Contact Management System

CMPT 308N-113

11161

The Data Bros

Marist College School of Computer Science and Mathematics

Shane Seeley (Team Leader)

Email: shane.seeley1@marist.edu

Jozef Maselek

Email: jozef.maselek1@marist.edu

GitHub: https://github.com/ShaneSeeley28/CMPT308N-113_Contact_Management_System_DataBros

Date: 9/26/2023

Table of Contents

Section 1

Why We Make a Great Team

Jozef's Paragraph:

I want to work with Shane because I know he is a student who makes good decisions on essential requirements. I know that he can lead our team to meet project conditions and submit work on time. Shane is a good leader due to his tenacity in doing every task assigned to him. Shane is also a hard worker and I know that he will submit our projects on time and get his work done effectively.

Shane's Paragraph:

I am excited to work with Jozef because I know we can get our work done on time and answer questions efficiently. We both are passionate about computer science and learning about topics like databases. The reason for me being the team leader is due to my knowledge of SQL and my personality, which will help guide our team to being as effective as possible.

Different Contact Management Systems:

Cloud-Based Contact Management System “Google Contacts”:

Advantages: Contacts are accessible from any device with internet access, and data is backed up regularly, which reduces the risk of data loss (1).

Disadvantages: Requires an internet connection to access data, which can create limitations (1).

CRM Systems “Salesforce”:

Advantages: Automation CRM systems automate specific tasks like email campaigns and customer interactions using stored data. (2)

Disadvantages: CRM systems can be expensive and may not be needed for small businesses. (2).

Spreadsheet Software “Excel”:

Advantages: allows for flexibility with data organization and sorting, data can also be integrated with other software using Excel. (3)

Disadvantages: Version control issues collaboration in Excel can be challenging for data entry. (3)

Why our Software is the right choice and Our Goals.

We believe that our system will be cost-effective and perfect for small businesses looking for a Contact Management System. We will create a secure SQL database to store customers' information that users can easily access with a state-of-the-art user interface. Our system will have up-to-date security making it perfect for storing user data. With our Contact Management System users can get the best possible CMS for a price that's affordable.

Section 2

Describe how you created this mini world and how you selected the entities, attributes, relationships, participations, and cardinality.

To create this mini world, we thought about what would be important for a contact management system to function and what kind of amenities users might want. Because it's acontact-focusedd database the entities are centered around the contacts entity. Attributes belonging to each entity are self-explanatory and are indicative of the table they are assigned to. The entities not directly related to the contacts entity are meant to further elaborate and give insight on information like departments or notes on interactions. Participations are tied together by necessity for the most part.

User ER Model - Provide a short description of each entity, attribute, relationship, participation, and cardinality.

Address is an entity that stores an AddressID, the street address, city, state, country, zip code, and the name of the person associated. It is related to the contacts entity and stores the address information of specific contacts. Many contacts can store many addresses.

Contacts is an entity with the attributes Fname, Lname, Cellphone, Workphone, FaxNum, Email, Gender, and Birthday. It stores the contact info for a person. It's related to most of the other entities in many different ways.

User is an entity with the attributes Username, Password, UserID, IsAdmin, Email, Fname, and Lname. It is an entity to store the user’s information and has an attribute to discern whether it's an admin. Its connected to contacts in a 1:N configuration.

Job, Company, and Department are related entities, describing the jobs of individual contacts, the company they work for, and the department they work in that company. Job has the attributes JobID, Title, JobLocation, CompanyID, and ContactID. It is connected to contacts and company in a 1:1 relationship. Company has CompanyID, CompanyName, CompanyLocation, CompanyType, and Employees and is connected to Job and Department in a 1:1 relationship. Department is connect to Company and Contacts in a 1:1 relationship and has the attributes DepartmentID, Dname, Dlocation, DcompanyID, Demployees, and Dmanagername.

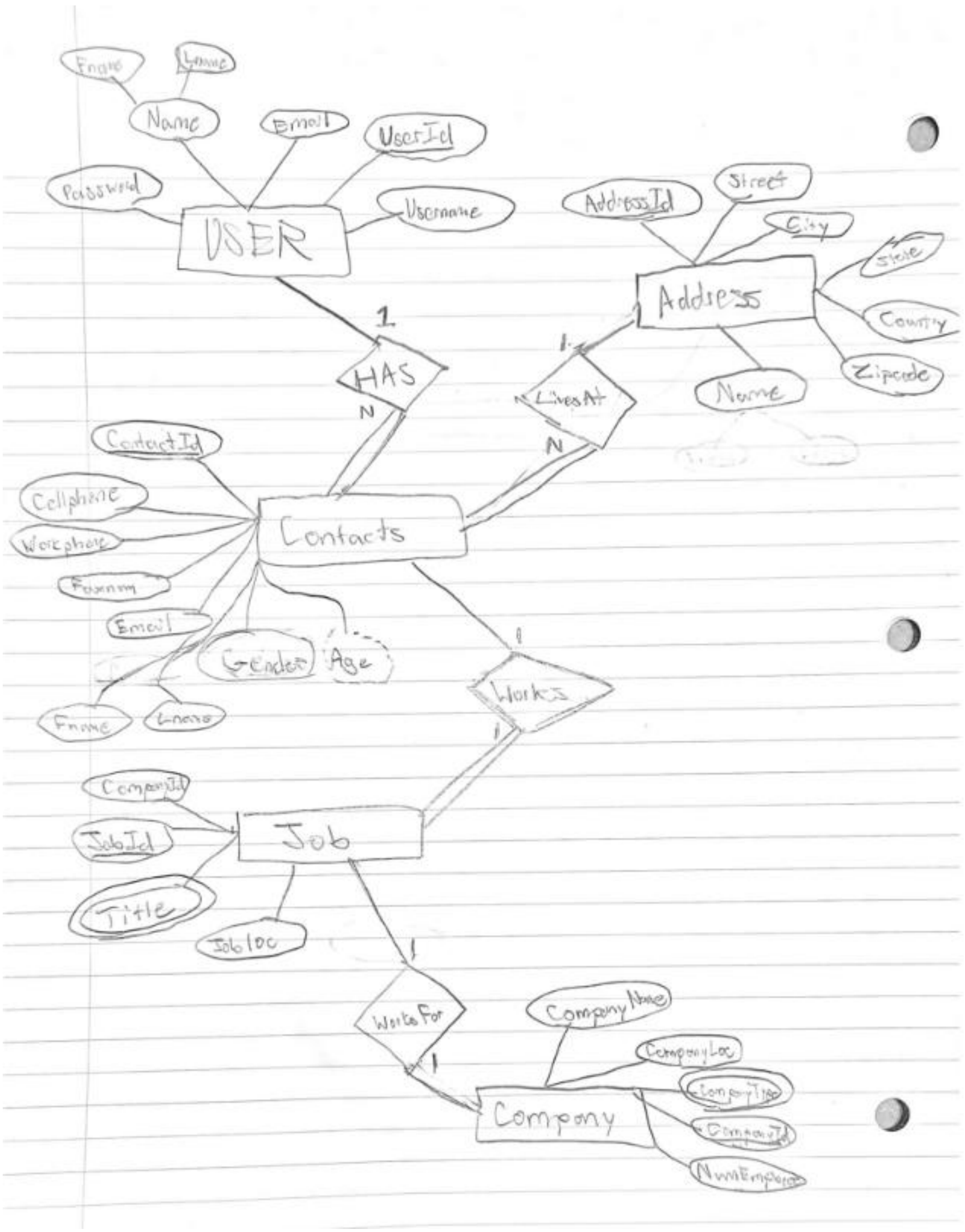


Figure 1 - User ER model

Admin ER Model - Provide a short description of each entity, attribute, relationship, participation, and cardinality.

Contacts is an entity with the attributes Fname, Lname, Cellphone, Workphone, FaxNum, Email, Gender, and Birthday. It stores the contact info for a person. Its function in this ER model is in the case an admin wants to access a list of Admin-related contacts.

Admin is an entity with the attributes Name, Admin_ID, Password, Email, and Username. It stores the info of Admins for the database system. It connects to all of the other entities other than contacts.

UserList is an entity with the attributes User_ID, username, name, email, contactsNum, and Date accessed. It stores the list of Users in the database and allows admins to change or view that information. It has a 1:1 relationship with Registration.

Registration is an entity with the attributes Name, RegID, Email, Password, and Username. This entity stores the list of users that can be added but aren't yet. The admin would have to go through and approve of the users in this list to be added to UserList. It has a 1:1 relationship with Admin.

AddNotice is an entity with the attribute DateTime, NoticeID, Note, Title, and Admin. This is an entity that stores the information for notices about the Contact management system. It has a 1:N relationship with Admin as an admin could have many notices.

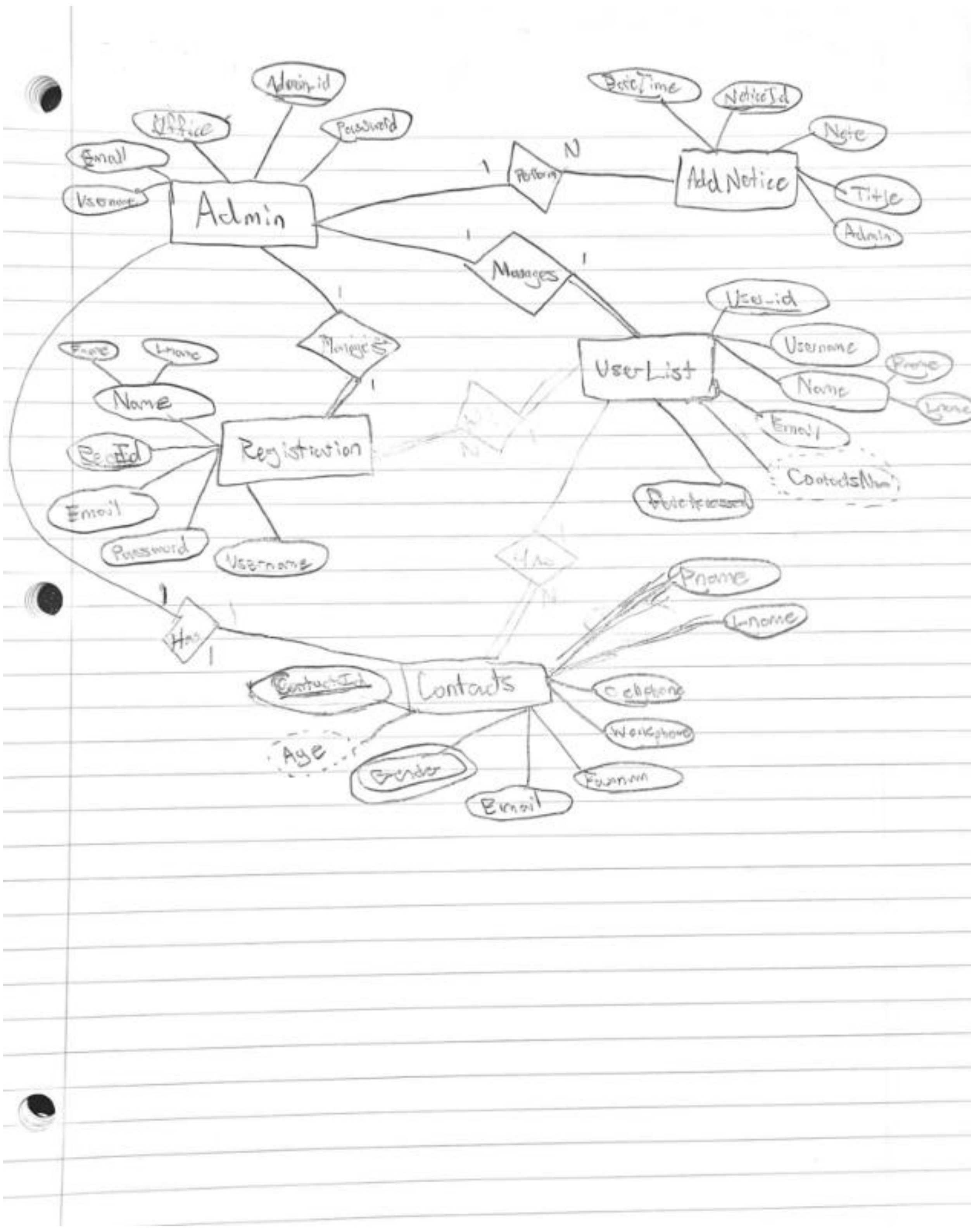


Figure 2 - Admin ER model

Description of Conceptual Model

This conceptual model provides a high-level view of the key elements and relationships within the Contact Management System, serving as a foundation for developing the system's detailed design and implementation. It helps people understand the system's structure and functionality without delving into technical specifics. The model provides a basic view of what the contact management system will end up looking like.

