CSE370 : Database Systems
Project Report
Project Title : Thesis Management System

Group No : 03, CSE370 Lab Section : 14, Summer 2025		
ID	**Name**	**Contribution**
24241095	Farhan Zarif	Structure Building (EER, Schema), Backend, Frontend
24241213	Md. Sajid Hossain	Structure Building (EER, Schema), Backend, Frontend

Table of Contents

Section No	Content	Page No
1	Introduction	3
2	Project Features	4
3	ER/EER Diagram	4
4	Schema Diagram	5
5	Normalization	6-8
6	Frontend Development	8-10
7	Backend Development	10-22
8	Source Code Repository	24
9	Conclusion	24
10	References	24

Introduction

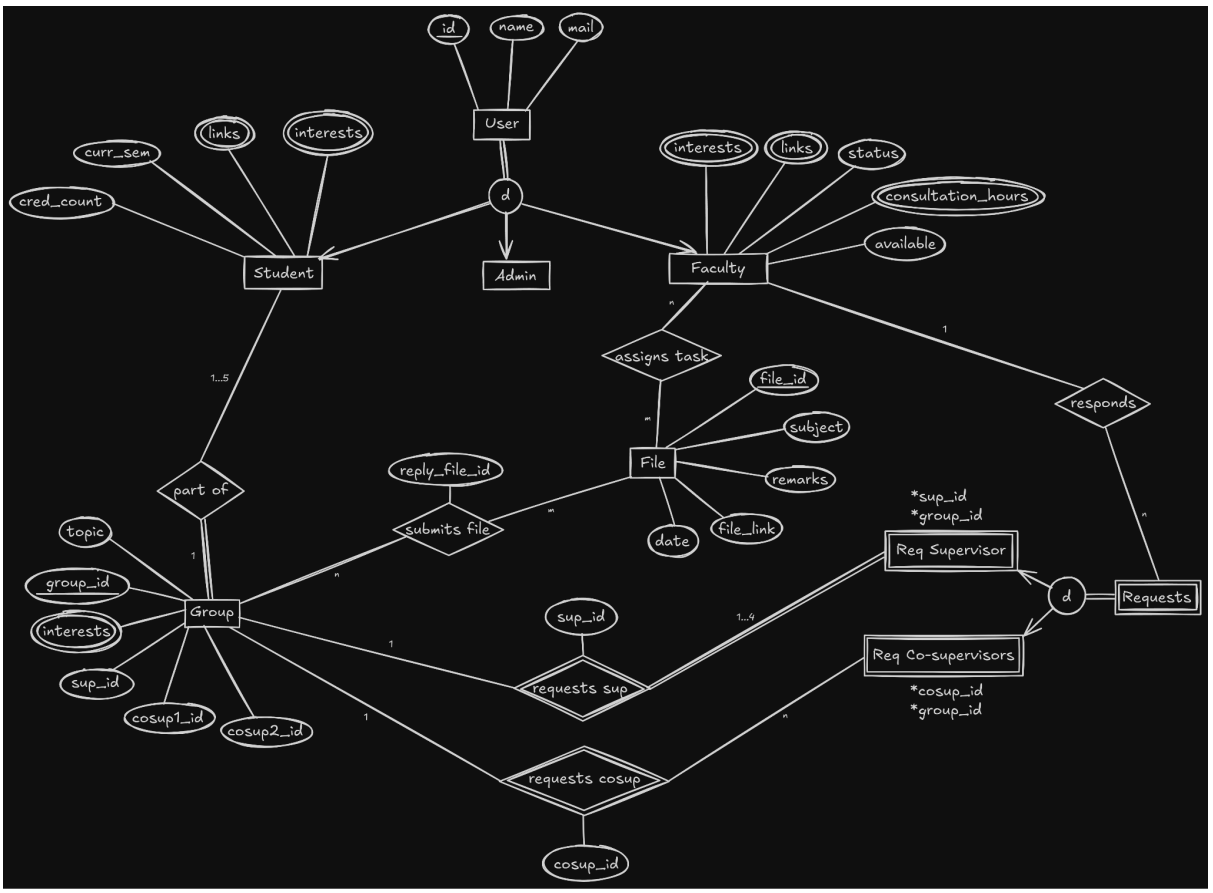
This project is called Thesis Management System which is a web based application designed to simplify and enhance the thesis supervision process for students and faculties. This offers a seamless solution for managing key academic tasks with efficiency and clarity.

