COMP 3005 - Fall 2025
Database Management Systems
Final Project Report
Raymond Liu 101264487
Afaq Virk 101338854

Table

Requirement	Assumption	Representation in ER Model
"The application will support three distinct categories of users—members, trainers, and administrative staff—each with specialized access privileges and clearly defined functional responsibilities."	All human actors share the same core attributes (name, contact information, credentials) and differ only by role. Each user holds exactly one role at a time.	Entity `User` with PK `id` and attributes `email`, `password`, `first_name`, `last_name`, `date_of_birth`, `sex`, `phone`. Entity `Role` with values *Member*, *Trainer*, *Admin*; FK `User.role` → `Role.id`.
"A member should be able to register by providing personal information such as name, date of birth, gender, and contact details, and later manage or update these details as needed."	The same personal-information structure applies to all user roles; registration and profile updates operate on a single user record.	Attributes listed above are stored once in `User`. No separate *Member* table is introduced; the distinction is represented by the `role` foreign key.
"Members will have the ability to establish and track personalized fitness goals—such as achieving a target body weight or reducing body fat percentage—and store health metrics like height, weight, heart rate [...]. These metrics should not overwrite previous entries but instead be recorded historically..."	Health metrics are modeled as individual time-stamped measurements. Goals are expressed in terms of these measurable metrics rather than free-form text. The owning user of a goal can be derived from the metric being targeted.	Entity `MetricType` (e.g., Weight, Body Fat %). Entity `Metric` with PK `id`, FK `user_id` → `User.id`, FK `metric_type` → `MetricType.id`, attributes `value`, `logged_date`. Entity `Goal` with PK `id`, FK `metric_id` → `Metric.id`, attribute `goal_date`. Redundant `user_id` is omitted from `Goal` to avoid transitive dependency.
"Members should be able to schedule, reschedule, or cancel personal training sessions [...] and [...] register for group fitness classes, subject to class capacity and schedule constraints. The system must ensure logical enforcement of business rules such as preventing overlapping bookings [...] and verifying trainer availability before confirming any reservation."	Actual classes/sessions are distinct from the availability slots defined by trainers. Sessions have a maximum capacity and may be attended by many members. Each member can enroll in a given session at most once.	Entity `Schedule` (trainer availability) with PK `id`, FK `trainer_id` → `User.id`, attributes `date`, `start_time`, `end_time`, FK `type` → `ScheduleType.id`. Entity `Session` with PK `id`, FK `schedule_id` → `Schedule.id`, attributes `name`, `size`, `desc`, `location`, `sex_restrict`. Associative entity `Enrollment` linking `User` (member) and `Session`, with PK `id`, FKs `session_id`, `member_id`, attribute `attended`, and `UNIQUE(session_id, member_id)`.
"Trainers should be able to specify their availability periods—either as recurring weekly slots or as individual time intervals—and update these schedules as needed. The system must prevent overlapping or inconsistent time slots for the same trainer..."	We store concrete time intervals; recurring availability patterns are handled at the application level by generating individual schedule rows. Overlap checks are enforced in application logic rather than as constraints in the ER model.	Entity `Schedule` as above, with each row representing one availability interval for a trainer, linked to `User` (trainer) and `ScheduleType`. No additional recurrence entity is modeled.
"Administrators will have the capability to manage room bookings to ensure that physical spaces—such as studios or training rooms—are properly allocated for classes and personal sessions, while avoiding conflicts in scheduling. They should also be able to oversee equipment management, including tracking the operational status of machines, logging maintenance issues, assigning repair tasks, and updating maintenance records once issues are resolved."	Rooms and equipment are modeled explicitly; each piece of equipment belongs to at most one room and has one current status. Detailed maintenance history and automatic room–session conflict detection are treated as application-level responsibilities in this version.	Entities `Room` (PK `id`, attributes `name`, `capacity`), `EquipmentStatus` (PK `id`, attribute `type`), and `Equipment` (PK `id`, attributes `name`, FK `room_id` → `Room.id`, FK `status_id` → `EquipmentStatus.id`). Session locations are stored as a text attribute `location`.
"The administrative component should also handle billing and payment processes, where the system can generate bills for various services such as membership subscriptions, personal training sessions, or class enrollments. Payments are to be simulated rather than processed through real financial gateways; however, the system should maintain a consistent record of invoices [...] and payment status..."	Bills may contain multiple services, and each service can appear on many bills. Only basic payment state is required (paid / unpaid); no sensitive payment details or external transaction identifiers are stored.	Entity `Service` for billable offerings. Entity `Bill` with PK `id`, FKs `admin_id` and `member_id` referencing `User`, attributes `date`, `paid`. Associative entity `Item` with PK `id`, FKs `bill_id` and `service_id`, and attribute `quantity` representing line items on a bill.