Interaction and the Notes entities are the big new entities. They are meant to keep tabs on interactions made between contacts, such as calls, and to keep notes on the interaction. Interaction has the attributes InteractionID, Date, Itype, AssociatedContact, and Company. It is connected to contacts as many to one contact. Notes has noteID, Ntype, NOTE, Date, and InteractionID. It is connected to Interactions in 1:1 and is meant to keep track of the interactions individually.

Relationships is an entity that is connected 1:1 to Contacts. It has the attributes RelationshipID, Fname, Lname, Rtype, Fname2, Lname2. It is an entity meant to keep track of personal relationships between people. Groups is a similar entity that keeps track of group relationships rather than personal relationships. Groups has the attributes GroupID, GroupName, Gmembers, Gcompany, and Gtype. It is connected to contacts in M:N.

The rest of the model just combines the two pre-existing ER models into one conceptual model. Therefore, it is the same as the other two.

Provide a short description of keys and relationships.

Every entity has a key with most having it connected to another entity through a relationship. All of them are along the lines of "EnitityID," such as AdminID, UserID, ContactID, or JobID. Entities are typically related in the way of being able to modify another. Users can add contact info to contacts such as addresses, jobs, companies, and so on. Admins can add registration info to users and can modify their contacts and so on.

The Conceptual model also changes an oversite made in the original user model where job had a relationship with company. This was an error and now it has a relationship with department in a many-to-one relationship. Also, Admin does not have user_list like it does in the original model as that would not make sense in te conceptual model. Instead admin now can make admin notes for othe

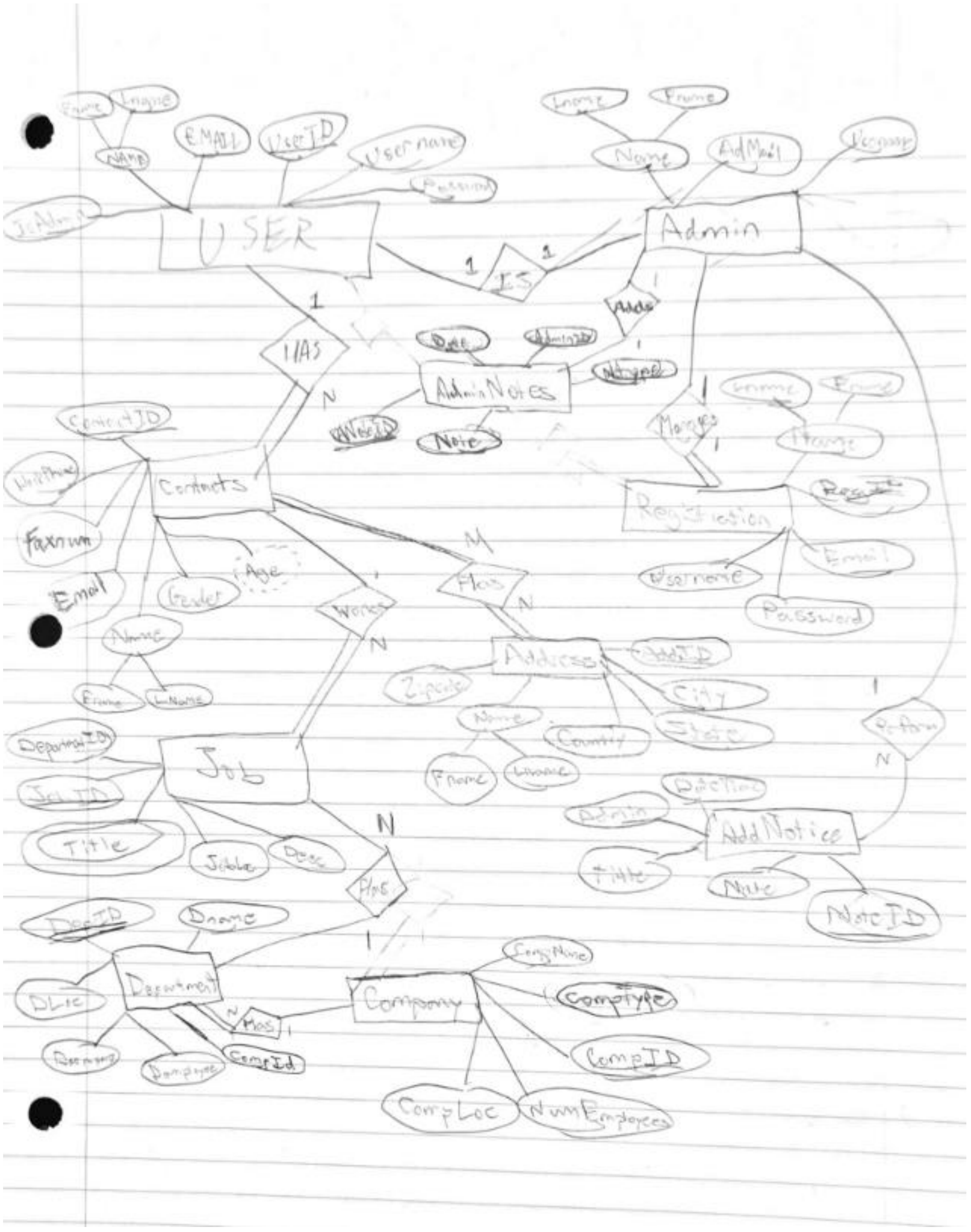


Figure 3 - Conceptual ER Model

Description of EER Diagram

Our EER diagram is a realization of our ER models and our conceptual model. It extends the capabilities of the ER model to a more usable form. It better represents the complex structure, inheritances, and specializations of our contact management system. It allows us to consider more intricate and precise relationships and hierarchies for our system. The EER diagram provides a structured representation of how contact data moves around our database. It ultimately provides a blueprint for our group to create and support a Contact Management database. It closely follows the conceptual diagram for the most part, aside from the absence of the Admin table. It has all of the same entities and provides the same relationships for the most part.

Describe how you implemented these features on your EER diagram.

I implemented these features accordingly into my EER model. The way I designed the contact management system meant the Conceptual and EER models are fairly similar. So for users and for the admin a lot of their capabilities line up similarly for them to come together in the EER model. One big change is that I moved the Admin functionalities to be an attribute of the user entity just to make the model a little tidier. Multivalued attributes are shown by having multiple columns in their entity. Composite attributes are shown by having a separate name.

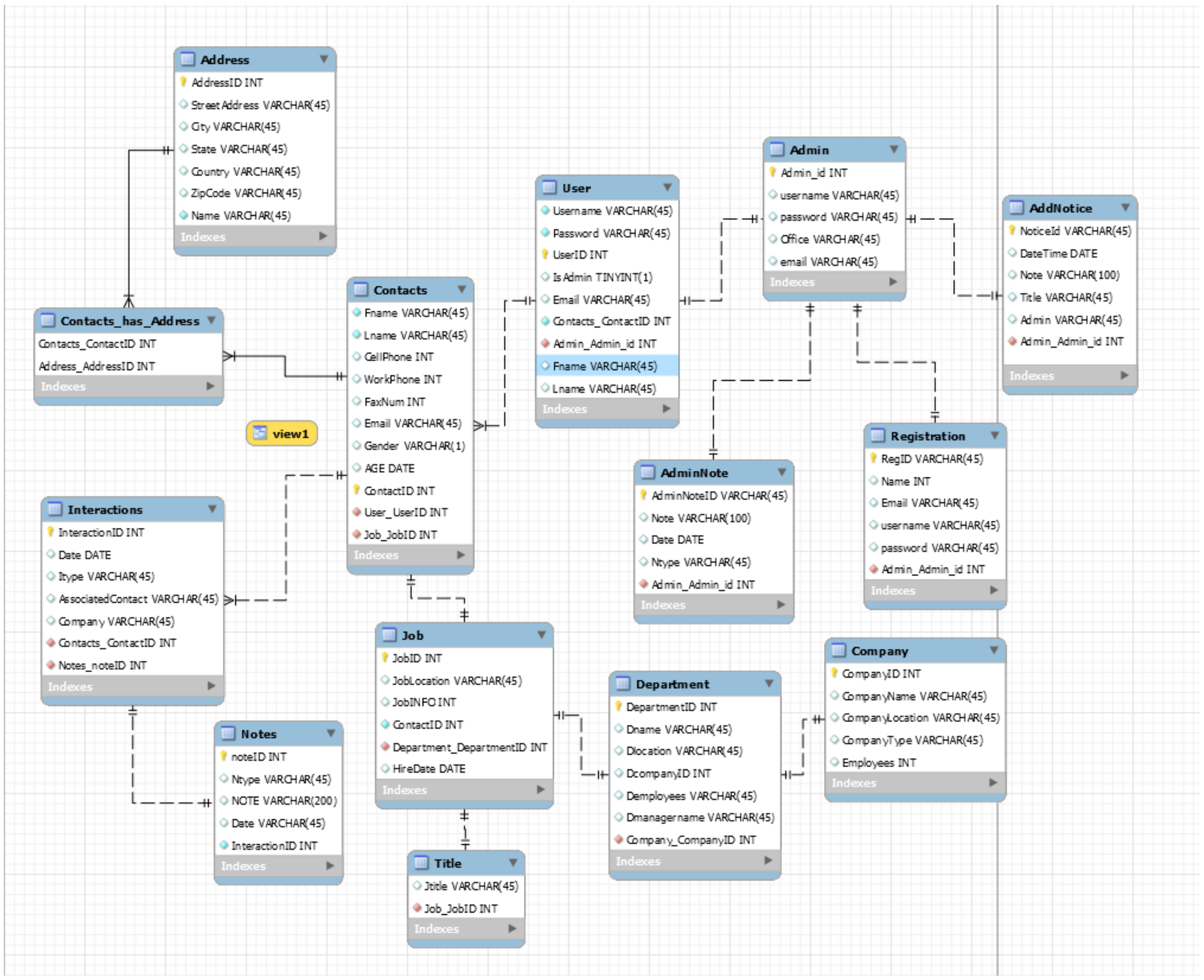
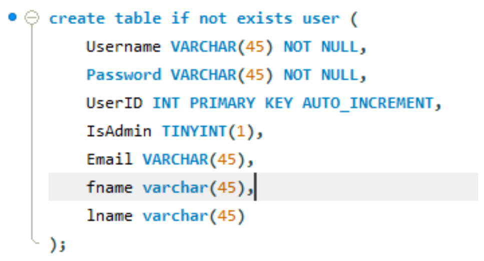


