**Course:** CS220 – Database Systems

**Semester:** Spring 2025

**Instructor:** Dr. Shams Qazi

**Team Members:**

* Rabia Amir – 508439
* Rabeea Karim – 509291
* Ibsham Tariq – 502451

DATABASE SYSTEMS – SEMESTER PROJECT REPORT - ASSIGNMENT 3

NUST LOST AND FOUND MANAGEMENT SYSTEM

BS-AI-1-A

**Table of Contents**

1. Project Overview
2. Team Members and Work Distribution
3. ERP to Relational Mapping
4. Database Normalization
5. UNF
6. 1NF
7. 2NF
8. 3NF
9. ER Diagram
10. Frontend
11. Database screenshots
12. Data insertion
13. Sample queries

### INTRODUCTION:

**Project Overview:**

The ***NUST Lost & Found System*** is a centralized platform designed to help students and staff at NUST report and recover lost or found items. It simplifies the process of item recovery, enhances accountability, and maintains transparency in handling claims. Core features include user registration, item posting, categorization, claim management, admin notifications, and chat-based communication.

* **Purpose:**

Help users report and retrieve lost/found items on campus.

* **Target Users:**

NUST students, faculty, staff, and admins.

* **Core Functionality:**
  + User accounts with admin roles.
  + Item postings categorized by department and type.
  + Claim and verification mechanism.
  + Notifications for approvals and rejections.
  + Chat system between claimants and item owners.
* **Scope:**

Campus-wide digital solution integrated with role-based access control and real-time updates.

### ERP to Relational Mapping:

**Mapping ERP Concepts to Tables:**

* **Users (employees/students/admins)**  
  → Mapped to the Users table  
  Stores user profile data, roles (admin or not), verification, and contact information.
* **Departments (organizational units)**  
  → Mapped to the Departments table  
   Used to classify where the item was found or where the user belongs.
* **Item inventory (assets/items lost or found)**  
  → Mapped to the Items table  
   Central table for managing reported items, their status (lost, found, claimed), location, and ownership.
* **Categories (item types)**  
  → Mapped to the Categories table  
  Categorizes items like bags, electronics, accessories, etc., for better search and filtering.
* **Item images/documents**  
  → Mapped to the ItemImages table  
  Stores multiple images per item using a one-to-many relationship with Items.
* **Claims (requests to recover items)**  
  → Mapped to the Claims table  
  Manages user-submitted claims with status tracking (pending, approved, rejected), proof submission, and timestamps.
* **Messaging (user-to-user communication)**  
  → Mapped to the Messages table  
  Allows item owners and claimants to communicate directly via two-way chat.
* **System notifications (alerts/updates)**  
  → Mapped to the Notifications table  
  Sends automated alerts to users about claim approvals, rejections, or new posts.

**Use of Foreign Keys for Relationships**

* Items.user\_id → References Users.user\_id
* Items.department\_id → References Departments.department\_id
* Items.category\_id → References Categories.category\_id
* ItemImages.item\_id → References Items.item\_id
* Claims.item\_id → References Items.item\_id
* Claims.claim\_by → References Users.user\_id
* Messages.sender\_id & recipient\_id → Reference Users.user\_id
* Notifications.user\_id → References Users.user\_id

**Rationale:**  
I identified key business processes (reporting, verifying, claiming) and mapped each entity to normalized relational tables. Foreign keys ensure relational integrity between modules.

**NORMALIZATION- UPTO THIRD NF**

### First Normal Form (1NF)

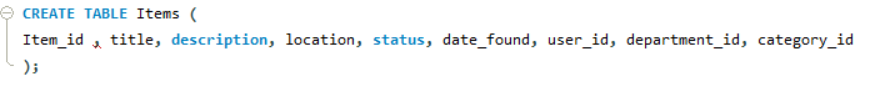
**1NF Rule:**

Eliminate repeating groups and ensure each attribute contains atomic (indivisible) values.

#### **Transformations:**

* Create separate tables for:
  + **Users**
  + **Departments**
  + **Categories**
  + **ItemImages**
  + **Claims**

Now the Items table looks like



Other atomic tables:

* ItemImages(image\_id, item\_id, image\_url)
* Users(user\_id, username, email, phone)
* Claims(claim\_id, item\_id, claim\_by, proof\_of\_ownership, claim\_status)

Repeating and multi-valued attributes are removed.

