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

Project Overview

This project implements a console-based management system for a health and fitness club using Python, PostgreSQL, and SQLAlchemy ORM. The system supports three distinct user roles—Member, Trainer, and Admin—with workflows centered on: member registration and profile management; fitness metric logging and goal tracking; trainer scheduling and session/class management; facility and equipment administration; and billing for services such as memberships and training. We ensured the database design focuses on a clear separation of concerns, data integrity via foreign-key and check constraints, and minimal redundancy through normalization. See the ERD in `docs/ERD.pdf` for a visual summary of the database schema.

ERD Requirement Mapping

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. Sessions are assigned to specific rooms, and room–session conflict detection (overlapping time windows on the same date) is enforced by the application at class creation time.	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` now includes FK `room_id` → `Room.id` (with optional `location` note).
"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.

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, more static domains like `sex` and `sex_restrict` are implemented as CHECK-constrained attributes instead of separate entities.

User model rationale (single entity + role) We represent Members, Trainers, and Admins using one `User` entity with a `role` foreign key to `Role`, rather than separate entities. This avoids duplicated personal data across multiple tables, simplifies authentication/registration, and ensures a single source of truth for shared attributes (name, contact, credentials). Least-privilege and separation of duties are enforced in the application via role-aware menus and permission checks: after login, a user's `role` gates access to role-specific actions and views. Sensitive operations (e.g., billing, schedule creation) are restricted to the appropriate role and validated server-side, while the database preserves integrity with foreign keys, uniqueness, and CHECK constraints.

Entity Sets

All entity sets in our final design use a surrogate key `id` and are modeled as regular (strong) entities. No weak entities are required because no entity's primary key depends on another entity's key.

- Regular entities: `Role`, `User`, `Service`, `Bill`, `Item`, `MetricType`, `Metric`, `Goal`, `Room`, `EquipmentStatus`, `Equipment`, `ScheduleType`, `Schedule`, `Session`, `Enrollment`.
- Note on `Enrollment`: This is an associative entity (between `User` as Member and `Session`) but is still a regular entity with its own surrogate key and a composite `UNIQUE(session_id, member_id)` constraint.

Relationship Sets, Cardinality, and Participation

Below are the main relationship sets (see ERD for diagram). Cardinalities and participation reflect our assumptions and constraints in code/DDL.

- **Role–User**: `Role(1)` to `User(N)`. Total participation on `User` (every user has exactly one role); partial on `Role`.
- **User–Metric**: `User(1)` to `Metric(N)`. Total on `Metric`; partial on `User` (a user may have no metrics).

- **MetricType–Metric**: `MetricType(1)` to `Metric(N)`. Total on `Metric`; partial on `MetricType`.
- **Metric–Goal**: `Metric(1)` to `Goal(N)`. Total on `Goal`; partial on `Metric`.
- **ScheduleType–Schedule**: `ScheduleType(1)` to `Schedule(N)`. Total on `Schedule`; partial on `ScheduleType`.
- **User(Trainer)–Schedule**: `User(1)` to `Schedule(N)`. Total on `Schedule`; partial on `User`.
- **Schedule–Session**: Intended `Schedule(1)` to `Session(0..1)` (one-to-one conceptually). The ORM enforces a single `Session` per `Schedule` slot; database FK ensures total participation on `Session`. (The DB could permit `1..N` without a uniqueness constraint; the application keeps it at most one.)
- **Session–Enrollment**: `Session(1)` to `Enrollment(N)`. Total on `Enrollment`; partial on `Session`. The pair `(session_id, member_id)` is unique.
- **User(Member)–Enrollment**: `User(1)` to `Enrollment(N)`. Total on `Enrollment`; partial on `User`.
- **Room–Session**: `Room(1)` to `Session(0..N)` via `Session.room_id`. Participation on `Session` is optional; overlapping bookings per room are prevented at class creation time in the application.
- **Room–Equipment**: `Room(1)` to `Equipment(0..N)`. Participation is optional on `Equipment` (room can be null during inventory/maintenance).
- **EquipmentStatus–Equipment**: `EquipmentStatus(1)` to `Equipment(N)`. Total on `Equipment`; partial on `EquipmentStatus`.
- **Bill–Item**: `Bill(1)` to `Item(N)`. Total on `Item`; partial on `Bill`.
- **Service–Item**: `Service(1)` to `Item(N)`. Total on `Item`; partial on `Service`.
- **User(Admin)–Bill**: `User(1)` to `Bill(N)` via `admin_id`. Total on `Bill`; partial on `User`.
- **User(Member)–Bill**: `User(1)` to `Bill(N)` via `member_id`. Total on `Bill`; partial on `User`.

Attributes and Keys

- **Primary keys**: All relations use surrogate integer `id` as PK.
- **Foreign keys**: See Database Schema Diagram (`ERD.pdf`).
- **Attribute types**: All attributes are simple and single-valued. There are no composite attributes. Multi-valued sets (e.g., a member attending many sessions) are represented by associative entities like `Enrollment`.
- **Derived attributes**: We avoid storing derived attributes. For example, age can be computed from `User.date_of_birth`. While BMI could be derived from height and weight, we chose to store it as a metric type to support explicit logging over time.
- **Constraints**: Uniqueness on `User.email`, `Role.name`, `EquipmentStatus.type`, `ScheduleType.type`, and `UNIQUE(session_id, member_id)` on `Enrollment`.
  CHECK constraints on `User.sex` and `Session.sex_restrict`.

