VINUNIVERSITY
College of Engineering and Computer Science

COMP-3030

DESIGN DOCUMENT

Club Management System

Tran Ho Chi Thanh - V202401676

Ha Kien - V202401550

Conceptual & Logical Design

1. Functional and Non-Functional Requirements

Functional Requirements

The Clubs Management System (CMS) is designed to support Student Affairs Management (SAM) in managing student clubs efficiently. The system provides:

Club Management

- Create, update, delete, and retrieve club records.
- Classify clubs by category and operational status.

Member & Membership Management

- Maintain student profiles.
- Track student participation across multiple clubs, including roles, join dates, and membership status.

Advisor Management

- Assign faculty or staff advisors to clubs.
- Track advisor assignment periods.

Event Management

- Create and manage club events.
- Track event schedules, locations, participation, and approvals.

Budget & Financial Management

- Allocate annual budgets to clubs.
- Record expenses by category and manage reimbursements.
- Monitor spending against approved budgets.

Reporting & Analytics

- Generate summary reports on club membership, events, and financial activities.

- Support SQL aggregation and analytical queries.

Security & Auditing

- Enforce role-based access control.
- Log all critical data changes for audit purposes.

Non-Functional Requirements

- **Performance:** Indexing on frequently queried attributes (e.g., ClubID, StudentID, EventDate) to ensure fast query execution.
- **Scalability:** Support growth in the number of clubs, members, and events.
- **Reliability:** Referential integrity enforced using foreign keys and constraints (ON DELETE CASCADE/SET NULL where appropriate).
- **Security:** Role-based access control and secure password storage (Hashing).
- **Maintainability:** Modular schema design and normalization up to 3NF. *
- **Usability:** Schema supports integration with a future web interface.

2. Entity Relationship Diagram

Updated Entity List:

Core Entities

- Clubs (ClubID, *CategoryID*, ClubName, Description, Email)
- Members (StudentID, First Name, Last Name, Email, YearLevel)
- Advisors (AdvisorID, First Name, Last Name, Department, Email)
- Events (EventID, EventName, Description, StartDate, EndDate, Location, Status)

Associative Entities

- Club_Memberships (Clubs ↔ Members) (ClubID, StudentID, RoleID) (Composite Key)
- Club_Advisors (Clubs ↔ Advisors) (ClubID, AdvisorID) (Composite Key)
- Event_Organisation (Events ↔ Members) (EventID, MemberID, RegistrationDate, Status, Description) (Compostie Key)

Financial & Governance Entities

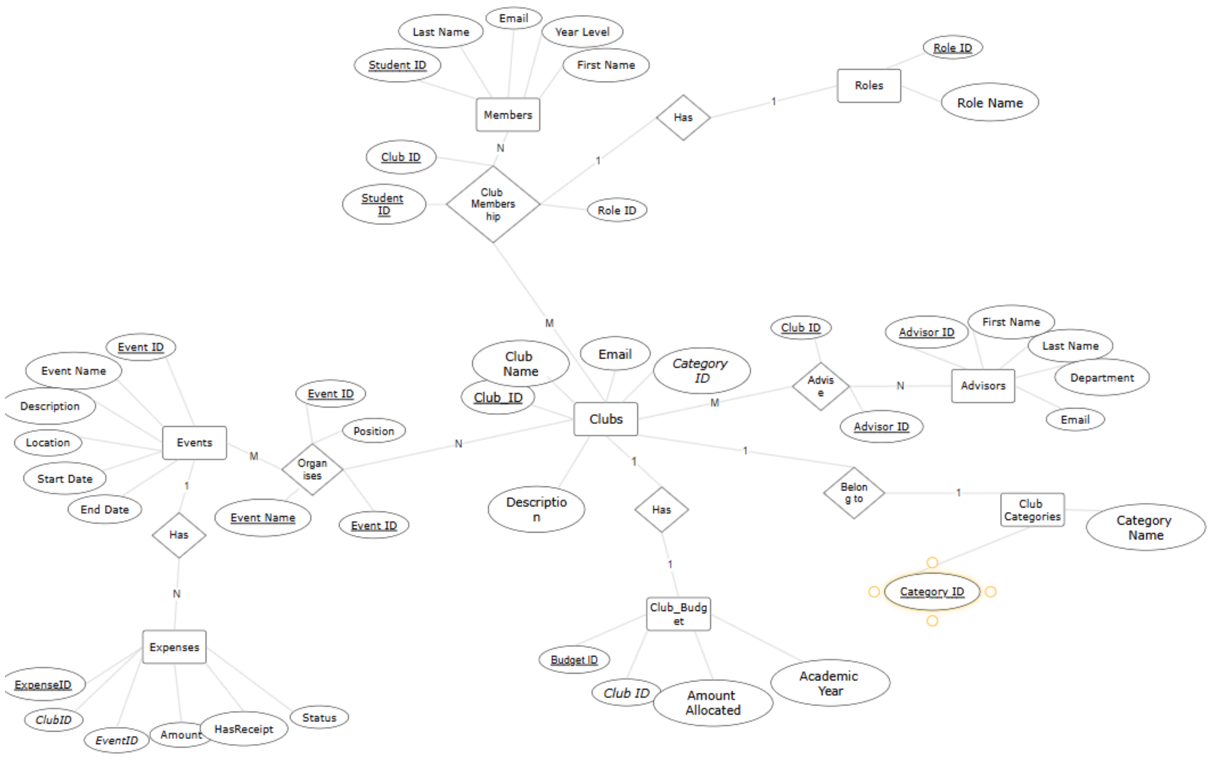
- Club_Budgets (BudgetID, *ClubID* , AcademicYear, TotalAllocated)
- Expenses (ExpenseID, *ClubID*, *EventID*, Amount, Description, ReceiptImage, ApprovalStatus)