Normalization (2NF and 3NF)

All relations in the system use a single surrogate primary key (`id`). Therefore, no table has a composite primary key, and no non-key attribute can depend on part of a composite key. This implies that every

relation satisfies **Second Normal Form (2NF)**.

To verify **Third Normal Form (3NF)**, we consider functional dependencies within each relation. For all tables except `Goal`, every non-key attribute depends directly and only on the primary key. Foreign keys (such as `role`, `metric_type`, `schedule_id`, `room_id`) point to other relations, but do not determine additional non-key attributes within the same table, so they do not introduce transitive dependencies.

Initial Violation in `Goal` and Correction

In the initial design, the `Goal` relation was defined as:

`Goal(id, user_id, metric_id, goal_date)`.

The `Metric` relation has the form:

`Metric(id, user_id, ...)`,

which implies the functional dependency

$$\texttt{metric\_id} \rightarrow \texttt{user\_id}.$$

Combining this with the primary key of `Goal` yields the transitive dependency

$$\texttt{id} \rightarrow \texttt{metric\_id} \rightarrow \texttt{user\_id},$$

so the non-key attribute `user_id` depends on another non-key attribute `metric_id`. This violates 3NF.

To restore 3NF, we removed the redundant attribute `user_id` from `Goal`, giving:

`Goal(id, metric_id, goal_date)`.

The user associated with a goal is now obtained by joining `Goal.metric_id` to `Metric.id` and then `Metric.user_id`. In the revised relation, all non-key attributes depend solely on the primary key `id`, and there are no non-key $\rightarrow$ non-key dependencies.

With this correction, **all relations in the Health and Fitness Club schema are normalized to 2NF and 3NF**, ensuring minimal redundancy and clear functional dependencies throughout the design.

Object-Relational Mapping (ORM) Usage

We model role, metric_type, schedule_type, and equipment_status as separate lookup entities to support data integrity, potential extension with additional attributes, and flexible configuration without changing the main schema. Simple static domains like sex and sex_restrict are implemented as CHECK-constrained attributes instead of separate entities.

We used SQLAlchemy's Declarative ORM to map Python classes to relational tables and to express relationships and constraints directly in code. A single shared base class is defined with `declarative_base()`, and each entity declares its table name, columns, constraints, and relationships. This allows us to perform type-safe CRUD operations, joins, and cascades using Pythonic APIs instead of writing raw SQL for common operations. Relationships (e.g., one-to-many between `User` and `Metric`) are modeled with `relationship(...)` along with `back_populates` for bidirectional navigation, and we use options such as `cascade="all, delete-orphan"` to enforce lifecycle rules.

The application layer interacts with the database through a SQLAlchemy `Session`. Typical patterns include querying with `session.query(...).filter_by(...).order_by(...)`; creating domain objects, calling `session.add(...)` and `session.commit()`; performing relational joins for cross-entity views; and deleting records with transactional rollback on error. This approach keeps business logic in Python while the ORM translates it into efficient SQL.

Listing 1: Example SQLAlchemy model mapping

```
from sqlalchemy import Column, Integer, String, ForeignKey
from sqlalchemy.orm import relationship
from sqlalchemy.ext.declarative import declarative_base

Base = declarative_base()

class User(Base):
    __tablename__ = 'user'
    id = Column(Integer, primary_key=True)
    email = Column(String(255), nullable=False, unique=True)
    first_name = Column(String(100), nullable=False)
    last_name = Column(String(100), nullable=False)
    role = Column(Integer, ForeignKey('role.id'), nullable=False)

    role_obj = relationship("Role", back_populates="users")
    metrics = relationship("Metric", back_populates="user",
                           cascade="all, delete-orphan")
```

Listing 2: Typical ORM query/CRUD usage

```
# Recent metrics for a member
recent = (session.query(Metric)
          .filter_by(user_id=user.id)
          .order_by(Metric.logged_date.desc())
          .limit(5)
          .all())

# Create and commit a new metric
new_metric = Metric(user_id=user.id, metric_type=metric_type_id, value=Decimal("72.5"))
session.add(new_metric)
session.commit()

# Join across relationships to see upcoming sessions
upcoming = (session.query(Enrollment)
            .join(Session)
            .join(Schedule)
            .filter(Enrollment.member_id == user.id, Schedule.date >= date.today())
            .order_by(Schedule.date, Schedule.start_time)
            .all())
```

Major mapped entities

- Role, User
- Service, Bill, Item
- MetricType, Metric, Goal
- Room, EquipmentStatus, Equipment
- ScheduleType, Schedule, Session, Enrollment