### Second Normal Form (2NF)

**2NF Rule:**

Must be in 1NF **and** all non-key attributes must depend on the **entire primary key** (no partial dependencies).

This mostly applies to **composite primary keys**, which you avoided by using **auto-increment id fields** (good choice). However, conceptually:

#### **Problem in UNF:**

If ItemImages had a composite key like (item\_id, image\_url), then uploaded\_at depends only on image\_url.

#### **Solution:**

Give ItemImages its own primary key:



Also ensure:

* proof\_of\_ownership in Claims depends on claim\_id, not partially on item\_id.

Partial dependencies resolved.

### Third Normal Form (3NF)

**3NF Rule:**

Must be in 2NF **and** no transitive dependencies i.e., **non-key attributes must not depend on other non-key attributes**.

#### **Transitive Dependency Example:**

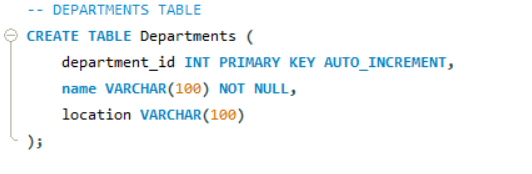
In Users, if we stored:



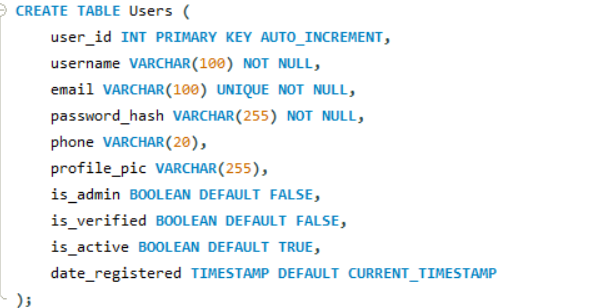
Here department\_name depends on department\_id, not directly on user\_id.

#### **Solution:**

* Keep department\_name in its own Departments table.

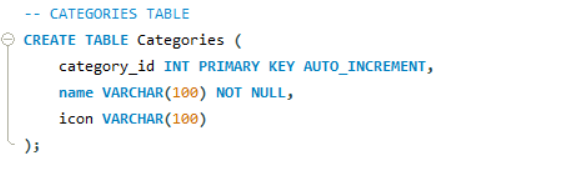


* Use a department\_id foreign key in Users.

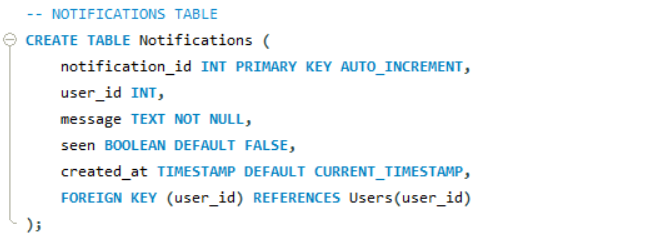


Likewise, keep:

* Category info in Categories



* All notification text in Notifications instead of embedding logic elsewhere.

