COMP3005 Final Project Report

Course Name: COMP3005 Database Management Systems
Instructor: Abdelghny Orogat
Group Members: Neil Varshney 101295007, Michael Fong 101300972, Liam McAnulty 101309972
Video Link: https://www.youtube.com/watch?v=k8grCX1jnFY

Summary:

This project implements a Health and Fitness Club management system using PostgreSQL, Python, and SQLAlchemy (ORM). This program supports three roles which are Members, Trainers, and Administrators. The program handles core operations including member profiles, health metrics, billings, group fitness classes, maintenance, etc.

The database schema consists of 15 3NF normalized tables with foreign keys, constraints, and relationships. The system also includes a database for querying unpaid bills, a trigger that sets payment dates when the bill is paid, and an index on the ParticipatesIn table on the class_id column. This system uses SQLAlchemy ORM for most of the database operations with bidirectional relationships. Limited use of raw SQL is present due to querying database views and creating triggers since direct ORM mapping is impractical for this functionality.

The system provides a foundation for managing a fitness club's daily operations with the use of proper database design.

ER Model: (ERD.pdf - Page 1)

We created the ER Model based on the project requirement and the roles of each type of user and the functionality we were going to implement. The entities and relationships with the cardinalities are listed below.

Entities:

- Bill
- Payment Method
- Admin Staff
- Maintenance Ticket
- Equipment
- Group Fitness Class
- Room
- Trainer
- Trainer Schedule
- Personal Training Session
- Member
- Health Metric

View relationship table below

Relational Schema and Mapping: (See ERD.pdf Page 2)

We mapped our ER Model into relational tables by using the methods learned based on cardinality between relationships and foreign keys.

Normalization: (See ERD.pdf Page 3)

The database schema was normalized to 3NF Normalization. This means each table's attributes have no repeating groups or multi-valued attributes, all non-key attributes fully depend on the primary key of that table, and no transitive dependencies (all attributes depend directly on the primary key). We ensured that when creating the entities and schema we would avoid normalization issues in the future. Full image as ERD.pdf.

Relationships:

Relationship Name	Entity 1	Entity 2	Description
ParticipatesIn	Member (N)	GroupFitness Class (M)	Many members can participate in many different group classes
Registers	Member (1)	PersonalTrainingSession (N)	1 member can register for many PT sessions
SetBy	Member (1)	FitnessGoal (N)	1 member can have many fitness goals
ChartedBy	Member (1)	HealthMetrics (N)	1 member can have many health metrics
PersonalTrainingBill	Bill (N)	PersonalTrainingSession (M)	Many bills are given out to many PT sessions
ChargedBy	Bill (N)	Admin (1)	Many bills are charged by/created by an admin
GroupFitnessBill	Bill (N)	GroupFitness Class (M)	Many bills are created for different group fitness classes
Assigned	Trainer (1)	GroupFitness Class (N)	Only 1 trainer per group fitness class
Assigned	Trainer (1)	PersonalTrainingSession (N)	Only 1 trainer per PT session
RequestedBy	Admin (1)	MaintenanceTicket (N)	1 admin can issue many tickets
Allocated	GroupFitness Class (N)	Room (1)	Many group fitness classes are allocated the same room

Has	Trainer (1)	TrainerSchedule (N)	A trainer can have many schedules
Allocated	PersonalTrainingSession (N)	Room (1)	Many PT sessions are allocated the same room
RequestedFor	Maintenance Ticket (N)	Equipment (1)	Many tickets can be issued for an equipment piece
Has	Room (1)	Equipment (N)	A room can have many pieces of equipment
Charges	Bill (N)	Member (1)	Many bills are created for a member

Application Functionality Implementation:

Member Functions:

Function	Logic
User Registration: Create a new member with unique email and basic profile info.	User registration prompts the user for their email, name, date of birth, gender and phone number and creates a member account.
Profile Management: Update personal details, fitness goals (e.g., weight target), and input new health metrics (e.g., weight, heart rate).	Profile updates query the Member by email and allow optional updates to name, gender, and phone. It validates inputs then commits changes. For updating fitness goals, it checks for the existing goal with member.fitness_goals relationship, then updates the amount, otherwise it creates a new FitnessGoal. Health metrics are added as new timestamped entries.
Health History: Log multiple metric entries; do not overwrite. Must support time-stamped entries.	The system creates a new HealthMetric object when a health metric is created for a member. It uses the composite primary key of (member_email, created) so each entry is unique and multiple entries per member is allowed. To view the latest health history, it queries through the member.health_metrics relationship and sorts by “created” descending and limits to 5 entries.
Dashboard: Show latest health stats, active goals, past class count, upcoming sessions.	On first loop calls view_member_fitness_goals and view_member_health_metrics and then displays all the upcoming group sessions and displays all the upcoming personal sessions before displaying the options the member has
Group Class Registration: Register for scheduled classes if capacity permits.	Goes through all the group classes one by one and gives the user the choice of joining the class, showing the next class in the list and exiting.