Figure 4 - EER Diagram

Section 3

Implementation of Our Database

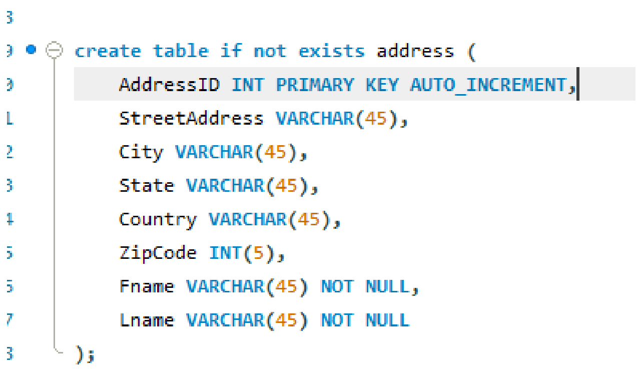
The user table consists of six attributes. Username and Password are both required VARCHAR fields. The UserID is an INT primary key with auto-increment. IsAdmin is a TINYINT indicating user privileges, and Email is a VARCHAR field for user email addresses. Fname and Lname are the attributes for the composite attribute Name in our conceptual diagram



```
create table if not exists user (
    Username VARCHAR(45) NOT NULL,
    Password VARCHAR(45) NOT NULL,
    UserID INT PRIMARY KEY AUTO_INCREMENT,
    IsAdmin TINYINT(1),
    Email VARCHAR(45),
    fname varchar(45),
    lname varchar(45)
);
```

Figure 5 - user table

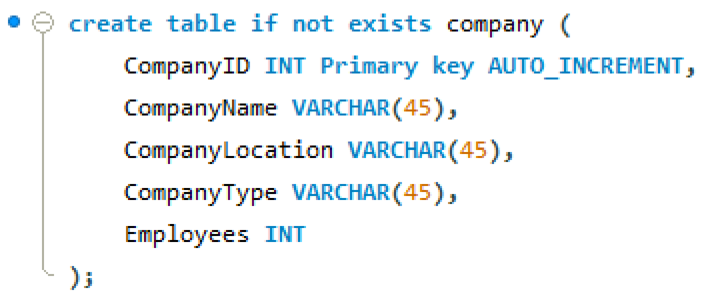
In the address table, there are seven attributes. AddressID is an INT primary key with auto-increment. StreetAddress, City, State, Country, and ZipCode are VARCHAR and INT fields representing address details. Fname and Lname are VARCHAR fields for first and last names, and they are required fields.



```
create table if not exists address (
    AddressID INT PRIMARY KEY AUTO_INCREMENT,
    StreetAddress VARCHAR(45),
    City VARCHAR(45),
    State VARCHAR(45),
    Country VARCHAR(45),
    ZipCode INT(5),
    Fname VARCHAR(45) NOT NULL,
    Lname VARCHAR(45) NOT NULL
);
```

Figure 6 - address table

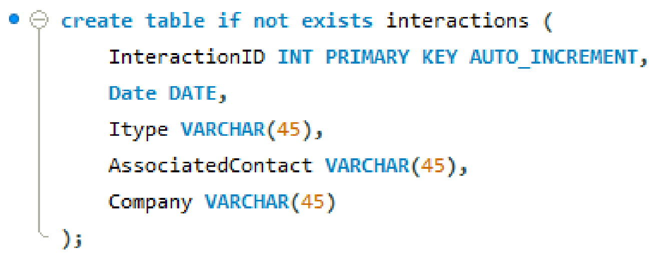
The company table comprises five attributes. CompanyID is an INT primary key with auto-increment. CompanyName, CompanyLocation, CompanyType, and Employees are VARCHAR and INT fields providing information about the company and its size.



```
create table if not exists company (
    CompanyID INT Primary key AUTO_INCREMENT,
    CompanyName VARCHAR(45),
    CompanyLocation VARCHAR(45),
    CompanyType VARCHAR(45),
    Employees INT
);
```

Figure 7 - company table

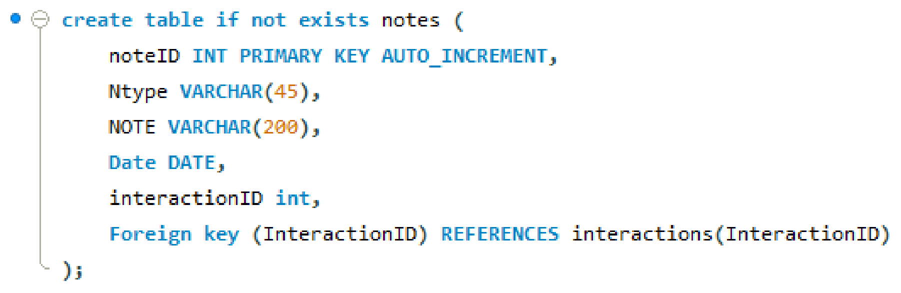
The interactions table contains five attributes. InteractionID is an INT primary key with auto-increment. The Date is a DATE field, and Itype is a VARCHAR field describing the type of interaction. AssociatedContact and Company are VARCHAR fields related to the interaction.



```
create table if not exists interactions (
    InteractionID INT PRIMARY KEY AUTO_INCREMENT,
    Date DATE,
    Itype VARCHAR(45),
    AssociatedContact VARCHAR(45),
    Company VARCHAR(45)
);
```

Figure 8 - interactions table

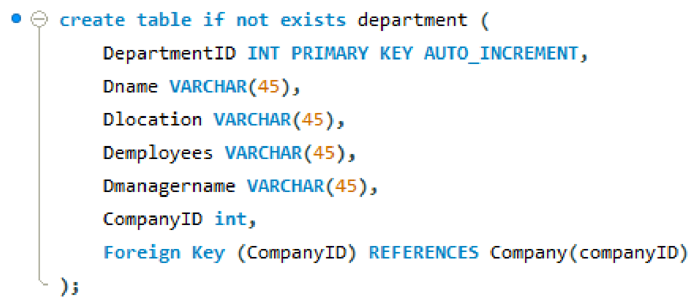
In the notes table, there are five attributes. noteID is an INT primary key with auto-increment. Ntype is a VARCHAR field describing the note type, and NOTE is a VARCHAR field with a maximum length of 200 characters. The Date is a DATE field, and interactionID is an INT field linking to the interactions table via a foreign key constraint.



```
create table if not exists notes (
    noteID INT PRIMARY KEY AUTO_INCREMENT,
    Ntype VARCHAR(45),
    NOTE VARCHAR(200),
    Date DATE,
    interactionID int,
    Foreign key (InteractionID) REFERENCES interactions(InteractionID)
);
```

Figure 9 - notes table

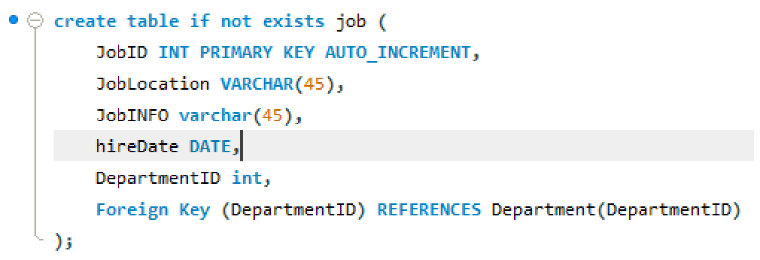
The department table has six attributes. DepartmentID is an INT primary key with auto-increment. Dname, Dlocation, Demployees, and Dmanagername are VARCHAR fields providing department details. The CompanyID is an INT field that references the Company table's CompanyID via a foreign key constraint.



```
create table if not exists department (
    DepartmentID INT PRIMARY KEY AUTO_INCREMENT,
    Dname VARCHAR(45),
    Dlocation VARCHAR(45),
    Demployees VARCHAR(45),
    Dmanagername VARCHAR(45),
    CompanyID int,
    Foreign Key (CompanyID) REFERENCES Company(companyID)
);
```

Figure 10 - department table

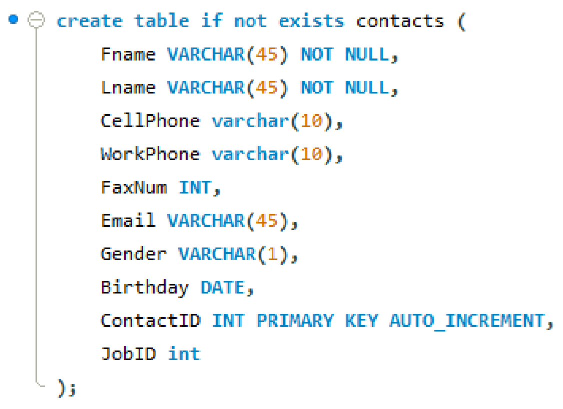
The job table contains seven attributes. JobID is an INT primary key with auto-increment. JobINFO, and JobLocation are VARCHAR fields related to job details. hireDate is a DATE field of ehwn the contact was hired. DepartmentID is an INT field that establishes a foreign key relationship with the Department table's DepartmentID.



```
create table if not exists job (
    JobID INT PRIMARY KEY AUTO_INCREMENT,
    JobLocation VARCHAR(45),
    JobINFO varchar(45),
    hireDate DATE,
    DepartmentID int,
    Foreign Key (DepartmentID) REFERENCES Department(DepartmentID)
);
```

Figure 11 - job table

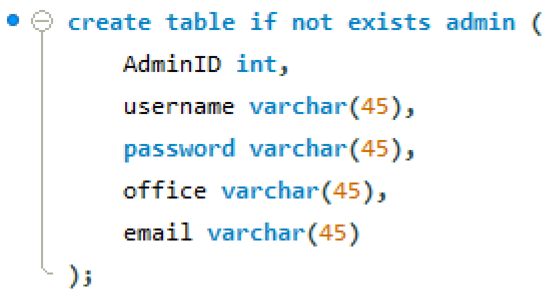
The contacts table encompasses nine attributes. Fname and Lname are required VARCHAR fields for first and last names. CellPhone and WorkPhone are VARCHAR fields for contact numbers, while FaxNum is an INT. Email is a VARCHAR field for email addresses, Gender is a VARCHAR field for gender, and Birthday is a DATE field. ContactID is an INT primary key with auto-increment, and JobID potentially relates to the job table.



```
create table if not exists contacts (
    Fname VARCHAR(45) NOT NULL,
    Lname VARCHAR(45) NOT NULL,
    CellPhone varchar(10),
    WorkPhone varchar(10),
    FaxNum INT,
    Email VARCHAR(45),
    Gender VARCHAR(1),
    Birthday DATE,
    ContactID INT PRIMARY KEY AUTO_INCREMENT,
    JobID int
);
```