This project offers a range of features to facilitate interaction between students and faculties. Administrators have the authority to manage user accounts including editing passwords, ensuring secure access control. Students and faculty can create and update their profiles, showcasing interests, professional links (like github), and consultation hours to foster better collaboration. The system enables students to browse available supervisors, view their profiles and select preferred supervisors based on available slots. Additionally, students can form thesis groups by exploring peers' profiles and aligning based on shared interests.
Slot Booking system allows student groups to request faculty for thesis supervision or co-supervision, with faculty members having the flexibility to accept or reject requests based on their group limits. Once a group is approved, other pending requests are automatically removed to avoid conflicts. The platform also supports thesis file submission in formats such as PDF, LATEX, allowing supervisors to review and provide structured feedback through a dedicated dashboard.

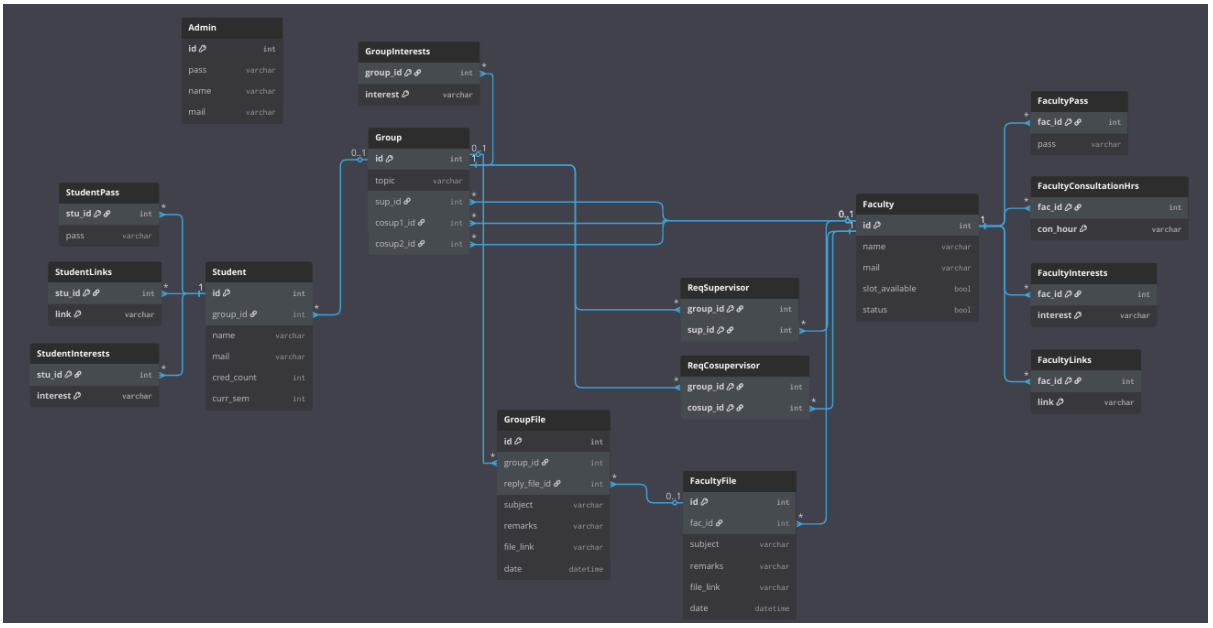
Project Features

ID, Name	Features	
24241095, Farhan Zarif	Ft 1	Admin Access
	Ft 2	Profile Edit and profile, gradesheet sharing
	Ft 3	Profile, Gradesheets and group file sharing
	Ft 4	Group Formation
24241213, Md. Sajid Hossain	Ft 1	Supervisor Selection
	Ft 2	Slot Booking System
	Ft 3	File submission
	Ft 4	Feedback System

ER/EER Diagram



Schema Diagram



Normalization

a. Explain if your converted Schema is in 1NF or not. If not, decompose it to 1NF.
b. Explain if your converted Schema is in 2NF or not. If not, decompose it to 2NF. Can there be any partial functional dependencies in your relational schema?
c. Explain if your converted Schema is in 3NF or not. If not, decompose it to 3NF. Can there be any transitive dependencies in your relational schema?

a) A relation is in First Normal form (1NF) if:

i) It has no repeating groups or arrays (all attributes are atomic)

ii) Each row is unique

iii) Domains are atomic (no sets or lists in a single attribute)

The converted relational schema is already in 1NF.