- Reimbursements (<u>ReimbursementID</u>, *ExpenseID*, *MemberID*, DateRequested, DateProcessed, TransactionReference)

Engagement & System Entities

- Roles (<u>RoleID</u>, RoleName)
- Club_Categories (<u>CategoryID,</u> CategoryName)

ERD:



3. Normalisation Proof

The database schema is normalized to Third Normal Form (3NF):

First Normal Form (1NF):

All tables contain atomic values (each column holds a single piece of data) and no repeating groups (e.g., event dates are not listed as a comma-separated list within a single cell, nor are multiple repeating columns used).

Second Normal Form (2NF):

All non-key attributes are *fully functionally dependent on the entire primary key.*

- **Example (Composite Key):** In the associative table Event_Organisation (PK:{EventID, MemberID}), non-key attributes like RegistrationDate and Status depend fully on both the specific event and the specific member. If the PK were only EventID, the registration status would be ambiguous. This confirms 2NF adherence for all tables with composite keys.

Third Normal Form (3NF):

No transitive dependencies exist.

- **Proof:** Descriptive attributes such as role names, category names, and status values are correctly isolated and stored in separate lookup tables (Roles, Club_Categories). For instance, the Clubs table only stores the foreign key *CategoryID*, which points to the *CategoryName* in the Club_Categories table. This prevents the category name from being redundantly stored with every club record, eliminating the transitive dependency.

Therefore, all relations satisfy the conditions of 3NF, minimizing redundancy and preventing update anomalies.