Figure 12 - contacts table

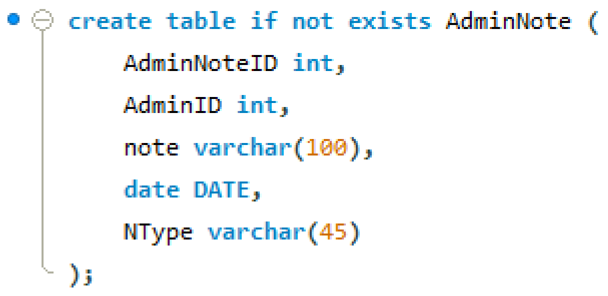
The admin table consists of five attributes: AdminID (INT), username (VARCHAR), password (VARCHAR), office (VARCHAR), and email (VARCHAR). These attributes hold information about administrators.



```
create table if not exists admin (
    AdminID int,
    username varchar(45),
    password varchar(45),
    office varchar(45),
    email varchar(45)
);
```

Figure 13 - admin table

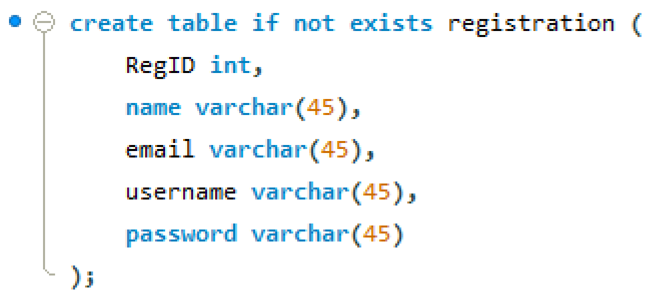
In the AdminNote table, there are five attributes: AdminNoteID (INT), AdminID (INT), note (VARCHAR), date (DATE), and NType (VARCHAR). These attributes store notes associated with administrators.



```
create table if not exists AdminNote (
    AdminNoteID int,
    AdminID int,
    note varchar(100),
    date DATE,
    NType varchar(45)
);
```

Figure 14 - admin note table

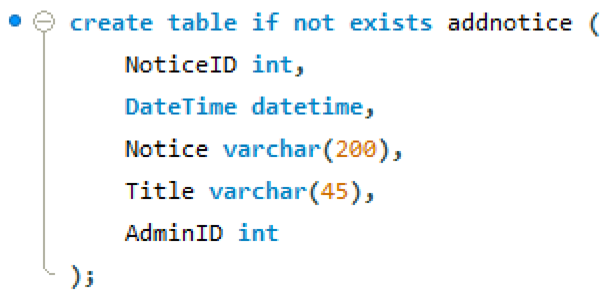
The registration table comprises five attributes: RegID (INT), name (VARCHAR), email (VARCHAR), username (VARCHAR), and password (VARCHAR). These attributes are used for user registration and authentication before they are added to the user table.



```
create table if not exists registration (
    RegID int,
    name varchar(45),
    email varchar(45),
    username varchar(45),
    password varchar(45)
);
```

Figure 15 - registration table

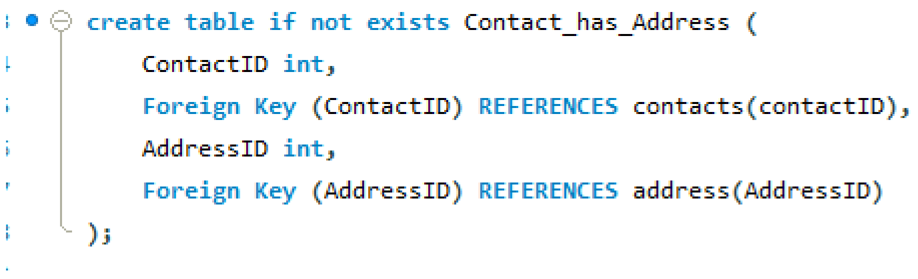
In the addnotice table, you will find five attributes: NoticeID (INT), DateTime (DATETIME), Notice (VARCHAR), Title (VARCHAR), and AdminID (INT). These attributes are used to record notices posted by administrators.



```
create table if not exists addnotice (
    NoticeID int,
    DateTime datetime,
    Notice varchar(200),
    Title varchar(45),
    AdminID int
);
```

Figure 16 - addnotice table

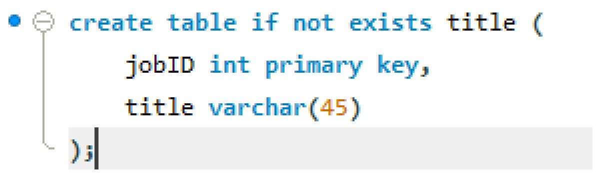
The Contact_has_Address table has two attributes: ContactID (INT) and AddressID (INT). These attributes establish a many-to-many relationship between contacts and addresses.



```
create table if not exists Contact_has_Address (
    ContactID int,
    Foreign Key (ContactID) REFERENCES contacts(contactID),
    AddressID int,
    Foreign Key (AddressID) REFERENCES address(AddressID)
);
```

Figure 17 - contact_has_address table

The title table has three attributes: JobID (INT), and title (VARCHAR). This table is used for storing the multivalued attribute title.

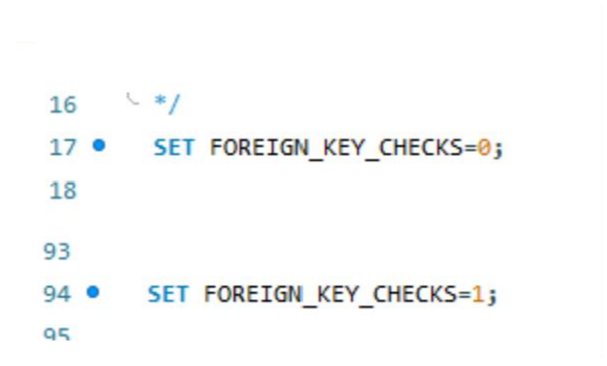


```
create table if not exists title (
    jobID int primary key,
    title varchar(45)
);
```

Figure 18 - title table

Section 4:

To handle foreign constraints we simply set FOREIGN_KEY_CHECKS to be equal to 0. This turns off MySQL's capability to check for foreign keys and allows us to add data to tables with disregard to them. This is a very simple way of adding data while circumventing the errors that may happen due to foreign keys. It is also much faster than removing the foreign keys attribute-by-attribute in each of the tables, then adding data, and then re-adding the foreign keys. This is a simple and effective way of side stepping the issues of foreign keys.

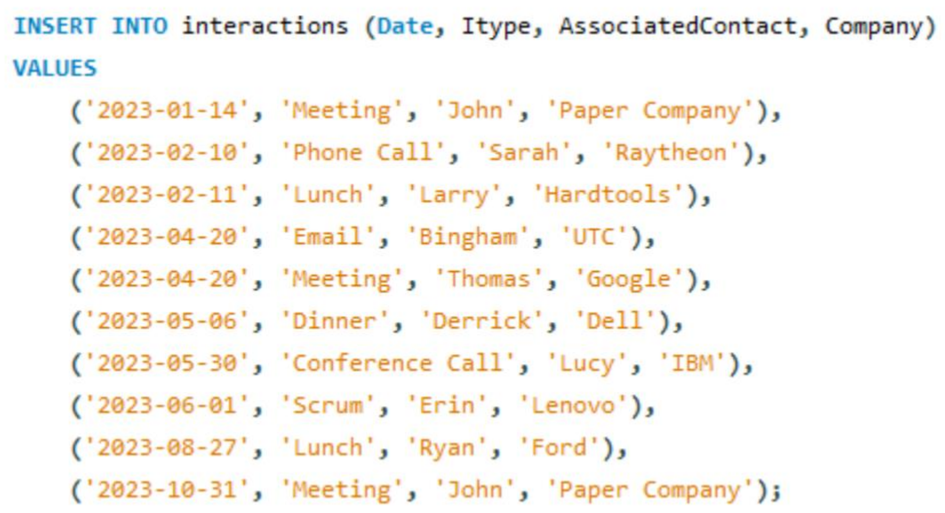


```
*/
SET FOREIGN_KEY_CHECKS=0;


SET FOREIGN_KEY_CHECKS=1;
```

Figure 19 - set foreign key statements

The best optimization practice is to use the LOAD DATA INFILE query. However, after some testing, that query doesn't ever work correctly, instead giving various errors. So, in its stead, we decided to instead go with the next best option available to us, compound insert into statements. This is considerably faster (many times faster in some cases) than using separate single-row INSERT statements. However, LOAD DATA is still much faster than using these insert into statements. LOAD DATA is usually 20 times faster than using INSERT statements. That's 2000% faster. However, due to errors in our work, this method does not work properly.

Inserting the data in one query through a multi-value insert statement, like the one shown in Figure 18, only took 0.016 sec. This applies all the data at once. Alternatively, doing a single insert statement for each piece of data takes 0.016 seconds to accomplish. To insert ten different statements would therefore take 0.16 seconds. This means that the multi-valued insert statement is 10x faster than the ten single insert statements.



```
INSERT INTO interactions (Date, Itype, AssociatedContact, Company)
VALUES
    ('2023-01-14', 'Meeting', 'John', 'Paper Company'),
    ('2023-02-10', 'Phone Call', 'Sarah', 'Raytheon'),
    ('2023-02-11', 'Lunch', 'Larry', 'Hardtools'),
    ('2023-04-20', 'Email', 'Bingham', 'UTC'),
    ('2023-04-20', 'Meeting', 'Thomas', 'Google'),
    ('2023-05-06', 'Dinner', 'Derrick', 'Dell'),
    ('2023-05-30', 'Conference Call', 'Lucy', 'IBM'),
    ('2023-06-01', 'Scrum', 'Erin', 'Lenovo'),
    ('2023-08-27', 'Lunch', 'Ryan', 'Ford'),
    ('2023-10-31', 'Meeting', 'John', 'Paper Company');
```

Figure 20 - a multi-valued insert into statement

Section 5:

Login Page: Gives access to Set Password for account creation and setting a new password for your username.

Input: Username and password

Output: Valid username and password enable the end user access to the

main menu. Invalid username or password cause showing a warning message of

invalid username and password. After 3 attempts the user will be sent back to the start and forced to create new password

Set Password Page:

Input: Password entered twice for confirmation

Output: Success updates the end user's password for login and displays that the password has been updated for the end user, failure redirects the end user back to the set password page.

Main Menu Page: Grants the end user access to Search Contact Page, Add Contact Page, Remove Contact Page, and access to exit the database application.

Input: Search Contact, Add Contact, Remove Contact, Exit.