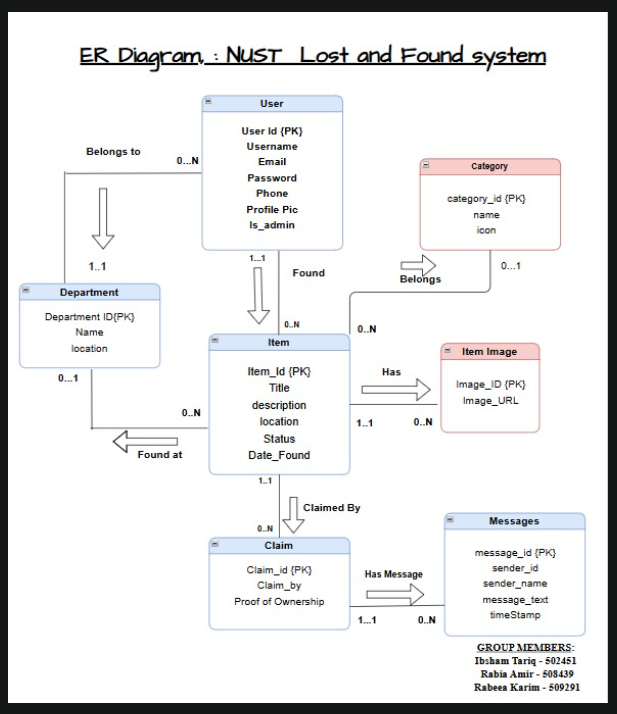


All attributes now depend directly on the primary key.

**SUMMARIZED IN TABLE:**

|  |  |  |
| --- | --- | --- |
| **Normalization Level** | **Main Goal** | **Application in our Schema** |
| **UNF** | Raw data with repeated & nested fields | Raw item records with images, claims, users all together |
| **1NF** | Remove repeating groups, atomic fields | Separate tables: Users, Claims, Departments, ItemImages |
| **2NF** | Eliminate partial dependencies | Use composite primary keys (like user\_id); all fields depend on full key |
| **3NF** | Eliminate transitive dependencies | Separate related data into reference tables (e.g., Departments, Categories) |

**ER DIAGRAM**

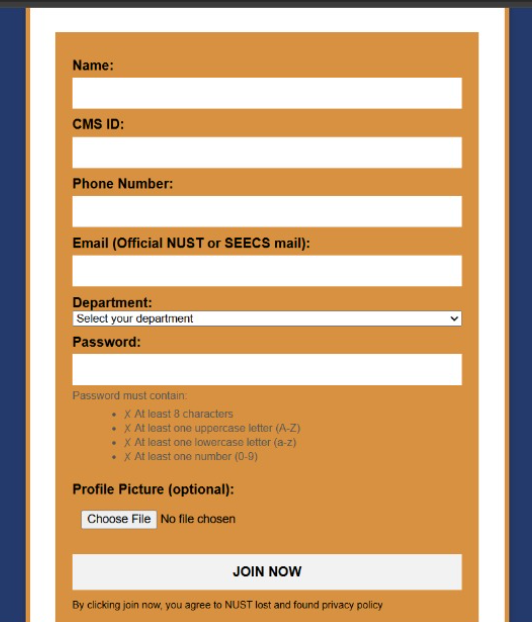
****

**Updation:**

At the time of execution of project ; table of messages has been removed and notifications table has been added