Physical Schema Definition

1. MySQL DDL

-- ----------------------------

-- 1. Metadata and Lookup Tables

-- ----------------------------

-- Table: Club_Categories

```
CREATE TABLE Club_Categories (
    CategoryID INT PRIMARY KEY AUTO_INCREMENT,
    CategoryName VARCHAR(50) NOT NULL UNIQUE
);

-- Table: Roles
CREATE TABLE Roles (
    RoleID INT PRIMARY KEY AUTO_INCREMENT,
    RoleName VARCHAR(50) NOT NULL UNIQUE -- e.g., 'President', 'Member', 'Treasurer'
);

-- ----------------------------
-- 2. Core Entities
-- ----------------------------

-- Table: Clubs
CREATE TABLE Clubs (
    ClubID INT PRIMARY KEY AUTO_INCREMENT,
    CategoryID INT NOT NULL,
    ClubName VARCHAR(100) NOT NULL UNIQUE,
    Description TEXT,
    Email VARCHAR(100) NOT NULL UNIQUE,
    FOREIGN KEY (CategoryID) REFERENCES Club_Categories(CategoryID)

);


-- Table: Members (Students)

CREATE TABLE Members (

    -- StudentID is used as PK as it's the university-assigned unique identifier

    StudentID VARCHAR(20) PRIMARY KEY,

    FirstName VARCHAR(50) NOT NULL,

    LastName VARCHAR(50) NOT NULL,

    Email VARCHAR(100) NOT NULL UNIQUE,

    YearLevel VARCHAR(100)

);


-- Table: Advisors (Staff/Faculty)

CREATE TABLE Advisors (

    AdvisorID INT PRIMARY KEY AUTO_INCREMENT,

    FirstName VARCHAR(50) NOT NULL,

    LastName VARCHAR(50) NOT NULL,

    Department VARCHAR(100),

    Email VARCHAR(100) NOT NULL UNIQUE

);
```

```
-- Table: Events

CREATE TABLE Events (

    EventID INT PRIMARY KEY AUTO_INCREMENT,

    EventName VARCHAR(255) NOT NULL,

    Description TEXT,

    StartDate DATETIME NOT NULL,

    EndDate DATETIME NOT NULL,

    Location VARCHAR(255),

    Status ENUM('Pending', 'Approved', 'Rejected', 'Completed') NOT NULL DEFAULT
'Pending',

    -- Add constraint to ensure EndDate is after StartDate

    CHECK (EndDate > StartDate)

);



-- ----------------------------

-- 3. Associative Entities (Many-to-Many Relationships)

-- ----------------------------



-- Table: Club_Memberships

CREATE TABLE Club_Memberships (

    -- Composite Primary Key ensures a member holds a specific role in a club only once

    ClubID INT NOT NULL,
    StudentID VARCHAR(20) NOT NULL,

    RoleID INT NOT NULL,

    PRIMARY KEY (ClubID, StudentID, RoleID),

    FOREIGN KEY (ClubID) REFERENCES Clubs(ClubID) ON DELETE CASCADE,

    FOREIGN KEY (StudentID) REFERENCES Members(StudentID) ON DELETE
CASCADE,

    FOREIGN KEY (RoleID) REFERENCES Roles(RoleID)

);


-- Table: Club_Advisors

CREATE TABLE Club_Advisors (

    -- Composite Primary Key ensures an advisor is linked to a club only once

    ClubID INT NOT NULL,

    AdvisorID INT NOT NULL,

    PRIMARY KEY (ClubID, AdvisorID),

    FOREIGN KEY (ClubID) REFERENCES Clubs(ClubID) ON DELETE CASCADE,

    FOREIGN KEY (AdvisorID) REFERENCES Advisors(AdvisorID) ON DELETE
CASCADE

);


-- Table: Event_Organisation (Linking Events and the participating Members)

CREATE TABLE Event_Organisation (

    -- Composite Primary Key ensures a member registers for an event only once
    EventID INT NOT NULL,
    MemberID VARCHAR(20) NOT NULL, -- Renamed from StudentID for clarity in this
table context
    RegistrationDate DATETIME NOT NULL,
    Status ENUM('Registered', 'Attended', 'Cancelled') NOT NULL DEFAULT 'Registered',
    Description VARCHAR(255),
    PRIMARY KEY (EventID, MemberID),
    FOREIGN KEY (EventID) REFERENCES Events(EventID) ON DELETE CASCADE,
    FOREIGN KEY (MemberID) REFERENCES Members(StudentID) ON DELETE
CASCADE
);


-- ----------------------------
-- 4. Financial & Governance Entities
-- ----------------------------


-- Table: Club_Budgets
CREATE TABLE Club_Budgets (
    BudgetID INT PRIMARY KEY AUTO_INCREMENT,
    ClubID INT NOT NULL,
    AcademicYear YEAR NOT NULL, -- e.g., 2024
    TotalAllocated DECIMAL(10, 2) NOT NULL DEFAULT 0.00,
    -- Unique constraint ensures only one budget entry per club per academic year
    UNIQUE KEY idx_club_year (ClubID, AcademicYear),

    FOREIGN KEY (ClubID) REFERENCES Clubs(ClubID) ON DELETE CASCADE

);


-- Table: Expenses

CREATE TABLE Expenses (

    ExpenseID INT PRIMARY KEY AUTO_INCREMENT,

    ClubID INT NOT NULL,

    EventID INT NULL, -- Expense can be general or tied to a specific event

    Amount DECIMAL(10, 2) NOT NULL,

    Description TEXT,

    ReceiptImage VARCHAR(255), -- Store file path or URL

    ApprovalStatus ENUM('Pending', 'Approved', 'Rejected') NOT NULL DEFAULT
'Pending',

    FOREIGN KEY (ClubID) REFERENCES Clubs(ClubID) ON DELETE CASCADE,

    FOREIGN KEY (EventID) REFERENCES Events(EventID) ON DELETE SET NULL

);


-- Table: Reimbursements

CREATE TABLE Reimbursements (

    ReimbursementID INT PRIMARY KEY AUTO_INCREMENT,

    ExpenseID INT NOT NULL UNIQUE, -- Ensures one reimbursement per approved
expense
```

```
    MemberID VARCHAR(20) NOT NULL,

    DateRequested DATETIME NOT NULL,

    DateProcessed DATETIME NULL,

    TransactionReference VARCHAR(100) UNIQUE,

    FOREIGN KEY (ExpenseID) REFERENCES Expenses(ExpenseID) ON DELETE
CASCADE,

    FOREIGN KEY (MemberID) REFERENCES Members(StudentID) ON DELETE
RESTRICT

);
```

