Task Manager

Database Management

308N-201

MaristTask

Marist College
School of Computer Science and Mathematics

Submitted to:
Dr. Reza Sadeghi

9/18/2023

Project Report of Red Fox Task

Team Name

RedFoxTask

Team Members

1. Kevin Reiff Kevin.Reiff1@marist.edu Team Member

Description of Team Members

1. Kevin Reiff - I am a computer science major from Vernon NJ. I have been working as an intern for the IT Security Department at Selective Insurance since the beginning of the summer and I am still working for them throughout the semester. I wanted to work with these teammates because I knew two of them previously and we all share similar interests. We chose the team head by discussing out-of-class responsibilities and who was best equipped to take on the responsibility.

Table of Contents

Project Objective

Project Title: MaristTask

The MaristTask is a task management system that will display a calendar for the selected day, week, month, or year. It organizes the specific tasks of different users each day, and users can update these tasks at any time. The calendar should update in day, week, month, and year form when a task is added. The task management system will store the data of different users in distinct SQL tables.

This task management system includes multi-level user authentication, a dynamic calendar display, personalized task management, an advanced search function, a user-friendly interface, data reporting, alerts and warnings, and an easy exit function.

The multi-level user authentication gives access to the system to both users and admins. Users can manage their task pages, while admins have access to security pages as well as the pages of regular users. The calendar display gives users options to choose their preference when it comes to listing their tasks, whether that is in an annual, monthly, weekly, or daily view. The personalization of the task manager continues, as users can edit each part of the tasks they create, all the way down to the color it displays. The task management system includes an advanced search function, where users can search within their tasks using a variety of parameters. All of these features are presented in a user-friendly interface that is easy to maneuver and aesthetically pleasing to the eye. For easier functionality, an alert system is integrated that will alert users if there are overlapping dates or tasks. Users can easily exit the task management system with a simple logout option.

Related Work Review:

Trello - A good task manager that is used for business. It uses all-around good software, where you can keep track of tasks done, upcoming tasks, tasks to do, and tasks you are in the middle of completing[3]. However, Trello does not have a way to directly message people if something needs to be done. This would help a lot in our application.

Jira - Another good task manager that uses a timeline to create tasks that you have to do. It shows the start and end date on a type of calendar that goes throughout months[2]. However, it is a bit too confusing. You can make teams, but the tasks are confusing and awkward to look at. I think they should make it simpler and more user-friendly to make people want it.

Todoist - Another great task managing software that is used by huge companies such as Disney, Microsoft, Amazon, and Netflix. This offers an easy-to-manage and visually pleasing to-do list of everything you need to complete. It shows personal tasks for yourself, team tasks if you have a team company you are working with, and different workspaces to list your groceries, fitness routines, appointments, and personal goals. You can also join teams to create joint tasks[1]. Just like with Trello, what would make this better is a messaging software within it to contact your team at any time.

Project Merits

1. Core Capabilities:
    a. **Multi-Level User Authentication:** Both admin and standard user roles are available in our task-managing system, each with unique rights. This improves security and manageability by ensuring that the appropriate users have access to the appropriate functionality. Unlike Trello and Todoist, our task managing system includes a dedicated admin role for enhanced security and user management.
    b. **Dynamic Calendar Display:** Users may choose between weekly, monthly, and annual views, giving them the flexibility to plan and arrange work as needed. While Jira provides a timeline, our task managing system offers a more straightforward calendar display that is easier to interpret.
    c. Personalized Task Management: Tasks particular to each account can be added, modified, or removed by the user. Each assignment has necessary details including the title, start and end times, as well as a description. Unlike Jira, our task managing system aims for simplicity and user-friendliness in task management.
    d. **Advanced Search Functionality:** Users have the option of searching for tasks using several parameters, including duration, title, and time. This makes finding certain activities simple and quick. This search functionality exceeds the capabilities of Trello, which lacks an advanced search feature based on multiple parameters.
    e. **User-Friendly Interface:** Our task managing system will have an initial welcome page as well as a structured menu of all functions on each page, making it easy to use and intuitive. Unlike Jira, which can be confusing to new users, our task manager system focuses on user experience.
    f. **Data Reporting:** To assist users in their responsibilities in a structured fashion, the system can create tabular reports that include well-organized calendars. Compared to Todoist, this function is more comprehensive.
    g. **Alerts and Warnings:** Our system will notify users anytime they try to input contact information that already exists in the database to prevent repetition. This feature is not common in the other task management systems mentioned.
    h. **Graceful Exit:** The user experience is improved via a special exit function that thanks the user for using the program.
2. Why Choose Our Product?
    a. **Highly Customizable:** Our task managing system may be customized to meet individual needs thanks to the option to switch between calendar views and alter task information.