**FRONTEND :**

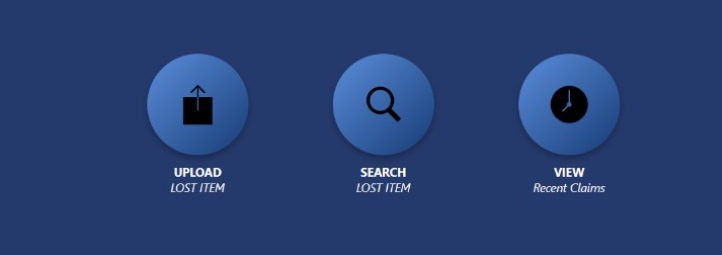
* **Signup page:**

****

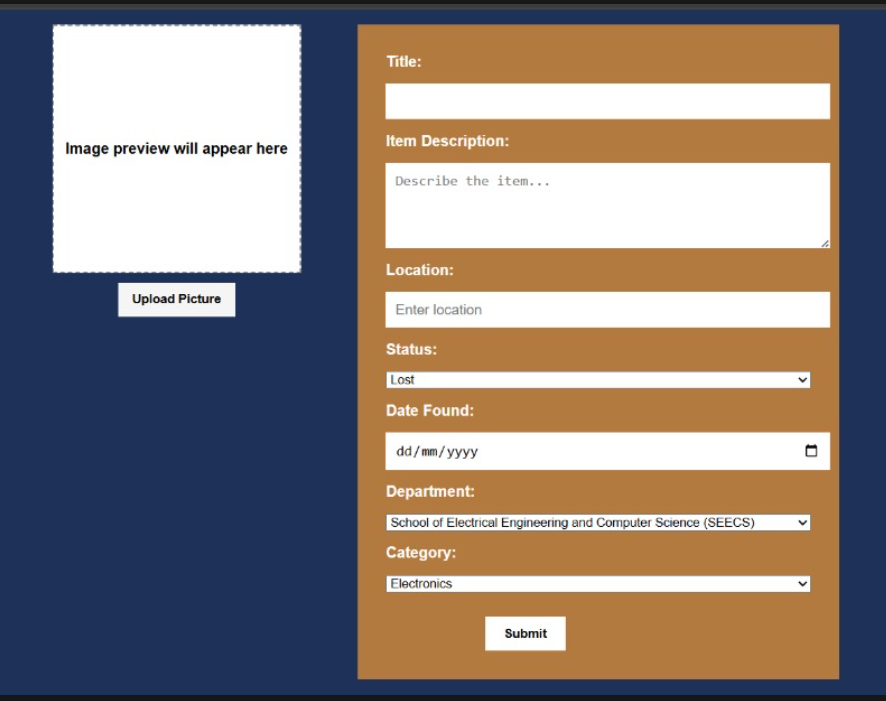
* **Login page:**

****

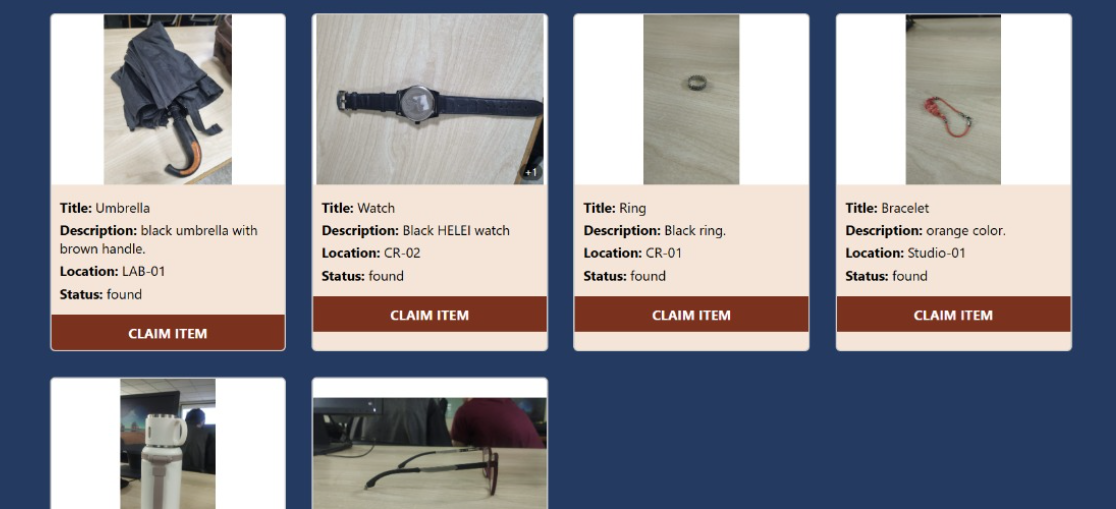
* **Home page:**

****

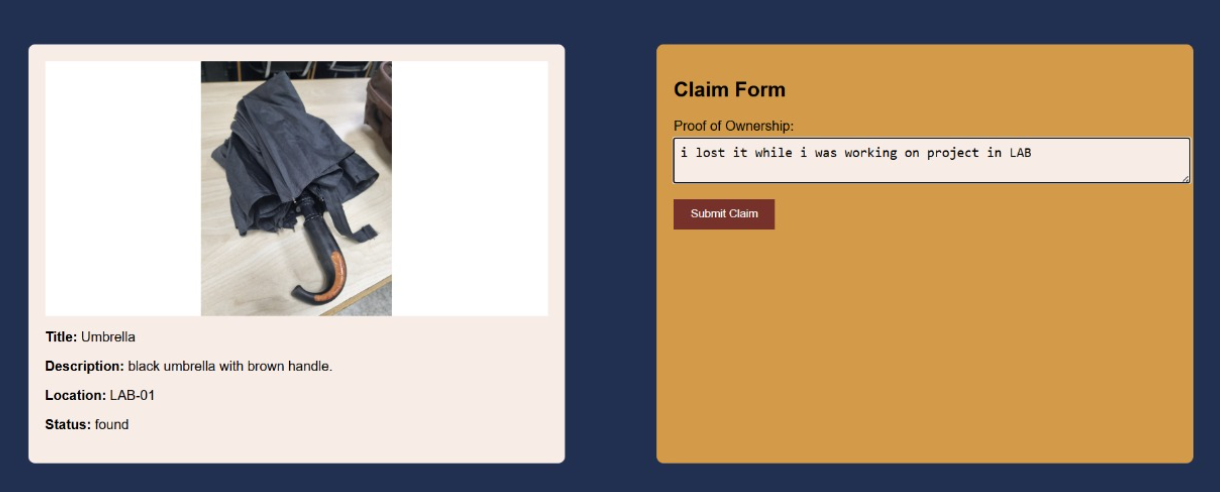