- all attributes are atomic (like 'name' is a single string, not a list of names, 'interest' in junction tables in a single value per row)
- Multi-valued attributes from the ER diagram (like interests, links, consultation hours) have been properly decomposed into separate junction or 1:1 tables (StudentInterests, FacultyLinks, FacultyConsultationHrs) avoiding repeating groups within any table
- Each table has primary key to ensure row uniqueness

b) A relation is in Second Normal Form (2NF) if :

i) It is in 1NF

ii) Every non-key attribute is fully functionally dependent on the entire primary key (no partial dependencies, where a non-key attribute depends on only part of a composite primary key)

The converted schema is already in 2NF

- Most tables have a single-column primary key (like student.id , Group.id, Admin.id, FacultyFile.id) so full dependency is inherent

- For tables with composite primary keys e.g StudentInterests (stu_id, interests), RegSupervisor (group_id, sup_id) have no non-key attributes, thus there are no chance for partial dependencies as FDs require a non-key attribute that depends on only a subset of the key.
- Inferred FDs like stu_id → studentLinks, here the link depends on (stu_id, link) as the key defines the unique link per student

No, there cannot be any partial functional dependencies in this schema

1. Partial FDs occur in relations with composite primary keys and non-prime attributes that depend on a proper subset of the key. Here junction tables lack non-prime attributes.

c) A relation is in Third Normal Form (3NF) if:

   a. It is in 2NF
   b. Every non prime attribute is non transitively dependant on the primary key (no transitive dependencies where A → B and B→ C implies A → C but C is not a part of the key and should depend directly on A)

The conventional schema is already in 3NF

- Single key tables (like student,faculty, group, admin, groupfile, facultyfile) have direct dependency: PK → all non prime attributes (e.g Student.id → {name, mail, cred_count, curr_sem)
- Domain logic supports directness: cred_count and curr_sem both depend directly on the student (id) not transitively . Similarly in groups: id → topic, sup_id, cosup1_id, student_count) are directly related.
- StudentInterests, FacultyConsultationHrs, RegSupervisor have no non key attributes, no transitive
- Separate tables for passwords (studentpass, facultypass) and files (groupfile, facultyfile) avoid derived data like file_link and date depend directly on file_id

No, there cannot be any transitive dependencies in this schema.

- ☐ Transitive FDs require a chain through a non key to non key attribute. Here all non prime attributes depend directly on the PK, as inferred from the domain. Attributes like name, mail, topic, subject, remarks and status are inherent properties of their entity. No attribute acts as an intermediary.

Frontend Development

Briefly discuss about Backend Development and add relevant Screenshots (if required) by mentioning Individual Contributions

Contribution of ID : 24241095 , Name : Farhan Zarif

→ I have worked on 9 front end files which are dependent on my respective features. All of the files are written in html and some of them use the jinja2 template.

1. create_group.html: This file serves as the frontend interface for creating a new group. It displays a form where a student can input a group name,submits it via POST request and includes links to return to the student dashboard or log out.
2. Admin_dashboard.html: This file contains the admin dashboard interface, displaying a welcome message and a table of users with their

IDs, names, emails and roles. It includes a search bar for filtering users by IDs and an “Edit Password” action link for each user. It also has a navigation back to the dashboard or logout.

3. Edit_faculty_profile.html: This file offers a faculty profile editing interface, showing the faculty ID and allowing updates to name, email, interests, links and consultation hours. It uses forms with delete/update options for existing interests/links/hours and input fields to add new ones, including dropdowns for consultation hour scheduling.
4. Group_profile.html: This file shows the group's profile, showing its name, ID, topic, supervisor and co-supervisor's name, interests and members with their details and profile links. It includes conditional sections for students or faculty using Jinja2 logic based on user role and membership with navigation option.
5. Edit_password.html : This provides a n interface for admins to edit a user’s password, displaying the user role and ID and featuring a form to input a new password with a back link to the dashboard.
6. Student_profile.html : This file shows a student's profile with ID, email, current semester, group ID , interests, and links using Jinja2 loops for dynamic content. It provides edit and search options for the student, conditional group-related actions and a logout link with a footer.
7. Faculty_profile.html: This presents a faculty profile, showing ID, email, interests, links and consultation hours with dynamic lists via jinja2 loops. It offers role based actions such as edit profile, search groups, manage requests for faculty and request options for students.
8. Edit_student_profile.html: This allows a student to edit their profile, showing ID, name, email, current semester and managing interest and links with delete/update/add options via forms
9. Edit_group_profile.html: This file enables group profile editing, displaying the group name and topic with options to manage interests like delete/update/add through forms.