Trainer Functions:

Function	Logic
Set Availability: Define time windows when available for sessions or classes. Prevent overlap.	Trainers can update their recurring (weekly) availability schedule or submit an ad hoc (one time) schedule that overrides that day of the week's schedule. These times are defined by time stamp ranges. For the recurring availability, a past date was used, the first week of January in 1996, as the Jan 1st is a Monday, for ease of use. Ad hoc time periods cannot overlap. This is done using an Exclude Constraint. To prevent overlap on sessions and classes for trainers, a Trigger was created on both tables to query the Trainer's Schedule, as well as when other classes or sessions are. It will raise an exception if the trainer is not available. Trainers can view their recurring schedules, as well as actual schedules over a time period. A Trigger was created after INSERT on the Trainer table to initialize the recurring Trainer Availability to M-F 9-5.
Schedule View: See assigned PT sessions and classes.	Trainers can view their assigned PT sessions and classes over a time period. The query is a Union All between GroupFitnessClass and PersonalTrainingSession, based on the trainer_email and time_stamp_range. The time_stamp_range uses the && operator which checks if the ranges overlap. Also, to differentiate the two types of activities, a temporary Literal Column was used.

Member Lookup: Search by name (case-insensitive) and view current goal and last metric. No editing rights.	Performs a query on members for any person with the provided name and calls their view fitness goals and health metrics.

Admin Functions:

Function	Logic
Room Booking: Assign rooms for sessions or classes. Prevent double-booking.	Rooms are set upon creation of new classes. A trigger was put onto the GroupFitnessClass and PersonalTrainingSession tables to prevent double-booking. It tries to query other activities happening at the same time. If found, the INSERT will be blocked through an Exception.
Equipment Maintenance: Log issues, track repair status, associate with room/equipment.	When creating a ticket, the system validates the equipment ID, creates a MaintenanceTicket linked to the equipment through the FK, and sets the equipment status to OUT_OF_SERVICE. It is also linked to Admin through a FK on who created it. An admin can update the status to completed, then check if all tickets for that equipment is completed to mark it IN_SERVICE again. All equipment is linked to rooms through a FK room_id.

Class Management: Define new classes, assign trainers/rooms/time, update schedules.	Based on the Triggers earlier mentioned, restrictions were put onto the UI to provide the Admin with valid options to schedule classes for. There are two queries that were involved in this. 1. Trainer: Checks Ad Hoc and Recurring availability. (Only checks recurring if no Ad Hoc scheduled for the day) Then checks if the Trainer has activities already assigned during that time. These conditions are then used on the Trainer table to return a list of trainer emails 2. Room: Checks if there are already activities during the time period. If none is found, then it is available.
Billing and Payment: Generate bills, add line items, record payments. Simulate status updates.	Bill creation creates a bill with member_email, admin_email, and amount_due, then uses session.flush() to get the bill ID. For group fitness classes, it creates a GroupFitnessBill linking the bill to the class, preventing duplicate billing by checking fitness_class.group_fitness_bills relationship. Adding items checks to see if bill isn't paid, then increments amount_due for membership fees, or creates a new GroupFitnessBill and updates the total. Payment updates bill.payment_method and sets bill.paid = True, which triggers a database trigger (trigger_set_paid_date) that sets paid_date to CURRENT_DATE when paid changes from False to True.

View, Trigger, and Index:

This system meets the requirement of implementing one or more views, triggers, and indexes.

The view we created was a view that would display all unpaid bills with the admin, member, and bill information for that bill. This view would be created once the application connects to the database, and would ensure this view doesn't already exist to prevent any errors. We created this view because we thought that as we get more members within this application, the number of bills and unpaid bills would increase, meaning querying for them would be very often. Creating this view would make it more efficient by pre-joining the necessary tables and pre-filtering the records on bills that status is unpaid. This would reduce query complexity, as well as improve the performance of the admin dashboard.