Output: Search Contact -> redirects to Search Contact Page, Add Contact -> redirects to Add Contact Page, Remove Contact -> redirects to Remove Contact Page, Exit -> exits application redirects to login page.

Add Contact Page:

Input: Contact information userId, isAdmin, email, Fname, Lname.

Output: if the information entered is valid the contact will be added and the end user will be redirected to the Display Contact page where all contact information is displayed userid, isAdmin, email, Fname, Lname. The end user will then be asked if would like to add another contact if yes, they will be redirected back to the Add Contact Page, if no they will be redirected back to the main menu.

Remove Contact Page:

Input: userId

Output: if the information entered is a valid contact the contact will be removed and the end user will be shown that the contact has been removed, the end user will then be asked if they would like to remove another contact if yes they will be redirected back to the Remove Contact Page, if no they will be redirected back to the main menu.

Search Contact Page:

Input: userId

Output: the program will check if the user exists, the end user will be redirected to the Display Contact Page where they will see the userId, isAdmin, email, Fname, Lname of the userId that they just entered, then they will be asked if they would like to search another user if yes they will be redirected back to the search contact page, if no then they will be redirected to the main menu.

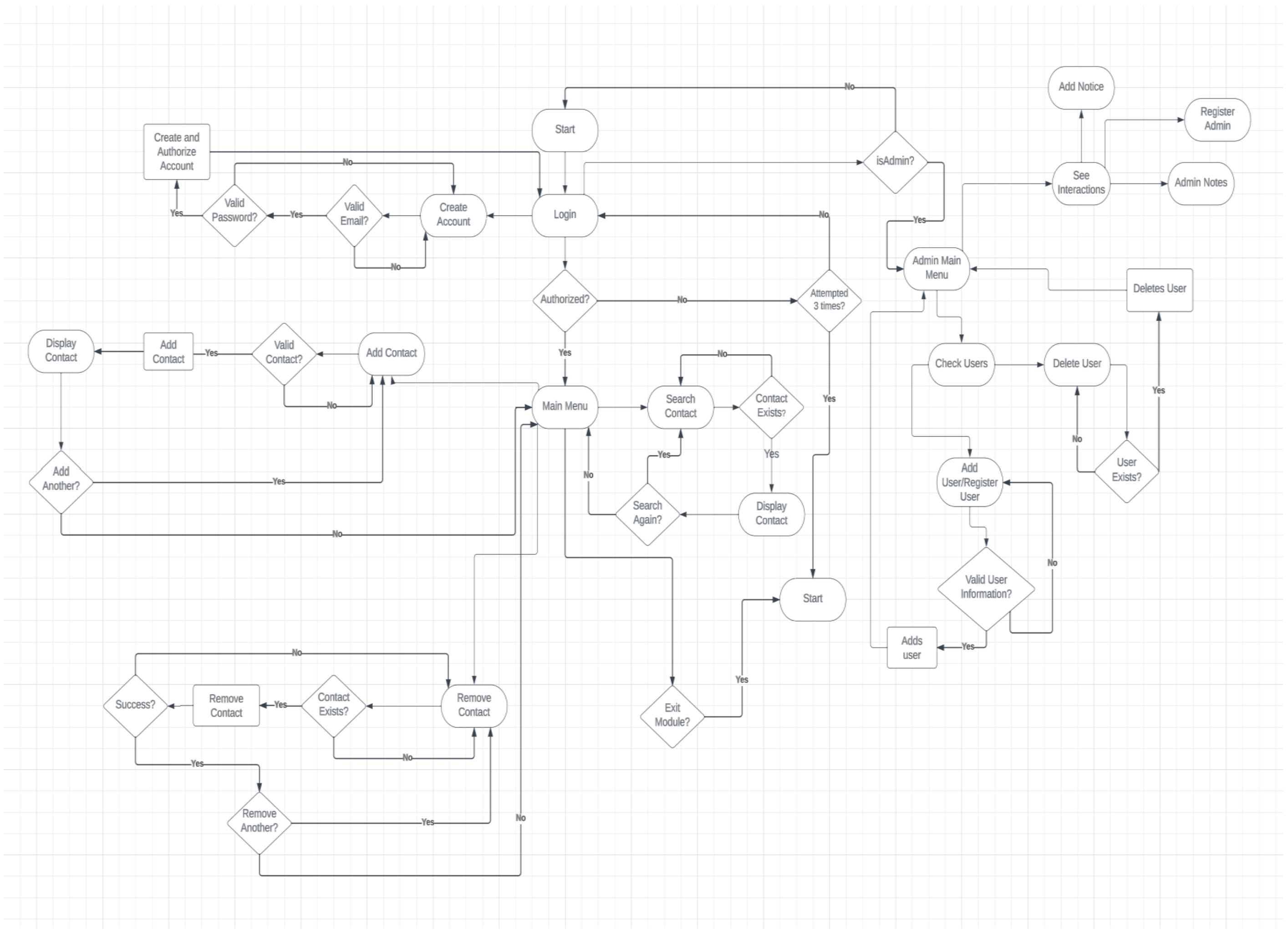
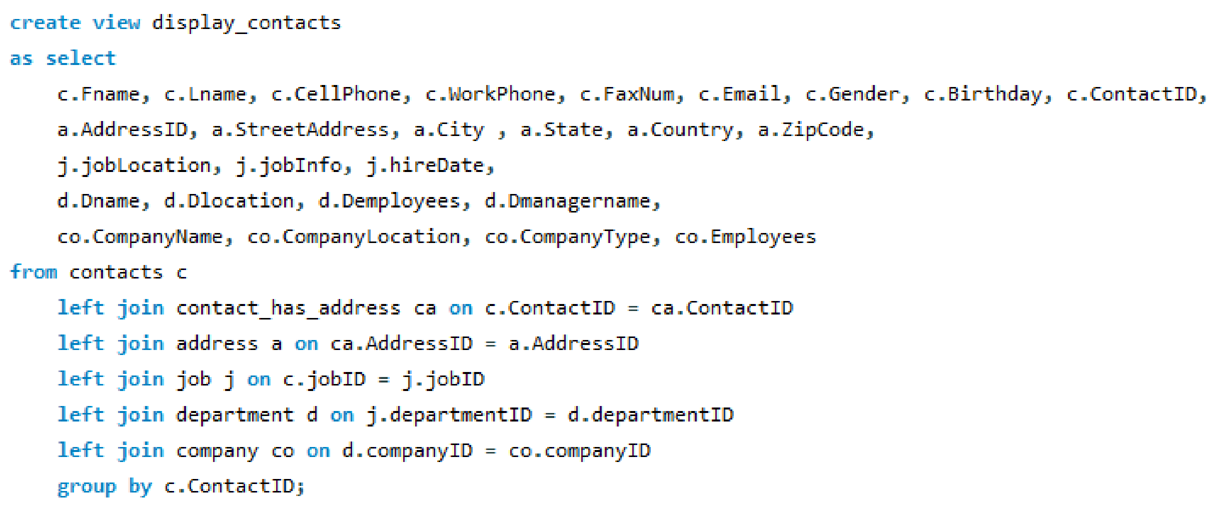


Figure 21 - GUI Flowchart

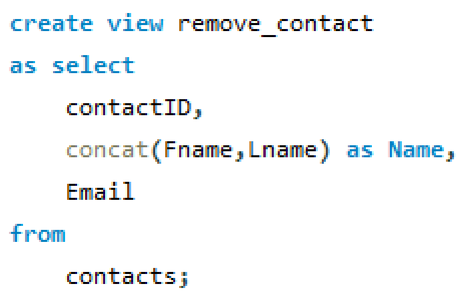
The display contacts page displays all of the contacts that the user has in the database management system. This view, display_contacts, displays all of the important information for each contact in the database. This includes their name, cellphone, email, gender, street address, city, state, and the company they work for.



```
create view display_contacts
as select
    c.Fname, c.Lname, c.CellPhone, c.WorkPhone, c.FaxNum, c.Email, c.Gender, c.Birthday, c.ContactID,
    a.AddressID, a.StreetAddress, a.City , a.State, a.Country, a.ZipCode,
    j.jobLocation, j.jobInfo, j.hireDate,
    d.Dname, d.Dlocation, d.Demployees, d.Dmanagername,
    co.CompanyName, co.CompanyLocation, co.CompanyType, co.Employees
from contacts c
    left join contact_has_address ca on c.ContactID = ca.ContactID
    left join address a on ca.AddressID = a.AddressID
    left join job j on c.jobID = j.jobID
    left join department d on j.departmentID = d.departmentID
    left join company co on d.companyID = co.companyID
    group by c.ContactID;
```

Figure 22 - Display Contacts View

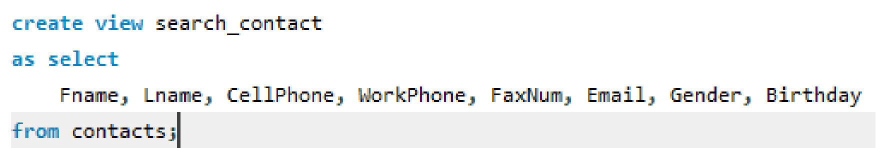
The remove contacts page displays all of the contacts in the database for a user so they can see what options they have if they want to remove a user. This view, remove_contact, displays just the name of the contact and the email. This keeps the page focused only on what's necessary for the user to remove.



```
create view remove_contact
as select
    contactID,
    concat(Fname,Lname) as Name,
    Email
from
    contacts;
```

Figure 23 - Remove contact view

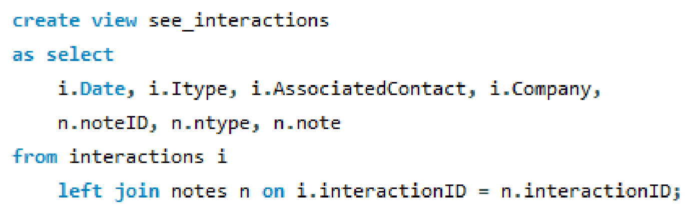
The search Contacts page shows you all the information of a contact from the contact table and nothing more.



```
create view search_contact
as select
    Fname, Lname, CellPhone, WorkPhone, FaxNum, Email, Gender, Birthday
from contacts;
```

Figure 24 - view search contact

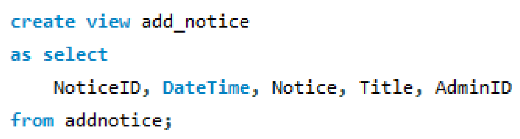
See interactions shows the user their interactions table and the notes that coincide with the interactions.



```
create view see_interactions
as select
    i.Date, i.Itype, i.AssociatedContact, i.Company,
    n.noteID, n.ntype, n.note
from interactions i
    left join notes n on i.interactionID = n.interactionID;
```

Figure 25 - view see_interactions

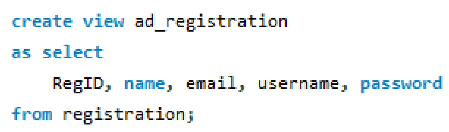
Add notice is for admins in case they want to add a notice.



```
create view add_notice
as select
    NoticeID, DateTime, Notice, Title, AdminID
from addnotice;
```

Figure 26 - view add_notice

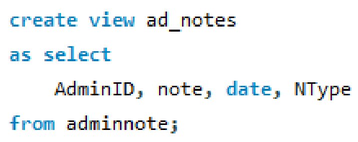
Ad_registration is for admins to be able to see what users are applying for registration. They can then make a decision based on whether they want to add or drop that user.



```
create view ad_registration
as select
    RegID, name, email, username, password
from registration;
```

Figure 27 - view for registration

Admin notes is for admins to leave themselves notes for later or for other admins.



```
create view ad_notes
as select
    AdminID, note, date, NType
from adminnote;
```

Figure 28 - admin notes view

Section 6:

Connection to Database:

```
        # Connect to MySQL database (replace these values with your database
credentials)
        self.connection = mysql.connector.connect(
            host="localhost",
            user="root",
            password="root",
            database="contact_manager_db"
        )
        self.cursor = self.connection.cursor()
```

Login Page:

The login page includes a logo picture that we made using a free logo maker on the internet, I imported this file in the code using the Tkinter PhotoImage Widget. This widget allows you to display pictures onto a tkinter screen. In the code I designed 2 functions, one for validating the login and one for getting the information from the user to login. The login() gets the username and password from the entered information in the entry widgets, it also validates the information to make sure both boxes have information entered. The validate_login() creates a dictionary containing the information needed to access the SQL database, then uses a try and except statement to try to query the user's table to check if the user exists. If the user exists, the login screen closes and the main menu class is fun. If not, the login fails and the database connection closes. Also added a hashing function to verify logi, and ensure user has entered a strong password.