Contribution of ID : 24241213**, Name : Md. Sajid Hossain**

→ I worked on 9 files which were part of my features and all the files here are written in html while some of them will use the jinja2 template.

1. Faculty_cosupervision_requests.html: Displays co-supervision requests including group ID, name, and topic with group details, offering accept/reject options via buttons using jinja2 loops.
2. Login.html: features a user-friendly login and signup interface offering role selection (student, faculty, admin) through dropdown menus within forms complete with back navigation links
3. Group_inbox.html: displays a comprehensive group inbox for file exchanges, allowing faculty and students to upload files or reply to existing ones with tables listing file details such as ID, sender, subject, remarks and download links powered by jinja2
4. Group_requests: showcases a table of sent group requests, including faculty ID, name, email and request type with a cancel option for each faculty
5. faculty_ my_groups.html: offers a detailed overview of groups supervised by faculty, displaying group ID, name, topic and supervisors with view profile links for each.
6. Search_faculties.html: includes a search bar for finding faculties by ID or name, presenting results in a table with ID, name, email and interests along with view profile links.
7. Search_groups: features a search interface for groups by ID or name displaying results with ID, name, and member count offering a join or view options based on user role
8. Faculty_supervision_request: Provides a clear list of supervision requests with group ID, name and topic featuring accept and reject buttons for faculty decisions updated via jinja2 loops.
9. Search_students.html: provides a search tool for students by ID or name, listing results in a table with ID, name, email and view profile links dynamically using jinja2 for interactive interface.

Backend Development

Briefly discuss about Backend Development and add relevant Screenshots (if required) by mentioning Individual Contributions

The backend uses FLASK, a lightweight Python web development framework with MySQL for data storage, bcrypt for password security and file handling via werkzeug. Sessions are secured with a random key and uploads are saved to a local “/uploads” folder.

Contribution of ID : 24241095 , Name : Farhan Zarif

→

1. Admin Access

- **Backend Implementation**: Admin access is secured through role-based authentication and authorization. The Admin table stores admin details (id, pass), with passwords hashed via bcrypt. Separate password tables (StudentPass, FacultyPass) allow admins to manage other users' credentials without accessing their full profiles.
- **How it Works**:
    - **Login**: On POST to /login, the route checks the selected role ("admin"). It queries SELECT pass FROM Admin WHERE id = %s with the provided user_id. If the password matches via

bcrypt.checkpw(), it sets session['user_id'] and session['role'] = 'admin', then redirects to the dashboard (url_for('dashboard')).
    - **Dashboard Functionality**: The admin dashboard route (inferred from frontend admin_dashboard.html and code structure) fetches all users via a JOIN query across Student, Faculty, Admin tables (e.g., SELECT id, name, mail, role FROM users_view or similar union). It renders a table with an "Edit Password" link per user, using Jinja2 loops ({% for user in users %}).
    - **Editing Passwords**: A dedicated route (e.g., /edit_password/<user_id>) handles POST requests to update passwords. It queries the appropriate table (e.g., UPDATE StudentPass SET pass = %s WHERE stu_id = %s), hashes the new password with bcrypt.hashpw(), and commits the change. Flash messages (flash()) provide user feedback.
    - **Authorization**: All routes check if 'role' is not in session or session['role'] != 'admin': return redirect(url_for('login_page')) to restrict access.
- **Framework Details**: Flask's session management ensures persistent login state. MySQL cursors are dictionary-based (dictionary=True) for easy JSON-like access in templates. Deployment involves securing the database connection string (e.g., via env vars) to prevent exposure in production logs.
- **Workflow Example**: Admin logs in → Session set → Dashboard loads user list from DB → Click "Edit Password" → POST new hash to DB → Redirect with success message.