One of the triggers we created was a trigger that would automatically set the paid_date to the current date once a bill is paid. This trigger is created once the application connects to the database and would ensure the trigger function and trigger are not created multiple times if it already exists. This trigger works by being called once a bill has been paid and checks if the old value of bill.paid changed from False to True, which means the bill has been paid and sets the current date to the paid_date. This would allow for users to accurately see when a bill has been paid.

The index we created was on the class_id in the ParticipatesIn model in our system. To do this, we simply included a parameter index=True when creating the column for the model. This would make it so when querying for records in this table, it will be more efficient, especially if the table has large amounts of records. In this system, as the amount of users gets large, the number of members and classes would become large as it also stores previous classes and not just future classes. This means, when checking for bills, or class capacity, or something else that requires checking who or how many participants there are for a class, it will be more efficient on the DBMS if using an index.

ORM Integration (SQLAlchemy):

We chose SQL Alchemy to map our PostgreSQL schema to our Python classes. This allowed us to use maintainable, object-oriented code instead of raw SQL. SQLAlchemy handles relationships, foreign keys, lazy/eager loading, and constraints. It also supports automatic session management and type safety which improves code quality and maintainability. This allowed us to work with Python objects and relationships (ex, member.bills, bill.group_fitness_bills) rather than manual joins and SQL strings.

For example, the member entity shows ORM mapping, relationships, and lazy loading. The columns are created with types and constraints, and uses relationship() with back_populates to create a bidirectional one-to-many link. These relationships are lazy loaded, which means the data is loaded on access with a separate query. For example, bills = relationship("Bill", back_populates = "member", lazy='select') creates a link where member.bills returns all bills for that member, and bills.member returns all bills associated with that member. This logic is used across our entities which allows us to access object attributes with manual SQL joins. However, across our application, there may be times where we want to bundle several joins to get related data across several entities which would be inefficient if doing this in separate queries. So we use eager loading to bundle the query into 1 database trip to get all the data needed for that specific entity.

Example of Entity Class

```
class Member(Base):
    __tablename__ = "Member"
    email = Column(String(255), primary_key=True)
    name = Column(String(100), nullable=False)
    date_of_birth = Column(Date, nullable=False)
    gender = Column(Enum(Gender), nullable=False)
    phone_number = Column(String(10), nullable=False)

    bills = relationship("Bill", back_populates = "member", lazy='select')

    health_metrics = relationship("HealthMetric",back_populates="member",lazy='select')

    fitness_goals = relationship("FitnessGoal", back_populates="member", lazy='select')

    personal_training_session = relationship("PersonalTrainingSession",back_populates="member"/
    ,lazy='select')

    participations = relationship("ParticipatesIn", back_populates="member", lazy='select')
```

Example of Using Relationships with Eager Loading

```
def view_my_bills(engine, member_email):
    with Session(engine) as session:
        try:
```

```
            member = session.query(Member).options(joinedload(Member.bills)/
            .joinedload(Bill.group_fitness_bills)/
            .joinedload(GroupFitnessBill.fitness_class))/
            .filter_by(email=member_email).first()

            if not member:
                print("Member not found.")
                return

            bills = member.bills


            if not bills:
                print("\nYou have no bills.")
                return

            print(f"\n=== Your Bills ===")
            ...
```

All 15 database tables are mapped to SQLAlchmy entity classes with popper column definitions. Relationships are handled correct by using relationship() with back_populates for bidirectional associations (one-to-many, many-to-many). We also check constraints in the models by using CheckConstrans for data validation (ex, non-negative amounts). The ORM was used for the majority of the CRUD operations, like session.query() for reading data, session.add() for insertions, updates through direct attribute assignment, with raw SQL being used only for querying database views where ORM mapping is impractical.

Some examples of Major Entities are Member, Trainer, Admin, Bill, GroupFitnessClass and TrainerAvailability.

Conclusion:

The Health and Fitness Club Management System meets the functional requirements and technical goals. It supports member profile management, health metric tracking with timestamps, fitness goal management, billing and payment processing, and equipment maintenance tracking. The system also supports the roles and restricted access with these roles for Members, Trainers, and Administrators. The database is also 3NF Normalized with 15 tables and uses proper foreign key relationships, lazy/eager loading queries, check constraints, views, triggers, and an index. The system properly incorporates SQLAlchemy ORM throughout the application and uses object-oriented concepts for the database operations. Overall, the system provides a solid foundation for managing a fitness club's daily operations while showing proficiency in database design and ORM integration.