* **Upload item:**

****

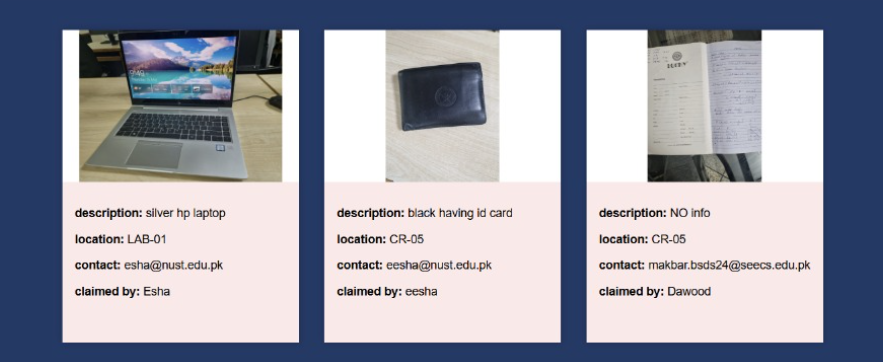
* **View items:**

****

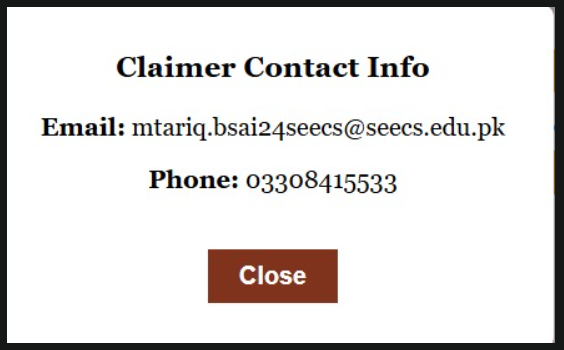
* **Claim item:**

****

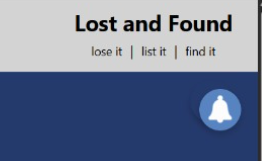
* **Claim request form:** ****
* **Recent claims:**