Schema Correctness and Constraints

Data types, primary keys, foreign keys, and constraints align between our ORM models and the documented SQL schema. Referential integrity is enforced with FKs and, where appropriate, `ON DELETE CASCADE` is used (e.g., deleting a `Schedule` removes its `Session`; deleting a `Bill` removes its `Item` rows; deleting a `User` cascades to their `Metric` entries; and deleting a `Session` cascades to `Enrollment`). Uniqueness constraints and CHECK constraints ensure domain correctness (e.g., valid `sex` values for user). These choices collectively guarantee consistency of the data model and enforce business invariants close to the data.

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:

$$\texttt{Goal(id, user\_id, metric\_id, goal\_date)}.$$

The `Metric` relation has the form:

$$\texttt{Metric(id, user\_id, ...)},$$

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:

$$\texttt{Goal(id, metric\_id, goal\_date)}.$$

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.

Indexes, Triggers, and Views

Indexes: PostgreSQL automatically creates B-tree indexes for all primary keys and unique constraints (e.g., on `user.id`, `user.email`, `enrollment(session_id, member_id)`, `role.name`). These cover the primary access paths used by the application, so no additional manual indexes were necessary.

Triggers: No database triggers were implemented. Business rules that are temporal or cross-entity (e.g., preventing overlapping trainer availability) are enforced in application logic, while referential and domain constraints are handled declaratively via FKs and CHECKs. Cascading deletes are defined via FK actions rather than triggers.

Views: We did not define database-side views. Instead, read models are composed with ORM queries (joins, filters, ordering) which serve the same purpose within the application. If a database view were required (e.g., for reporting), the queries shown in the ORM section can be translated directly into a SQL `CREATE VIEW`.

Object-Relational Mapping (ORM) Usage

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 (Session with room assignment)

```
class Session(Base):
    __tablename__ = 'session'

    id = Column(Integer, primary_key=True)
    schedule_id = Column(Integer, ForeignKey('schedule.id', ondelete='CASCADE'), nullable=
        False)
    size = Column(Integer, nullable=False)
    name = Column(String(100), nullable=False)
    desc = Column(Text)
    location = Column(String(255))
    room_id = Column(Integer, ForeignKey('room.id'))
    sex_restrict = Column(CHAR(1), CheckConstraint("sex_restrict␣IN␣('M',␣'F',␣'A')"))

    schedule = relationship("Schedule", back_populates="session")
    enrollments = relationship("Enrollment", back_populates="session", cascade="all,␣delete-
        orphan")
    room = relationship("Room", back_populates="sessions")
```

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"),
    logged_date=datetime.now())
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

Implemented Features

Member Functions

- **User Registration**: Implemented (unique email, profile fields) via main menu.
- **Profile Management**: Implemented (phone/password updates). Goals and metrics are managed in dedicated flows.
- **Health History**: Implemented (time-stamped logging; history and simple trends; no overwrite).
- **Dashboard**: Implemented (latest metrics, active goals, upcoming sessions). Past class count is not displayed.
- **PT Session Scheduling**: Implemented (partially) as enroll/cancel flows. Rescheduling is supported as cancel + re-enroll. Availability is respected through scheduled sessions; room-conflict validation is enforced at class creation for session time slots.
- **Group Class Registration**: Implemented (capacity check; unique enrollment enforced).

Trainer Functions

- **Set Availability**: Implemented with overlap prevention on date/time windows.
- **Schedule View**: Implemented; shows assigned sessions and enrollment counts.
- **Member Lookup**: Implemented; case-insensitive name search, shows last metric and current goal when available; no edit rights.

Administrative Staff Functions

- **Room Booking**: Implemented. Admin assigns a room to each class; overlapping bookings on the same room/date/time are prevented by an application-level conflict check.
- **Equipment Maintenance**: Implemented at the level of updating equipment status (with association to rooms). No separate issue log/history is maintained.
- **Class Management**: Implemented (define new classes, assign trainer/time, cancel; lists upcoming classes).
- **Billing & Payment**: Implemented (generate bills, add line items, record simulated payments).

Validation, constraints, and error handling Foreign keys, uniqueness, and CHECK constraints enforce core integrity at the database level; the application adds business validations (e.g., preventing overlapping schedules, enforcing one enrollment per member per session) and handles invalid input with clear error messages and transactional rollbacks.

Role separation & usability After authentication, users are routed to role-specific menus (Member, Trainer, Admin). Commands are grouped by workflow, require few steps, and provide immediate, meaningful feedback (success/failure messages, lists, confirmations). Navigation is consistent across roles, leading to a straightforward CLI experience without major usability issues.

On indexes, triggers, and views vs. CLI Indexes, triggers, and views are database-layer constructs and are not "covered" by printing or input handling in the CLI. Our design relies on primary-key and unique indexes created automatically by PostgreSQL, uses declarative constraints and cascade rules instead of triggers, and composes read models via ORM queries instead of SQL views. This approach, we believe, meets the rubric's allowance to justify these as handled by the ORM/database automatically.