2. Definition of Views, Indexes and Partitioning Strategies

A. View for Active Club Financial Summary

```
-- View: FinancialSummary_V

-- Purpose: Calculate current financial standing for all active clubs.

CREATE VIEW FinancialSummary_V AS

SELECT

    C.ClubID,

    C.ClubName,

    CB.AcademicYear,

    CB.TotalAllocated,

    (SELECT SUM(Amount) FROM Expenses E WHERE E.ClubID = C.ClubID AND
E.ApprovalStatus = 'Approved') AS TotalExpensesApproved,

    CB.TotalAllocated - IFNULL((SELECT SUM(Amount) FROM Expenses E WHERE
E.ClubID = C.ClubID AND E.ApprovalStatus = 'Approved'), 0) AS CurrentBalance

FROM
```

```
    Clubs C
JOIN
    Club_Budgets CB ON C.ClubID = CB.ClubID
WHERE
    C.ClubID IN (SELECT ClubID FROM Clubs); -- Add logic for 'Active' status if a Status column existed
```

B. View for Club Leadership/President Access

SQL

```
-- View: ClubLeadershipReport_V
-- Purpose: Shows current members holding an executive role for easy reporting.
CREATE VIEW ClubLeadershipReport_V AS
SELECT
    C.ClubName,
    CONCAT(M.FirstName, ' ', M.LastName) AS LeaderName,
    R.RoleName,
    CM.StudentID
FROM
    Club_Memberships CM
JOIN
    Clubs C ON CM.ClubID = C.ClubID
JOIN
    Members M ON CM.StudentID = M.StudentID
```

```
JOIN

    Roles R ON CM.RoleID = R.RoleID

WHERE

    R.RoleName IN ('President', 'Treasurer')

ORDER BY

    C.ClubName, R.RoleName;
```

C. View for Event Attendance Analytics

This view calculates the total attendance for all approved events, supporting analytical queries.

```
-- View: EventAttendanceAnalytics_V

-- Purpose: Calculate total attendance for reporting on club engagement.

CREATE VIEW EventAttendanceAnalytics_V AS

SELECT

    E.EventID,

    E.EventName,

    E.StartDate,

    (SELECT COUNT(P.MemberID) FROM Event_Organisation P WHERE P.EventID = E.EventID AND P.Status = 'Attended') AS AttendedCount

FROM

    Events E

WHERE
```

```
    E.Status = 'Approved';
```

2. Index Creation Skeletons (Performance)

```
-- Index 1: Optimize searches for club members by student ID (frequent join column)

CREATE INDEX idx_members_email ON Members (Email);


-- Index 2: Speed up joins on the Club_Memberships table (frequent reporting table)

CREATE INDEX idx_memberships_club_student ON Club_Memberships (ClubID,
StudentID);


-- Index 3: Accelerate searches for events by date range (frequent query type for calendars)

CREATE INDEX idx_events_date_range ON Events (StartDate, EndDate);


-- Index 4: Optimize expense lookups and aggregations by status and club

CREATE INDEX idx_expenses_club_status ON Expenses (ClubID, ApprovalStatus);


-- Index 5: Speed up advisor lookups (since advisors might be searched by name/department)

CREATE INDEX idx_advisors_name ON Advisors (LastName, FirstName);
```

3. Partitioning Strategies (Scalability & Maintenance)

Partitioning is key to the **Scalability** requirement, especially for large tables like Events and Expenses. It physically separates data, improving query performance on time-series data.

A. Partitioning Strategy for Events Table (By Year)

This is ideal as events naturally cycle year-to-year, and most queries are likely historical (e.g., "Events in 2024").

SQL

```
-- Strategy: RANGE Partitioning on the year extracted from StartDate

ALTER TABLE Events

PARTITION BY RANGE (YEAR(StartDate)) (

   PARTITION p2023 VALUES LESS THAN (2024),

   PARTITION p2024 VALUES LESS THAN (2025),

   PARTITION p2025 VALUES LESS THAN (2026),

   PARTITION pCurrent VALUES LESS THAN (MAXVALUE) -- Catch-all for future or current year

);
```

B. Partitioning Strategy for Expenses Table (By Hash)

If queries are often complex and do not frequently use a time range, a HASH partition can distribute data evenly across disks, speeding up data access across the board.

SQL

```
-- Strategy: HASH Partitioning on the primary key (ExpenseID)

-- This distributes rows evenly across a fixed number of partitions (e.g., 4)

ALTER TABLE Expenses

PARTITION BY HASH(ExpenseID)

PARTITIONS 4;
```

Task Division and Plan

The following link will lead you to our Gantt Chart, which contains all details about our work division and project progress.

Gantt chart

Supporting Documentation

Rationale:

The revised schema prioritizes core club management and financial integrity by maintaining distinct tables for Clubs, Members, Advisors, and Events. This design adheres to Normalization principles.

Associative Entities with Composite Keys enforce the crucial Many-to-Many relationships, guaranteeing data integrity for memberships, advisories, and event participation.

Financial accountability is achieved by separating budget allocation (Club_Budgets) from transaction records (Expenses), ensuring a clear audit trail for spending.

Metadata tables (Roles, Club_Categories) ensure system-wide consistency and simplified maintenance.