2. Profile Edit and Profile, Gradesheet Sharing

- **Backend Implementation**: Profiles are stored in role-specific tables (Student, Faculty) with multi-valued attributes normalized into junction

tables (e.g., StudentInterests, StudentLinks, FacultyInterests, FacultyLinks). Gradesheet sharing is handled as file uploads or links (e.g., GitHub links in links field or via GroupFile for formal submissions). Editing routes update these tables atomically.

- **How it Works**:
    - **Profile Viewing/Editing**: Routes like /student_profile/<id> (for viewing) fetch data via queries (e.g., SELECT * FROM Student WHERE id = %s; SELECT interest FROM StudentInterests WHERE stu_id = %s;). For editing (/edit_student_profile/<id> or /edit_faculty_profile/<id>), a POST form updates core fields (e.g., UPDATE Student SET name = %s, mail = %s, curr_sem = %s WHERE id = %s) and handles multi-valued data: DELETE existing (e.g., DELETE FROM StudentInterests WHERE stu_id = %s), then INSERT new ones in loops based on form arrays (e.g., for interests/links).
    - **Gradesheet Sharing**: Students submit gradesheets as files or links during profile setup or group formation. If a file, it's uploaded to /upload (secure_filename checks), saved to UPLOAD_FOLDER, and the path inserted into StudentLinks or a dedicated Gradesheet table (inferred; e.g., INSERT INTO StudentLinks (stu_id, link) VALUES (%s, %s) for GitHub, or file_link for uploads). Supervisors review via profile views, querying the links/files.
    - **Authorization**: Only owners or admins can edit (if session['user_id'] != id and session['role'] != 'admin': abort(403)). Sharing is public within the system (e.g., students browse via search routes).
    - **Validation**: Flask's request.form.get() handles inputs; required fields use HTML5 validation, backed by DB constraints (e.g., NOT NULL on id).

- **Framework Details**: Jinja2 templates pass data (e.g., render_template('edit_student_profile.html', profile=profile_data, interests=interests_list)). Flask-WTF could be extended for CSRF protection (not shown, but recommended). MySQL transactions ensure atomic updates (e.g., conn.commit() after all INSERTs/DELETEs).
- **Workflow Example**: Student logs in → Access /edit_student_profile → Form submits name, interests (as comma-separated or array), gradesheet link/file → Backend loops to normalize/insert → Redirect to updated profile view.

3. Profile, Gradesheets and Group File Sharing

- **Backend Implementation**: Builds on profile sharing but extends to groups via Group table and file-specific tables (GroupFile, FacultyFile). Profiles/gradesheets are shared pre-approval (e.g., via links in Student or Group), while group files (thesis PDFs/LaTeX) are uploaded post-formation for review/feedback. reply_file_id in GroupFile enables threaded feedback.
- **How it Works**:
    - **Profile/Gradesheet Sharing in Groups**: During group formation or requests, students' profiles (including gradesheets) are queried and displayed (e.g., JOIN Student with Group on group_id). Sharing happens via inbox or dashboard: Upload gradesheet to uploads, insert file_link into GroupFile with group_id. Supervisors access via /group_profile/<group_id>, fetching via SELECT * FROM GroupFile WHERE group_id = %s.
    - **Group File Sharing/Feedback**: The /group_inbox/<group_id> route (visible in code) handles uploads. Faculty POST to insert into FacultyFile (e.g., INSERT INTO FacultyFile (fac_id, subject, remarks, file_link, date) VALUES (%s, %s, %s, %s, %s)), then

links it to GroupFile for the group. Students reply by providing reply_file_id, inserting into GroupFile with self-reference. Queries fetch faculty files (JOIN FacultyFile f ON ... WHERE reply_file_id IS NULL) and student replies (WHERE reply_file_id IS NOT NULL), ordered by date DESC.

    - **Download/Review**: /download/<filename> serves files securely, checking session. Feedback loop: Supervisor uploads feedback → Student replies → Automatic removal of pending requests on approval (via trigger or route logic updating RegSupervisor table).
    - **Authorization**: Role-checked (e.g., only group members/faculty can upload/view). Date handling uses datetime.now() for timestamps.