****

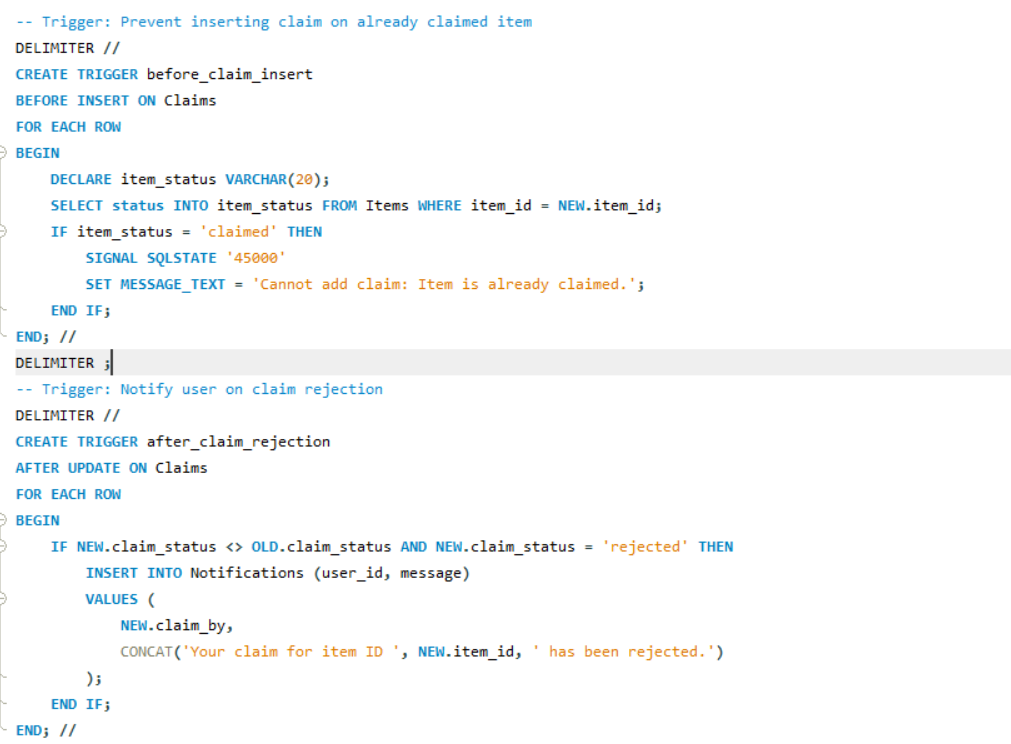
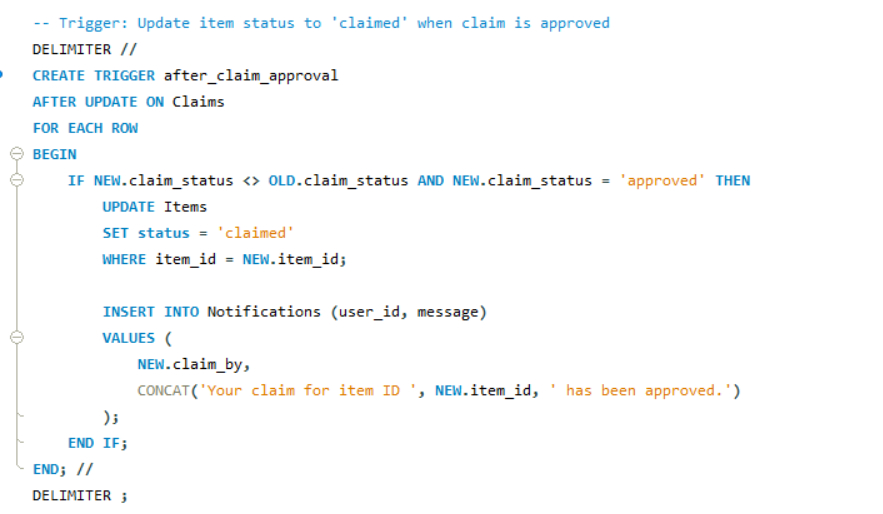
* **Claimer info:**