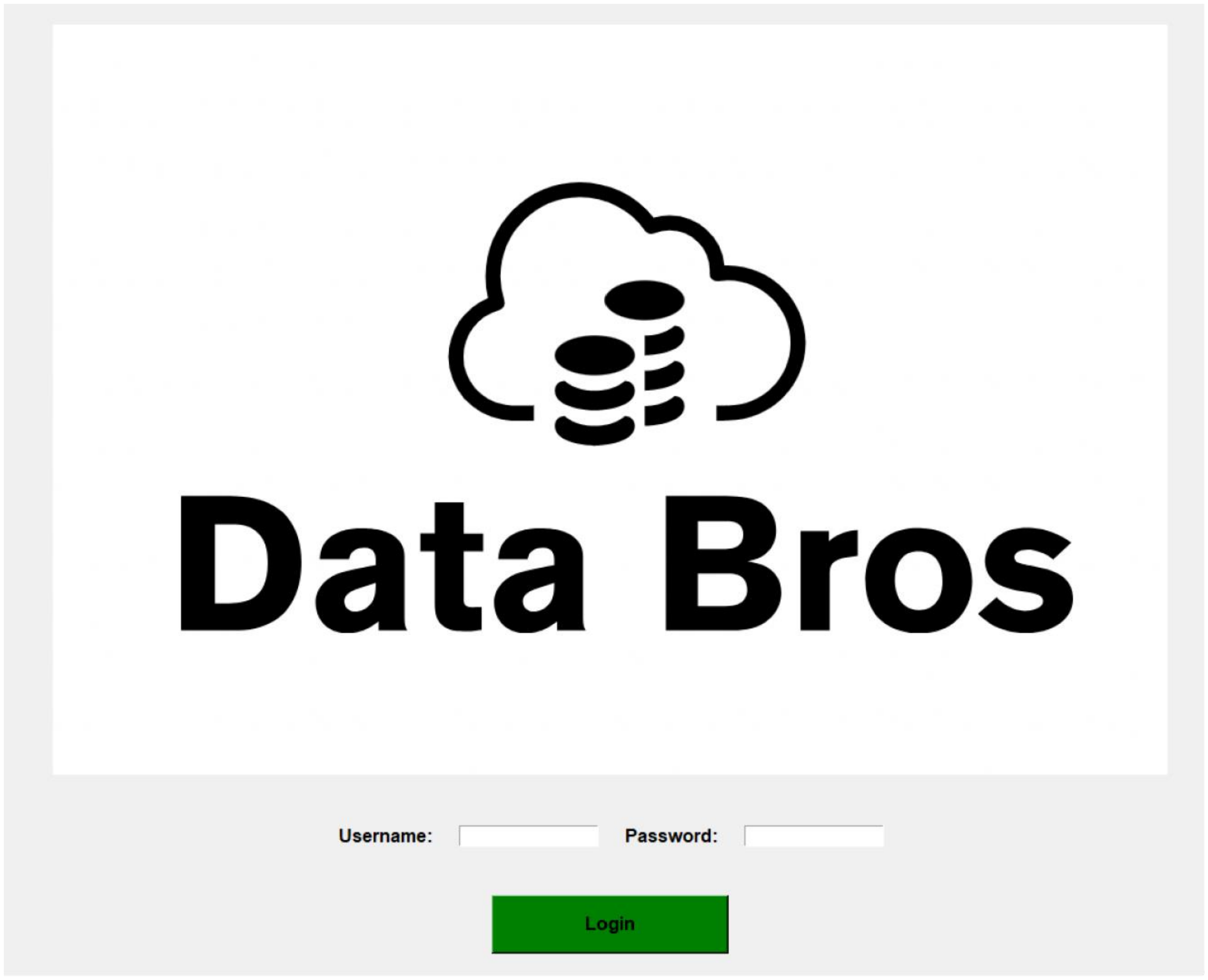


Figure 29: Gui Login Page

```
def validate_login(username, password, login_window):
    try:
        # Connection to MySQL database
        conn = mysql.connector.connect(
            host="localhost",
            user="root",
            password="root",
            database="contact_manager_db")
        cursor = conn.cursor()

        # SELECT query to check if the user exists
        query = "SELECT * FROM user WHERE username = %s"
        cursor.execute(query, (username,))
        user = cursor.fetchone()
        # Assuming the hashed password is stored in the second column
        if user and verify_password(password, user[1]):
            messagebox.showinfo("Login Successful",
                                "Welcome, {}".format(username))
            login_window.destroy()  # get rid of the login window
            main_menu_window = tk.Tk()
            app = MainMenu(main_menu_window)
            main_menu_window.mainloop()
        else:
            messagebox.showerror(
                "Login Failed", "Invalid username or password")

    except mysql.connector.Error as err:
        messagebox.showerror("Error", "MySQL Error: {}".format(err))

    finally:
        # Close the database connection
        if 'conn' in locals() and conn.is_connected():
            cursor.close()
            conn.close()


def verify_password(input_password, hashed_password):
    # Hash the input password and compare it with the stored hashed password
    hashed_input_password = sha256(input_password.encode()).hexdigest()
    return hashed_input_password == hashed_password


def login():
    username = entry_username.get()
    password = entry_password.get()

    # validation
    if not username or not password:
        messagebox.showwarning(
            "Incomplete Input", "Please enter both username and password.")
        return

    validate_login(username, password, root)
```

Main Menu Page:

This main menu has buttons for add contact, remove contact, display contacts, search, and user settings. The GUI is very simple and just displays what the user may need. Each button will take the user to their respective page. The class MainMenu initializes the title and root, and makes the connection to the mySql database, it then uses buttons and an execute_query to handle errors when changes occur in a SQL execution.

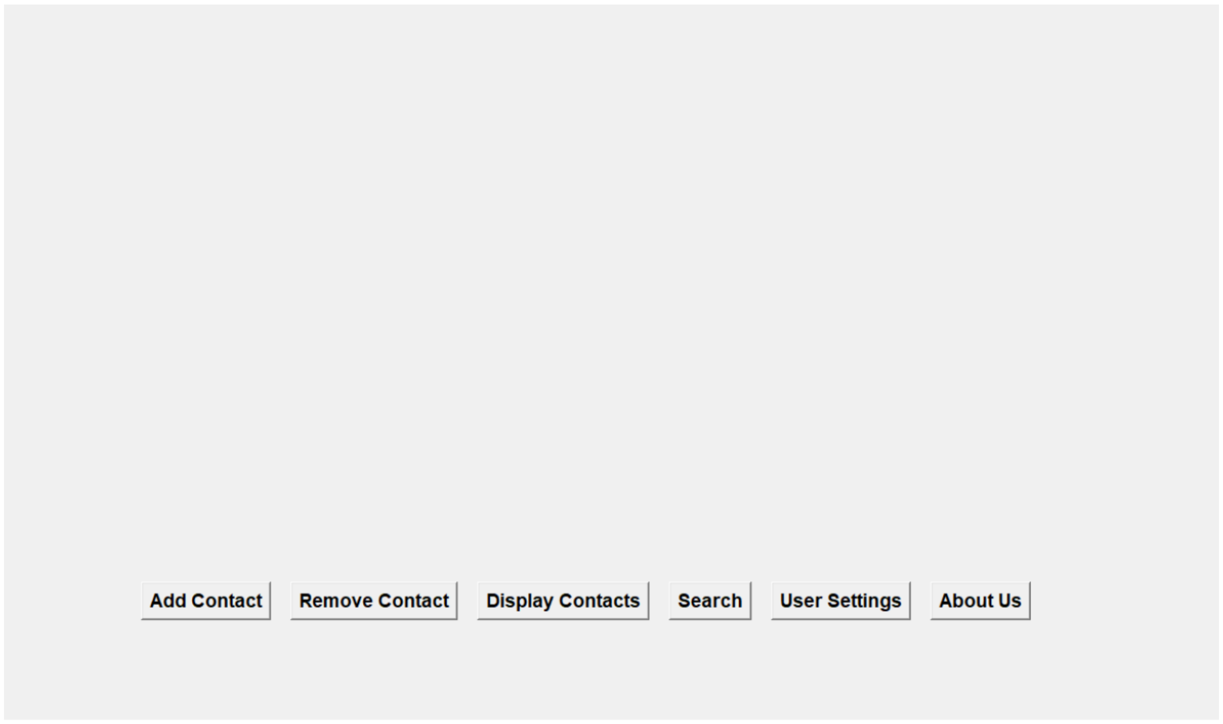


Figure 30: Main Menu Page

Add and Remove Contact Page:

The show_add_contact_page() will show the add contact page and allow users to enter information to add a contact to the DB it uses widgets to display the entry information and uses a lambda single line function to call the add_contact() with the entered information. The show_remove_contact_page() does the same thing but removes a contact based on contact ID. They also both include a back to main menu button that will close the window.