- **Framework Details**: Flask's file upload API (request.files['file']) saves to disk; send_from_directory for serving. MySQL fetches use dictionary cursors for easy template passing (e.g., faculty_files = cursor.fetchall()). For large files, streaming could be added, but current setup is basic.
- **Workflow Example**: Group formed → Student uploads gradesheet/thesis to inbox → DB insert with file_link → Faculty queries inbox, downloads, uploads reply → Threaded via reply_file_id for ongoing sharing.

4. Group Formation

- **Backend Implementation**: Managed via Group table (id, topic, sup_id, cosup1_id, cosup2_id, student_count) and junction tables (RegSupervisor, RegCoSupervisor for requests; GroupInterests). Limits (e.g., max 5 members) enforced via queries/checks.
- **How it Works**:
    - **Creation**: /create_group POST inserts into Group (e.g., INSERT INTO Group (topic, sup_id, ...) VALUES (%s, NULL, ...) initially

without supervisors). Sets student_count = 1 for creator, updates student's group_id in Student.

  - **Browsing/Joining**: Search routes (e.g., /search_groups) query SELECT id, name, student_count FROM Group WHERE name LIKE %s or by interests (JOIN GroupInterests). Students request join via POST to /request_join/<group_id>, inserting into a requests table (e.g., INSERT INTO PendingRequests (group_id, stu_id, status) VALUES (%s, %s, 'pending')).
  - **Formation/Approval**: Group leaders approve via /approve_request/<req_id>, updating student_count, adding to group members (perhaps via a GroupMembers junction), and removing pending requests (DELETE FROM PendingRequests WHERE group_id = %s). Supervisor assignment: POST to /request_supervisor/<group_id>/<fac_id> inserts into RegSupervisor; faculty accepts via /accept_sup/<req_id>, setting sup_id in Group and auto-rejecting others (UPDATE RegSupervisor SET status = 'rejected' WHERE group_id = %s AND sup_id != %s).
  - **Interests Matching**: During formation, shared interests from StudentInterests are used to suggest groups (query JOINs).
  - **Authorization**: Only students without groups can create/join; faculty views via /faculty_my_groups.
- **Framework Details**: Flask redirects (redirect(url_for('group_profile', group_id=group_id))) after actions. MySQL ensures integrity via foreign keys (e.g., group_id references Group.id). Loops in routes handle multi-member updates.
- **Workflow Example**: Student searches faculties/groups → Forms group via /create_group → Invites peers (email or in-app requests) → Peers

join/approve → Request supervisors → Faculty accepts → Group finalized, pending removed.

Contribution of ID : 24241213, Name : Md. Sajid Hossain

→

1. Supervisor Selection

- **Backend Implementation**: Supervisor selection is facilitated by the RegSupervisor and RegCoSupervisor tables, which store requests as composite keys (group_id, sup_id/cosup_id) with status fields (e.g., pending/accepted/rejected). The Faculty table includes availability indicators like slot_available (bool) and status (bool), while FacultyInterests and FacultyConsultationHrs provide filtering criteria. Student groups (from Group table) initiate requests, and faculty responses update the sup_id, cosup1_id, or cosup2_id in Group.
- **How it Works**:
    - **Browsing Supervisors**: A route like /search_faculties (inferred from frontend) handles GET requests with query parameters (e.g., interests). It executes a JOIN query: SELECT f.id, f.name, f.mail, fi.interest FROM Faculty f JOIN FacultyInterests fi ON f.id = fi.fac_id WHERE f.slot_available = TRUE AND fi.interest LIKE %s ORDER BY f.name, fetching available faculty matching student interests. Results are passed to the template for display, including profile links.

- **Request Submission**: On POST to /request_supervisor/<group_id>/<fac_id>, the backend checks eligibility (e.g., SELECT student_count FROM Group WHERE id = %s to ensure under limits; if session['role'] != 'student' or not in_group(session['user_id'], group_id): abort(403)). It inserts into RegSupervisor: INSERT INTO RegSupervisor (group_id, sup_id, status) VALUES (%s, %s, 'pending'), commits, and flashes a confirmation. For co-supervision, a similar insert targets RegCoSupervisor.
- **Faculty Review and Response**: Faculty access via /faculty_supervision_requests or /faculty_cosupervision_requests, querying SELECT g.id as group_id, g.name, g.topic, rs.status FROM RegSupervisor rs JOIN Group g ON rs.group_id = g.id WHERE rs.sup_id = %s AND rs.status = 'pending'. On POST to /accept_request/<req_id>, it updates UPDATE RegSupervisor SET status = 'accepted' WHERE id = %s, sets UPDATE Group SET sup_id = %s WHERE id = (SELECT group_id FROM RegSupervisor WHERE id = %s), auto-rejects others (UPDATE RegSupervisor SET status = 'rejected' WHERE group_id = %s AND sup_id != %s), and notifies via session flash or email (if extended).
- **Edge Cases**: Automatic removal of pending requests on approval uses triggers or post-update queries. Limits (e.g., max groups per faculty) are enforced by checking SELECT COUNT(*) FROM Group WHERE sup_id = %s before acceptance.
- **Integration**: Role-based access via session checks ensures only logged-in students can request and faculty/admins can approve. Queries use cursor.fetchall() for lists, enabling dynamic Jinja2 rendering (e.g., {% for req in requests %}).