b. **Secure User Management:** Our task managing system prioritizes data security and will have separate SQL databases for various user categories as well as strict password requirements.
c. **Efficiency:** Users can better manage their time and responsibilities with the help of the sophisticated search capabilities and tabular reports that our system will have.
d. **User-Centric Design**: Even individuals with rudimentary technological knowledge can utilize the system with ease because of its user-friendly interface.
e. **Data integrity:** The built-in alter system that we will implement into our task managing system will prevent duplicate entries, maintaining the reliability of the stored data.
f. **Comprehensive Solution:** Our task managing system will provide a one-stop shop for all task management requirements, covering task creation, reporting, and even user administration (for admins).

GitHub Repository Address

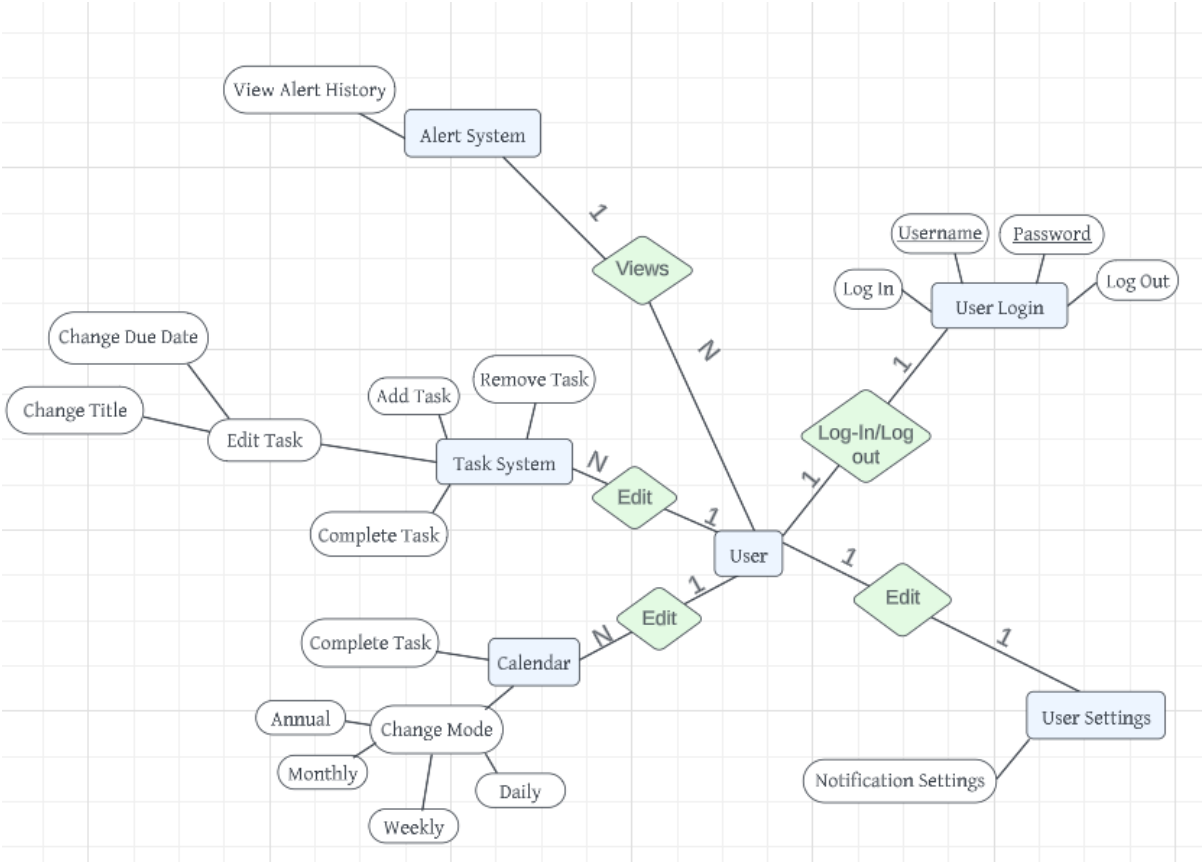
GitHub Repository:
https://github.com/KevinReiff/CMPT308_TaskManager_MaristTask/tree/main

Phase 2: ER Model and EER Model:

External Model 1: User

Users are central to various operations and can engage with multiple subsystems. The "User Login" subsystem is responsible for authenticating users through the "Username" and "Password" fields, allowing them to "Log In" and "Log Out". Once logged in, users can manage their "User Settings" including "Notification Settings", enabling personalized configurations of the application. The "Task System" is a crucial component of this model. It enables users to "Add Task" to track their work or personal items. Tasks can be "Edited" to change details like "Change Due Date" or "Change Title". Moreover, users can "Remove Task" when a task is no longer relevant or has been completed erroneously. The "Complete Task" feature indicates that users can mark tasks as done, removing them from the active list or moving them to a completed list.