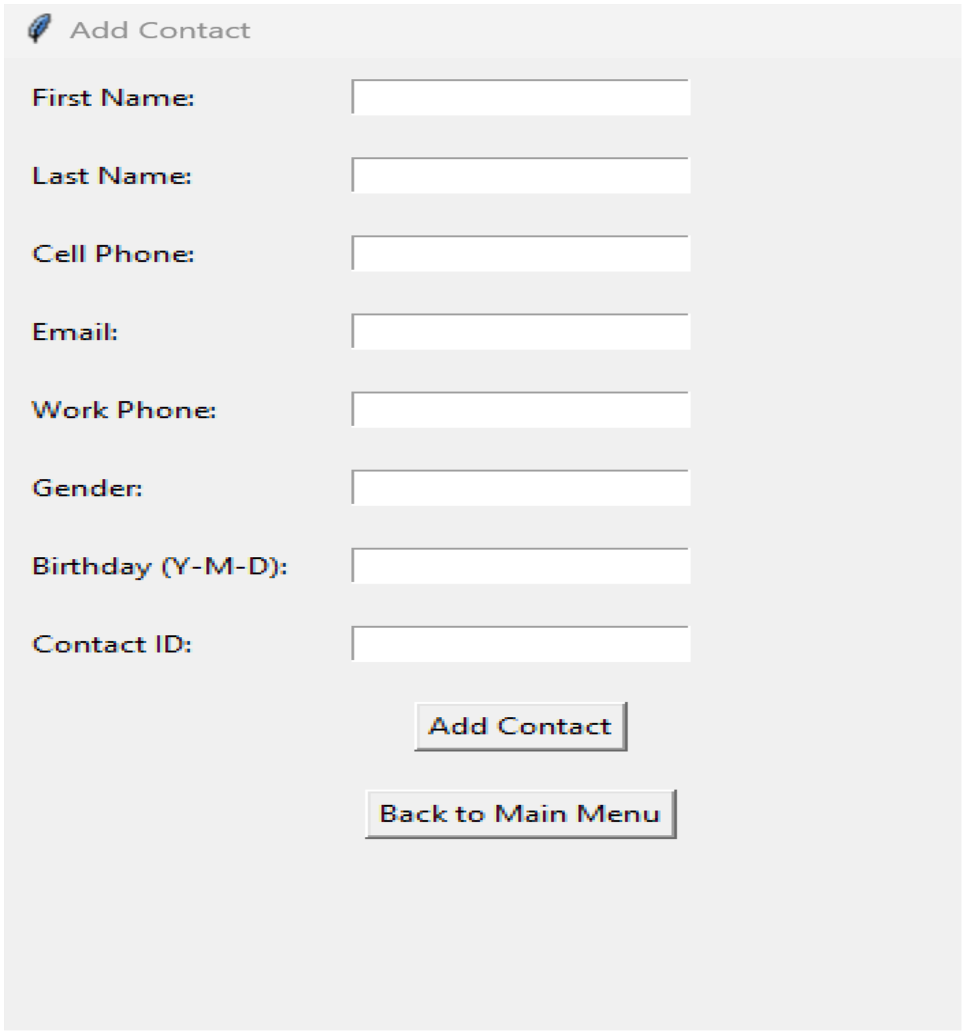


Figure 31: Add Contact Page

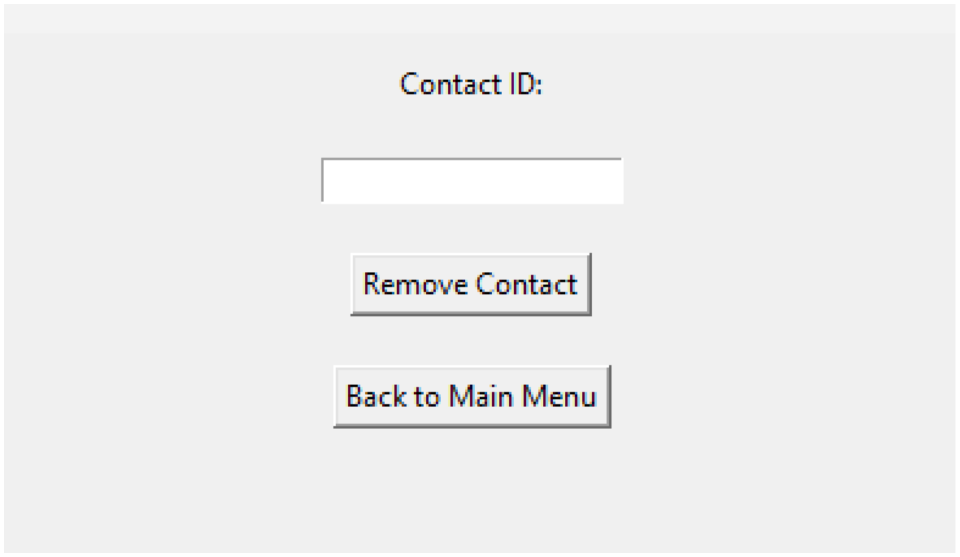
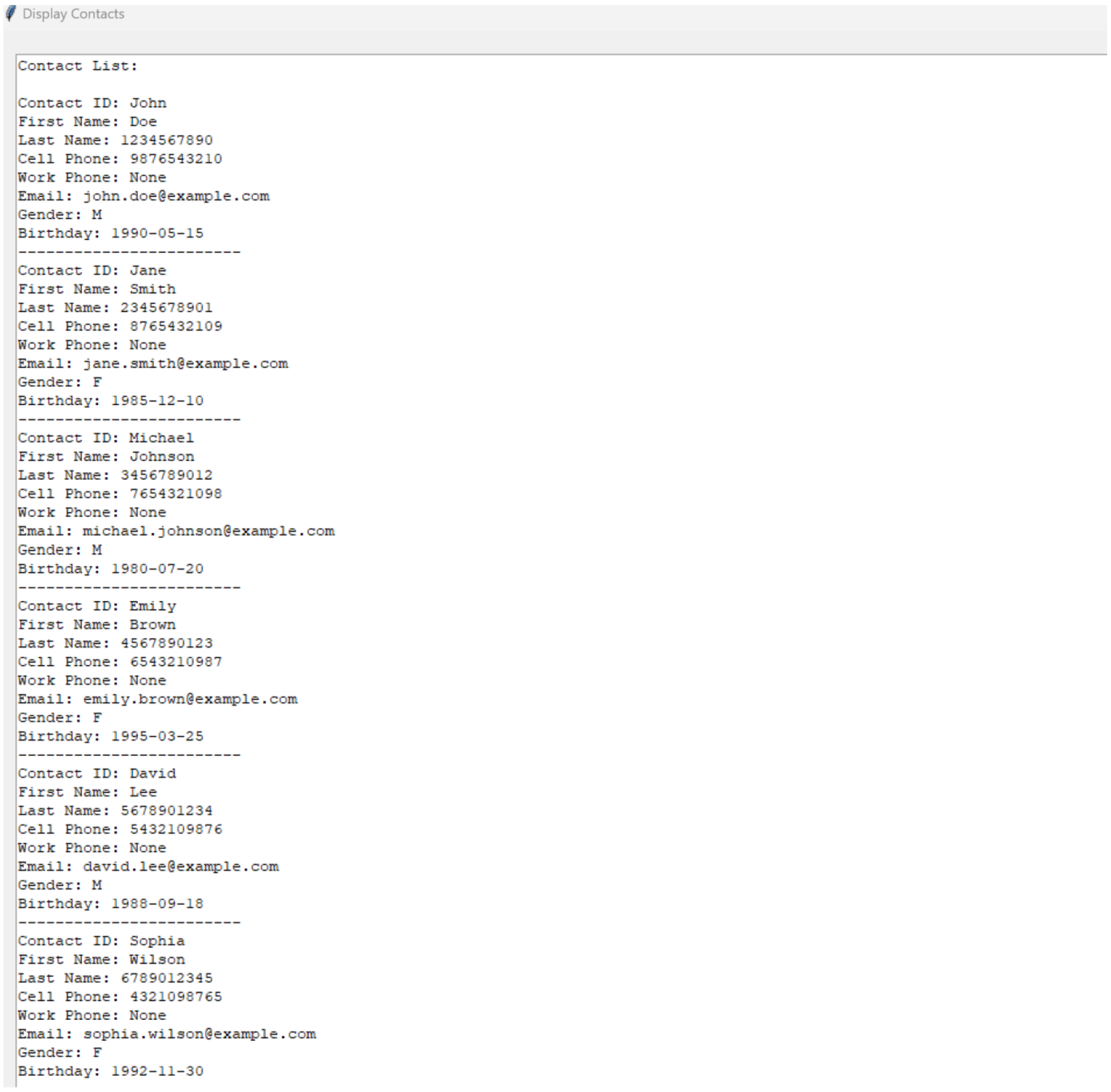


Figure 32: Remove Contact Page

```
import tkinter as tk
from tkinter import messagebox, PhotoImage
import mysql.connector


class MainMenu:
    def __init__(self, root):
        self.root = root
        self.root.title("Main Menu")

        # Connect to MySQL database (replace these values with your database
credentials)
        self.connection = mysql.connector.connect(
            host="localhost",
            user="root",
            password="root",
            database="contact_manager_db"
        )
        self.cursor = self.connection.cursor()

        # Create buttons
        buttons = [
            ("Add Contact", self.show_add_contact_page),
            ("Remove Contact", self.show_remove_contact_page),
            ("Display Contacts", self.display_contacts),
            ("Search", self.search),
            ("User Settings", self.user_settings),
        ]
        for text, command in buttons:
            button = tk.Button(root, text=text, command=command)
            button.pack(pady=10)

    def execute_query(self, query, data=None):
        try:
            self.cursor.execute(query, data)
            self.connection.commit()
            return True
        except mysql.connector.Error as err:
            messagebox.showerror("Error", f"MySQL Error: {err}")
            return False

    def show_add_contact_page(self):
        # new window for the Add Contact page
        add_contact_window = tk.Toplevel(self.root)
        add_contact_window.title("Add Contact")
        # place widgets for the Add Contact page
        label_first_name = tk.Label(add_contact_window, text="First Name:")
        entry_first_name = tk.Entry(add_contact_window)

        label_last_name = tk.Label(add_contact_window, text="Last Name:")
        entry_last_name = tk.Entry(add_contact_window)

        label_cell_phone = tk.Label(add_contact_window, text="Cell Phone:")
        entry_cell_phone = tk.Entry(add_contact_window)

        label_email = tk.Label(add_contact_window, text="Email:")
        entry_email = tk.Entry(add_contact_window)

        label_contactId = tk.Label(add_contact_window, text="Contact ID:")
        entry_contactId = tk.Entry(add_contact_window)

        btn_add_contact = tk.Button(
            add_contact_window, text="Add Contact", command=lambda:
self.add_contact(entry_first_name.get(), entry_last_name.get(),
entry_cell_phone.get(), entry_email.get(), add_contact_window))
        btn_back_to_menu = tk.Button(
            add_contact_window, text="Back to Main Menu",
command=add_contact_window.destroy)

        label_first_name.grid(row=0, column=0, padx=10, pady=10, sticky=tk.W)
        entry_first_name.grid(row=0, column=1, padx=10, pady=10)

        label_last_name.grid(row=1, column=0, padx=10, pady=10, sticky=tk.W)
        entry_last_name.grid(row=1, column=1, padx=10, pady=10)

        label_cell_phone.grid(row=2, column=0, padx=10, pady=10, sticky=tk.W)
        entry_cell_phone.grid(row=2, column=1, padx=10, pady=10)

        label_contactId.grid(row=3, column=0, padx=10, pady=10, sticky=tk.W)
        entry_contactId.grid(row=3, column=1, padx=10, pady=10)

        label_email.grid(row=3, column=0, padx=10, pady=10, sticky=tk.W)
        entry_email.grid(row=3, column=1, padx=10, pady=10)

        btn_add_contact.grid(row=4, column=1, pady=10)
        btn_back_to_menu.grid(row=5, column=1, pady=10)

    def show_remove_contact_page(self):
        # Create a new window for the Remove Contact page
        remove_contact_window = tk.Toplevel(self.root)
        remove_contact_window.title("Remove Contact")

        # Create and place widgets for the Remove Contact page
        label_id = tk.Label(remove_contact_window, text="Contact ID:")
        entry_id = tk.Entry(remove_contact_window)
        btn_remove_contact = tk.Button(
            remove_contact_window, text="Remove Contact",
command=self.remove_contact)
        btn_back_to_menu = tk.Button(
            remove_contact_window, text="Back to Main Menu",
command=remove_contact_window.destroy)

        label_id.pack(pady=10)
        entry_id.pack(pady=10)
        btn_remove_contact.pack(pady=10)
        btn_back_to_menu.pack(pady=10)

    def add_contact(self):
        # Get values from the entry widgets
        # Add logic Later
        name = entry_name.get()

        # Validate the input
        if not name:
            messagebox.showwarning("Incomplete Input", "Please enter a name.")
            return

        # Execute the INSERT query to add a new contact
        query = "INSERT INTO contacts (name) VALUES (%s)"
        data = (name,)
        if self.execute_query(query, data):
            messagebox.showinfo(
                "Success", f"Contact '{name}' added successfully.")
            add_contact_window.destroy()  # Close the Add Contact window

    def remove_contact(self):
        # Get values from the entry widgets
        # Add logic Later
        contact_id = entry_id.get()

        # Validate the input
        if not contact_id.isdigit():
            messagebox.showwarning(
                "Invalid Input", "Please enter a valid contact ID.")
            return

        # Execute the DELETE query to remove a contact
        query = "DELETE FROM contacts WHERE id = %s"
        data = (int(contact_id),)
        if self.execute_query(query, data):
            messagebox.showinfo(
                "Success", f"Contact with ID {contact_id} removed successfully.")
            remove_contact_window.destroy()  # Close the Remove Contact window

    def display_contacts(self):
        # Implement the logic for displaying contacts from the database
        messagebox.showinfo(
            "Display Contacts", "Implement the display contacts functionality
here.")

    def search(self):
        # Implement the logic for searching contacts in the database
        messagebox.showinfo(
            "Search", "Implement the search functionality here.")

    def user_settings(self):
        # Implement the logic for user settings
        messagebox.showinfo(
            "User Settings", "Implement the user settings functionality here.")
```