- **Framework Details**: Flask's request.form.get() parses form data; url_for generates dynamic links. MySQL foreign keys maintain referential integrity (e.g., sup_id → Faculty.id). For scalability, pagination could be added to queries (e.g., LIMIT %s OFFSET %s). Deployment: Cache frequent searches with Redis to reduce DB load.

2. Slot Booking System

- **Backend Implementation**: Slots are managed through the Meet table (composite PK: group_id, fac_id, date_time; attributes: duration, status) and FacultyConsultationHrs (fac_id, con_hrs as varchar for time strings). This supports browsing available slots and booking for supervision consultations, tied to approved groups.
- **How it Works**:
    - **Browsing Available Slots**: In /faculty_profile/<fac_id>, the backend fetches consultation hours: SELECT con_hrs FROM FacultyConsultationHrs WHERE fac_id = %s, and available slots via SELECT date_time, duration FROM Meet WHERE fac_id = %s AND status = 'available' AND date_time > NOW(). It JOINs with Group for context, filtering by interests if needed (e.g., WHERE fi.interest IN (SELECT interest FROM GroupInterests WHERE group_id = %s)). Results are rendered with dropdowns or calendars in the template.
    - **Requesting a Slot**: POST to /book_slot/<group_id>/<fac_id> validates session (must be group member) and availability (SELECT * FROM Meet WHERE fac_id = %s AND date_time = %s AND status != 'booked'). If available, it inserts/updates: INSERT INTO Meet (group_id, fac_id, date_time, duration, status) VALUES (%s, %s, %s, %s, 'requested') or UPDATE Meet SET status = 'requested', group_id = %s WHERE fac_id = %s AND

date_time = %s. Faculty limits are checked (e.g., max concurrent slots).

- **Faculty Acceptance/Rejection**: Via /faculty_slot_requests/<fac_id>, queries pending: SELECT m.id, g.name, m.date_time FROM Meet m JOIN Group g ON m.group_id = g.id WHERE m.fac_id = %s AND m.status = 'requested'. On POST /accept_slot/<meet_id>, updates UPDATE Meet SET status = 'confirmed' WHERE id = %s, rejects conflicts (UPDATE Meet SET status = 'rejected' WHERE fac_id = %s AND date_time OVERLAPS %s), and notifies students (e.g., via redirect with flash).
- **Conflict Resolution**: Overlaps are detected with SQL conditions (e.g., WHERE date_time BETWEEN %s AND %s). Approved bookings remove pending supervisor requests if tied to selection.
- **Integration**: DateTime handling uses datetime module for NOW() and parsing form inputs. Status transitions (available → requested → confirmed/rejected) ensure atomicity with transactions (conn.commit() after checks).

3. File Submission

- **Backend Implementation**: File submissions use GroupFile table (id, group_id, reply_file_id, subject, remarks, file_link, date) for student/group uploads and FacultyFile (id, fac_id, subject, remarks, file_link, date) for supervisor reviews. Supports formats like PDF/LaTeX via MIME checks, with file_link storing paths to the UPLOAD_FOLDER.
- **How it Works**:
  - **Student/Group Submission**: POST to /submit_file/<group_id> (or inbox route) handles request.files['file']. It secures the filename

(secure_filename(f.filename)), saves to UPLOAD_FOLDER (e.g., f.save(os.path.join(app.config['UPLOAD_FOLDER'], filename))), and inserts: INSERT INTO GroupFile (group_id, reply_file_id, subject, remarks, file_link, date) VALUES (%s, %s, %s, %s, %s, NOW()). For initial thesis/gradesheet, reply_file_id = NULL; for replies, it's set to the parent file_id. Validation ensures group membership (SELECT * FROM GroupMembers WHERE stu_id = session['user_id'] AND group_id = %s).

