reservation` (supertype) and its subtype `Booking`.
- Insert one or more rows into `Reservation_Equipments` to record which equipment is borrowed and in what quantity.
- Rely on the derived `Available_Quantity` (calculated from `Equipment.Total_Quantity`, `Reservation_Equipments`, and active `Maintenance` records) when checking availability, rather than updating a stored column.

In SQL-style pseudocode, this is wrapped in a transaction block:

```sql
START TRANSACTION;
  -- insert Reservation and Booking
  -- insert Reservation_Equipments rows
  -- (Available_Quantity is a derived attribute, no update needed)
COMMIT;
```

If any step fails (for example, a foreign key violation, a `CHECK` constraint failure, or an application-level validation error), the application will execute `ROLLBACK`, causing all inserts and updates in this block to be undone. Thus, we never end up in a state where the booking exists but the equipment allocation was not recorded, or vice versa. The same pattern applies to other multi-step actions such as cancelling a booking (update booking status and delete `Reservation_Equipments` records, which automatically frees equipment) or approving a `Visitor_Application` and creating the corresponding `Member` and subtype row (`Student`, `Staff`, or `External_Visitor`).

7.2 Consistency

Consistency means that every committed transaction takes the database from one valid state to another, where all constraints and business rules are satisfied. In our design, consistency is mainly enforced through the integrity constraints defined in Task 6, combined with transactional execution.

Key mechanisms include:

- **Entity integrity:** Primary keys on tables such as `Member`, `Coach`, `Facility`, `Equipment`, `Reservation`, and `Maintenance` ensure that no duplicate or `NULL` identifiers are inserted.
- **Referential integrity:** Foreign keys (for example, from `Booking` to `Reservation`, from `Reservation_Equipments` to both `Reservation` and `Equipment`, from subtypes to `Member`) ensure that a transaction cannot create orphan records. If a booking references a non-existent member or facility, the insert fails and the transaction is rolled back.
- **Domain integrity:** `CHECK` constraints such as `Start_Time < End_Time`, `Capacity > 0`, and the status enumerations (e.g. `Booking_Status IN ('Pending', 'Confirmed', 'Cancelled')`) prevent invalid attribute values from being committed.
- **Business rules on conflicts:** For facilities, we use indexed time fields (`Reservation(Facility_ID, Date, Start_Time, End_Time)` and `Maintenance(Facility_ID, Scheduled_Date)`) together with conflict-checking logic (e.g. a stored procedure that checks for overlapping time ranges, or a view such as `v_facility_occupancy`). If a new reservation would overlap with an existing booking or maintenance slot, the procedure signals an error and the transaction is not committed.

Because all these constraints are checked within the transaction, any attempt to violate a rule causes that transaction to fail and roll back. As a result, the database is always left in a state that is consistent with the conceptual EERD and the real-world rules of the sports complex.

7.3 Isolation

Isolation ensures that concurrent transactions do not interfere with each other in a way that produces incorrect results. Each transaction should behave as if it is running alone, even when many users are booking facilities or borrowing equipment at the same time.

Consider the case where there are 5 rackets available. User A tries to borrow 4, and at nearly the same time User B tries to borrow 3. If these operations are not properly isolated, both might read "5 available" and we could end up with `Available_Quantity = -2`. To avoid this, we rely on the DBMS's isolation mechanism (for example, the default `READ COMMITTED` isolation level with row-level locking) and explicitly treat stock updates as critical sections:

```sql
START TRANSACTION;
    -- Lock the equipment row and calculate Available_Quantity
    SELECT
        Total_Quantity - IFNULL(
            (SELECT SUM(Quantity) FROM Reservation_Equipments
             WHERE Equipment_ID = ? AND ...), 0
        ) - IFNULL(
            (SELECT COUNT(*) FROM Maintenance
             WHERE Equipment_ID = ?
             AND Status IN ('Scheduled', 'In_Progress')), 0
        ) AS Available_Quantity
    FROM Equipment
    WHERE Equipment_ID = ?
    FOR UPDATE;

    -- check if Available_Quantity is sufficient
    -- if sufficient, insert Reservation_Equipments
COMMIT;
```

The `FOR UPDATE` clause (or the equivalent locking mechanism in the chosen DBMS) locks the selected equipment row so that only one transaction can modify the equipment allocation at a time. The second transaction will have to wait until the first commits, and will then calculate the updated `Available_Quantity` based on the new state. Combined with application-level validation that prevents negative or over-allocated quantities, this prevents lost updates and over-booking of equipment.

For facility bookings and training sessions, similar isolation is achieved: conflict-checking queries run inside the same transaction that inserts the new reservation. If isolation is not strong enough in the default setting, we can raise the isolation level (e.g. to `SERIALIZABLE`) or use stricter locking on the `Reservation` rows for a given facility and time range to avoid double bookings and phantom conflicts.

7.4 Durability

Durability means that once a transaction has been committed, its effects are permanent, even if there is a system crash or power failure immediately afterwards.

In our system, durability is provided by the underlying DBMS. When the application issues a `COMMIT` for a transaction (for example, after successfully creating a booking, enrolling a member into a training session, or marking a maintenance job as `Completed`), the DBMS:

- First writes the changes to the transaction log (often using a write-ahead logging mechanism).
- Then applies the changes to the data files on disk.

If a crash occurs, the DBMS replays the log during recovery to redo committed transactions and undo any incomplete ones, so that confirmed bookings and maintenance records are not lost. This technical durability is complemented by operational practices from our access-control design: the System Administrator (DBA role) is responsible for regular backups and recovery procedures, ensuring that even in the case of hardware failure, committed data can be restored from backups and logs.

Therefore, once a member receives confirmation that their booking has been successfully stored (i.e. the transaction has committed), they can be confident that the booking will remain in the system despite unexpected failures.