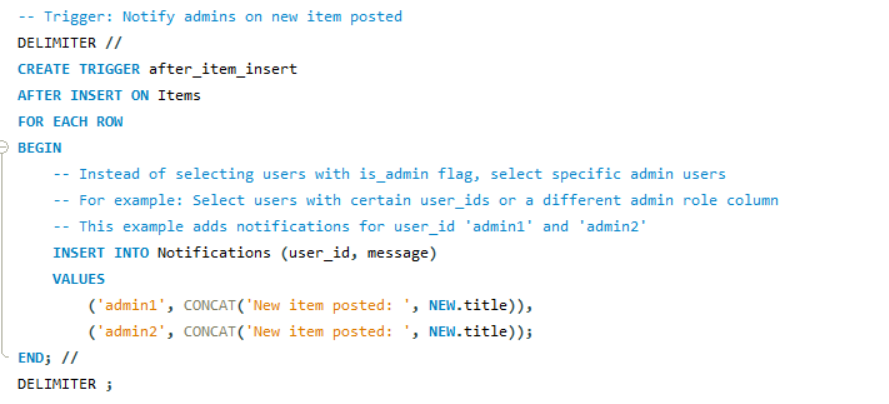
****

* **Notifications of recent claims:**

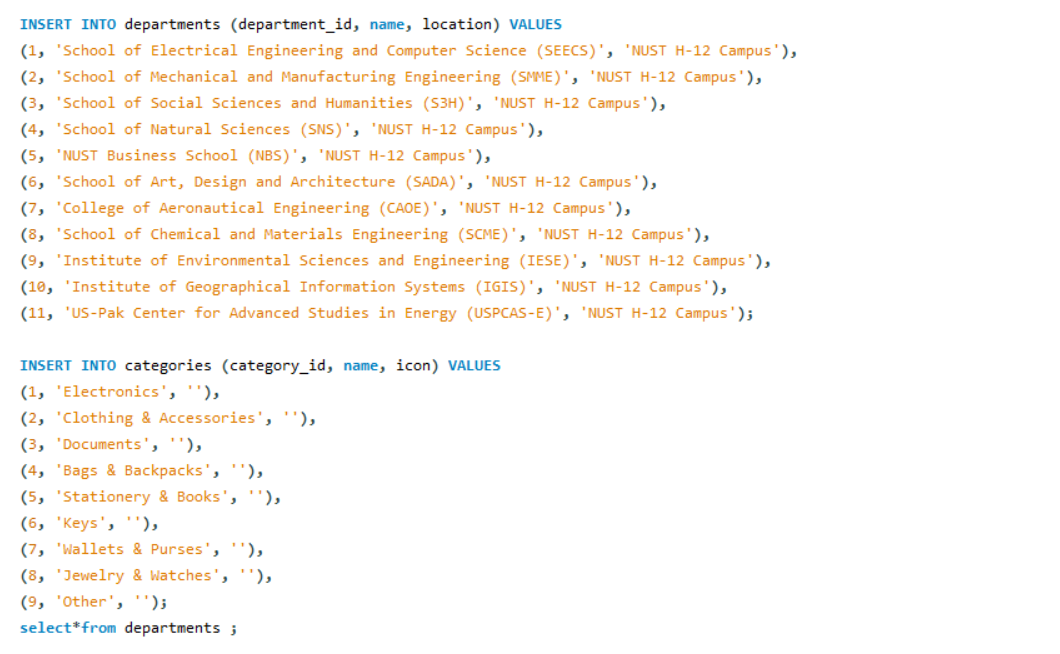
****

**DATABASE SCHEMA**

****  

****

**DATA INSERTION**

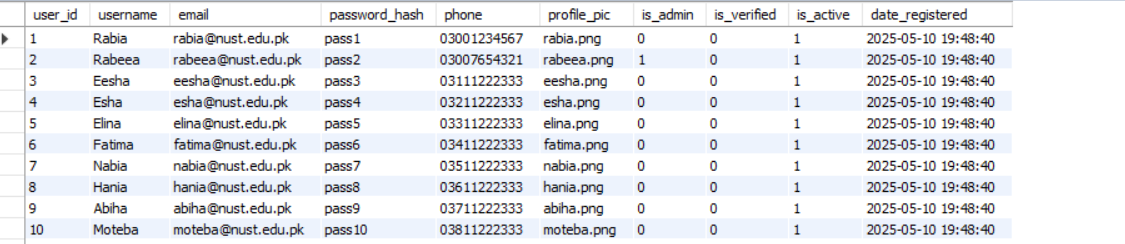


**QUERIES**

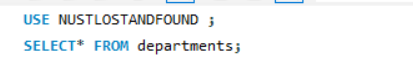
**Query:**

****

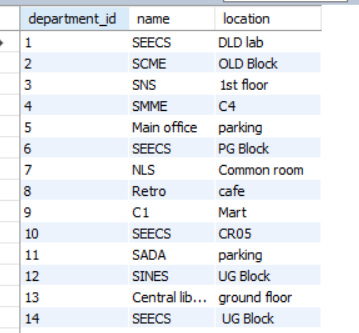
**Output:**

****

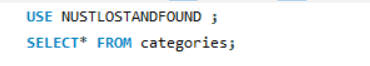
**Query:**

****

**Output:**

****

**Query:**

****

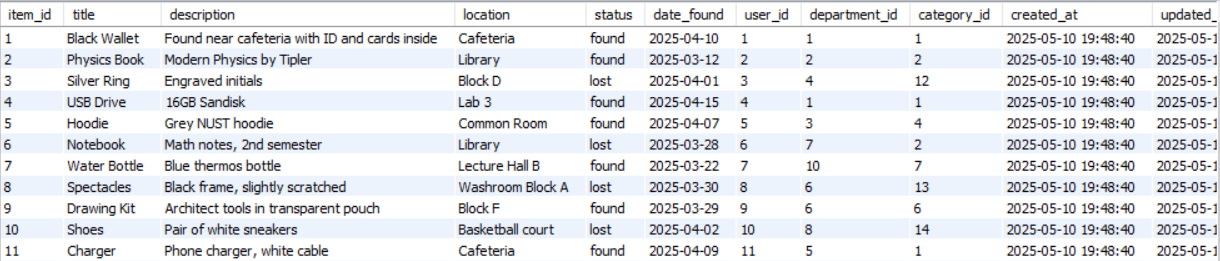
**Output:**

****

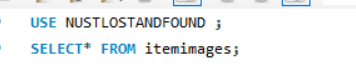
**Query:**

****

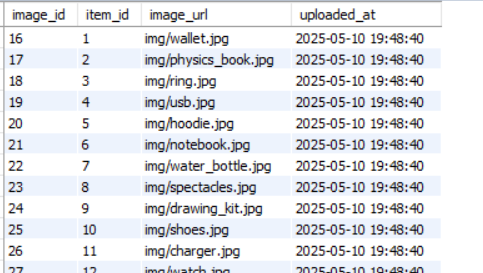
**Output:**

****

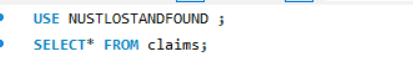
**Query:**

****

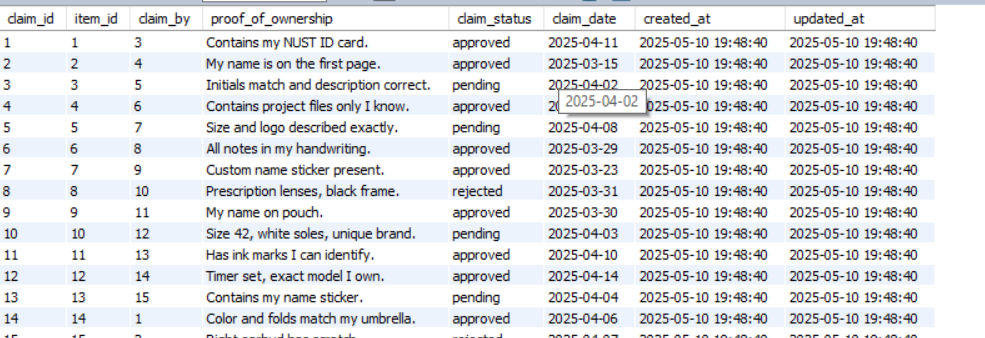
**Output:**

****

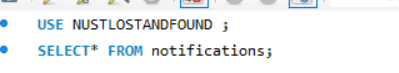
**Query:**

****

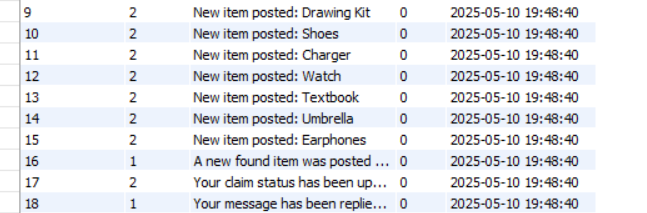
**Output:**

****

**Query:**

****

**Output:**

****

**CONCLUSION**

This project offered practical experience in modeling real-life processes within a relational database system.I gained valuable insights into normalization, the use of foreign key constraints, and the role of triggers in automating workflows. One of the main challenges was maintaining data integrity during claim processing and ensuring item statuses were updated correctly. I overcame these issues by implementing SQL triggers and establishing clear, well-defined relationships between entities.