    - **Supervisor Review**: Faculty access via inbox or dashboard, querying SELECT gf.id, gf.subject, gf.file_link FROM GroupFile gf WHERE gf.group_id IN (SELECT id FROM Group WHERE sup_id = %s) ORDER BY date DESC. Files are listed with download links; metadata (subject, remarks) is editable pre-upload.
    - **Gradesheet Integration**: During profile sharing or selection, gradesheets are submitted similarly, linked via StudentLinks (for GitHub) or as GroupFile entries. Auto-upload on group approval.
    - **Error Handling**: Checks file size/type (e.g., if f.content_type not in ['application/pdf', 'text/plain']), rolls back on failures (conn.rollback()).
    - **Integration**: Ties to supervisor selection (only approved groups can submit) and feedback (submissions trigger notifications).
- **Framework Details**: Flask's multipart form handling; MySQL datetime for auditing. For large files, chunked uploads could be implemented. Deployment: Use S3 for storage instead of local folder, updating file_link to URLs.

4. Feedback System

- **Backend Implementation**: Feedback is threaded via reply_file_id in GroupFile, allowing supervisors to reply to student submissions.

FacultyFile stores supervisor feedback files, linked back to groups. The dashboard/inbox queries both for a conversation view.

- **How it Works**:
    - **Providing Feedback**: POST to /provide_feedback/<group_id>/<file_id> (from inbox) uploads a file or text remarks. For faculty: Insert into FacultyFile (INSERT INTO FacultyFile (fac_id, subject, remarks, file_link, date) VALUES (%s, %s, %s, %s, NOW())), then link to GroupFile (UPDATE GroupFile SET reply_file_id = LAST_INSERT_ID() WHERE id = %s). Students reply similarly, self-referencing in GroupFile. Remarks are stored as varchar for text feedback.
    - **Viewing Feedback**: In /group_inbox/<group_id>, fetches threads: Faculty files (SELECT f.*, fac.name FROM FacultyFile f JOIN Faculty fac ON f.fac_id = fac.id WHERE f.id IN (SELECT reply_file_id FROM GroupFile WHERE group_id = %s AND reply_file_id IS NULL)), and replies (SELECT gf.*, g.name FROM GroupFile gf JOIN Group g ON gf.group_id = g.id WHERE gf.reply_file_id = %s). Displays in tables with download/reply buttons, ordered by date DESC.
    - **Dashboard Integration**: Supervisors review via /faculty_my_groups, linking to inboxes; students see in group profiles. Auto-removal of old threads or notifications on new feedback via session flashes.
    - **Approval Workflow**: Post-feedback, status updates in Meet or Group (e.g., UPDATE Group SET status = 'feedback_provided') trigger next steps like revisions.
    - **Integration**: Builds on file submission; ensures only authorized users (group members/supervisors) access via session and JOIN queries.

- **Framework Details**: Flask renders dynamic lists with Jinja2 ({% for file in faculty_files %}); MySQL self-joins for threading. For advanced feedback, add ratings (new column). Deployment: Log feedback actions for auditing, with backups for file integrity.

Source Code Repository

Upload source code to GitHub or Google Drive and share a publicly accessible link in this section of the report.

https://github.com/farhan-9820/CSE_370_Project

Conclusion

The Thesis Management System represents a successful implementation of a Flask-based web application integrated with a MySQL backend, designed to streamline thesis coordination at BRACU. The platform enables group formation, supervisor selection, slot booking, file submission, and feedback processes within a unified interface. Despite challenges in database design and coding, the project demonstrates effective functionality and potential to enhance academic efficiency. Future improvements, such as notification systems or mobile compatibility, could further optimize its utility.

References

Elmasri, R., & Navathe, S. B. (2016). *Fundamentals of database systems* (7th ed.). Pearson.

Silberschatz, A., Korth, H. F., & Sudarshan, S. (2011). *Database system concepts* (6th ed.). McGraw-Hill.

Connolly, T., & Begg, C. (2015). *Database systems: A practical approach to design, implementation, and management* (6th ed.). Pearson.