Display Contact Page:

The display contact page involves fetching all the data in the contact manager database contacts and printing them all to the Gui. On this page you can view all the contacts that are currently in the database, and it updates when a contact is added and removed. It is done by using the select * from contacts which gets all the data and contacts from the database and displays it to the display contact page.



```
Contact List:

Contact ID: John
First Name: Doe
Last Name: 1234567890
Cell Phone: 9876543210
Work Phone: None
Email: john.doe@example.com
Gender: M
Birthday: 1990-05-15
------------------------
Contact ID: Jane
First Name: Smith
Last Name: 2345678901
Cell Phone: 8765432109
Work Phone: None
Email: jane.smith@example.com
Gender: F
Birthday: 1985-12-10
------------------------
Contact ID: Michael
First Name: Johnson
Last Name: 3456789012
Cell Phone: 7654321098
Work Phone: None
Email: michael.johnson@example.com
Gender: M
Birthday: 1980-07-20
------------------------
Contact ID: Emily
First Name: Brown
Last Name: 4567890123
Cell Phone: 6543210987
Work Phone: None
Email: emily.brown@example.com
Gender: F
Birthday: 1995-03-25
------------------------
Contact ID: David
First Name: Lee
Last Name: 5678901234
Cell Phone: 5432109876
Work Phone: None
Email: david.lee@example.com
Gender: M
Birthday: 1988-09-18
------------------------
Contact ID: Sophia
First Name: Wilson
Last Name: 6789012345
Cell Phone: 4321098765
Work Phone: None
Email: sophia.wilson@example.com
Gender: F
Birthday: 1992-11-30
```

Figure 33: Display Contacts Page

```
def display_contacts(self):
        # Fetch all contacts from the database
        query = "SELECT * FROM contacts"
        self.cursor.execute(query)
        contacts = self.cursor.fetchall()

        # Create a new window for displaying contacts
        display_window = tk.Toplevel(self.root)
        display_window.title("Display Contacts")
        display_window.geometry("1920x1080")

        # Create a text widget to display contacts with a scroll bar
        display_text = tk.Text(display_window, height=30, width=100)
        display_text.pack(padx=20, pady=20, fill=tk.BOTH, expand=True)

        # I tried making a scroll bar but it wasnt working
        scrollbar = tk.Scrollbar(display_window)
        scrollbar.pack(side=tk.RIGHT, fill=tk.Y)

        # This still does nothing I tried configuirng it
        display_text.config(yscrollcommand=scrollbar.set)
        scrollbar.config(command=display_text.yview)

        # Format and display the contacts
        display_text.insert(tk.END, "Contact List:\n\n")
        for contact in contacts:
            display_text.insert(tk.END, f"Contact ID: {contact[0]}\n")
            display_text.insert(tk.END, f"First Name: {contact[1]}\n")
            display_text.insert(tk.END, f"Last Name: {contact[2]}\n")
            display_text.insert(tk.END, f"Cell Phone: {contact[3]}\n")
            display_text.insert(tk.END, f"Work Phone: {contact[4]}\n")
            display_text.insert(tk.END, f"Email: {contact[5]}\n")
            display_text.insert(tk.END, f"Gender: {contact[6]}\n")
            display_text.insert(tk.END, f"Birthday: {contact[7]}\n")
            display_text.insert(tk.END, "------------------------\n")

        # Disable text editing
        display_text.config(state=tk.DISABLED)

        # Function to go back to the main menu
        def back_to_main_menu():
            display_window.destroy()

        # Button to go back to the main menu
        btn_back_to_menu = tk.Button(
            display_window, text="Back to Main Menu", command=back_to_main_menu)
        btn_back_to_menu.pack(pady=10)
```

Search Page:

The search page allows users to search for a specific contact by searching with first and last names, when the first and last names are entered the program queries the database with those parameters and looks for a matching contact with the same first and last name, once found the program brings the user to a new page with the entire column of data for the specified contact.

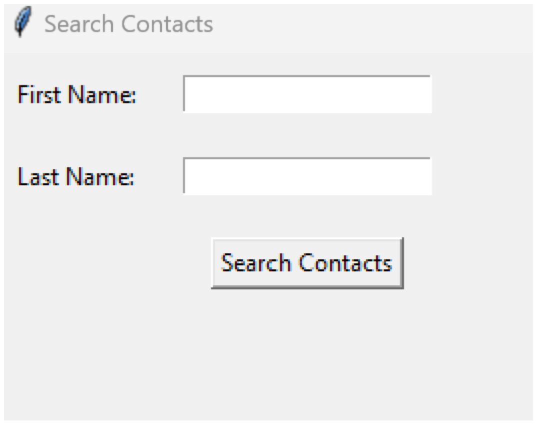
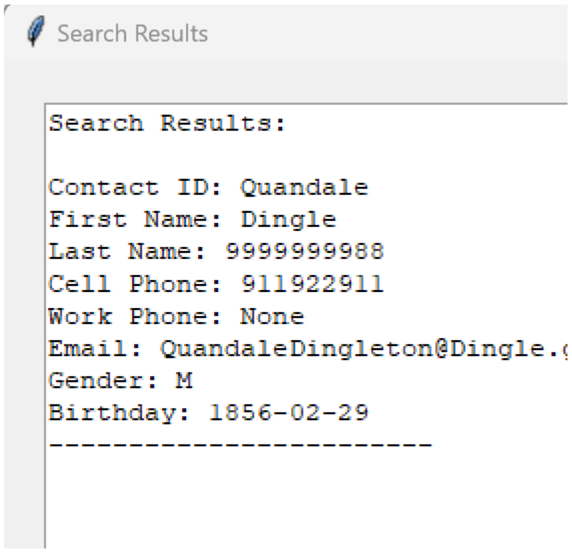


Figure 34: Search Page Gui



```
Search Results:

Contact ID: Quandale
First Name: Dingle
Last Name: 9999999988
Cell Phone: 911922911
Work Phone: None
Email: QuandaleDingleton@Dingle.
Gender: M
Birthday: 1856-02-29
------------------------
```

Figure 35: Search Results Page

```
def search_contacts(self, first_name, last_name, search_window):
        # Execute the SELECT query to search contacts
        query = "SELECT * FROM contacts WHERE Fname = %s AND Lname = %s"
        data = (first_name, last_name)
        self.cursor.execute(query, data)
        contacts = self.cursor.fetchall()

        # Create a new window for displaying search results
        search_result_window = tk.Toplevel(search_window)
        search_result_window.title("Search Results")
        search_result_window.geometry("1920x1080")

        # Create a text widget to display search results with a scroll bar
        search_result_text = tk.Text(search_result_window, height=20, width=60)
        search_result_text.pack(padx=20, pady=20, fill=tk.BOTH, expand=True)

        # Scrollbar for search result text
        scrollbar = tk.Scrollbar(search_result_window)
        scrollbar.pack(side=tk.RIGHT, fill=tk.Y)
        search_result_text.config(yscrollcommand=scrollbar.set)
        scrollbar.config(command=search_result_text.yview)

        # Format and display the search results
        search_result_text.insert(tk.END, "Search Results:\n\n")
        for contact in contacts:
            search_result_text.insert(tk.END, f"Contact ID: {contact[0]}\n")
            search_result_text.insert(tk.END, f"First Name: {contact[1]}\n")
            search_result_text.insert(tk.END, f"Last Name: {contact[2]}\n")
            search_result_text.insert(tk.END, f"Cell Phone: {contact[3]}\n")
            search_result_text.insert(tk.END, f"Work Phone: {contact[4]}\n")
            search_result_text.insert(tk.END, f"Email: {contact[5]}\n")
            search_result_text.insert(tk.END, f"Gender: {contact[6]}\n")
            search_result_text.insert(tk.END, f"Birthday: {contact[7]}\n")
            search_result_text.insert(tk.END, "------------------------\n")

        # Disable text editing
        search_result_text.config(state=tk.DISABLED)

        # Function to go back to the search window
        def back_to_search():
            search_result_window.destroy()

        # Button to go back to the search window
        btn_back_to_search = tk.Button(
            search_result_window, text="Back to Search", command=back_to_search)
        btn_back_to_search.pack(pady=10)
```

References:

(1) "Google Contacts." *Google Contacts Software Reviews, Demo & Pricing - 2023*, www.softwareadvice.com/crm/google-contacts-profile/. Accessed 25 Oct. 2023.
(2) "Customer Relationship Management Definition - Salesforce Us." *Salesforce*, www.salesforce.com/crm/what-is-crm/. Accessed 25 Oct. 2023.
(3) "WhatCMS - Microsoft Excel." *What CMS?*, whatcms.org/c/Microsoft-Excel. Accessed 25 Oct. 2023.