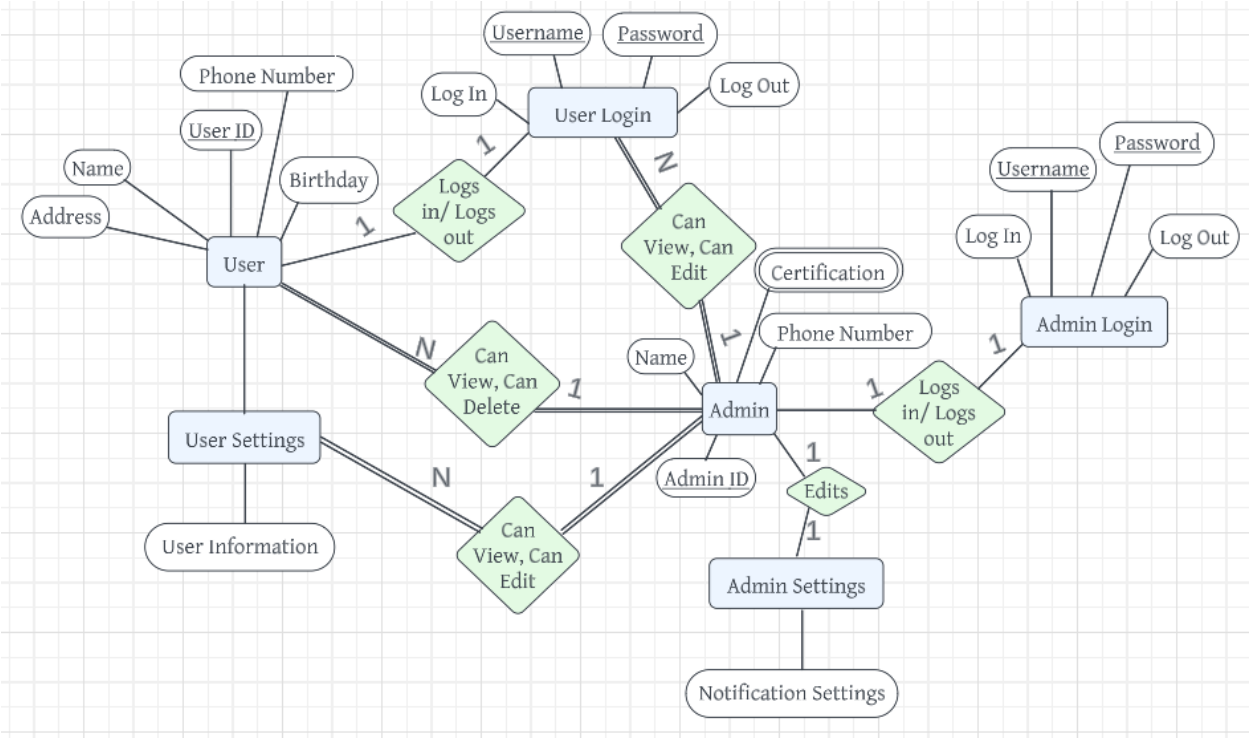
An "Alert System" complements task management by offering a "View Alert History", which suggests a logging system to review past notifications or reminders. Additionally, the model features a "Calendar" subsystem. This subsystem allows users to "Change Mode" to view tasks or events on a "Daily", "Weekly", "Monthly", or "Annual" basis, providing a comprehensive overview of the time-related aspects of tasks and events. The "Views" association, represents the some of information that is displayed to the user, showing that they can view the alert history.



External Model 2: Admin

In this system, the "Admin" acts as a specialized user with enhanced privileges. The "Admin Login" subsystem is similar to the "User Login" but is specifically designed for administrative access, requiring a username" and "Password" for authentication. The "Admin" can "Log In" and "Log Out", with the "Logs in/Logs out" use case indicating that there is tracking of the admin's session activity. Upon logging in, the "Admin" has access to "Admin Settings" and "Notification Settings," suggesting the ability to configure system settings and preferences for notifications at a higher level than regular users. The "Admin ID" is a unique identifier for administrative personnel.

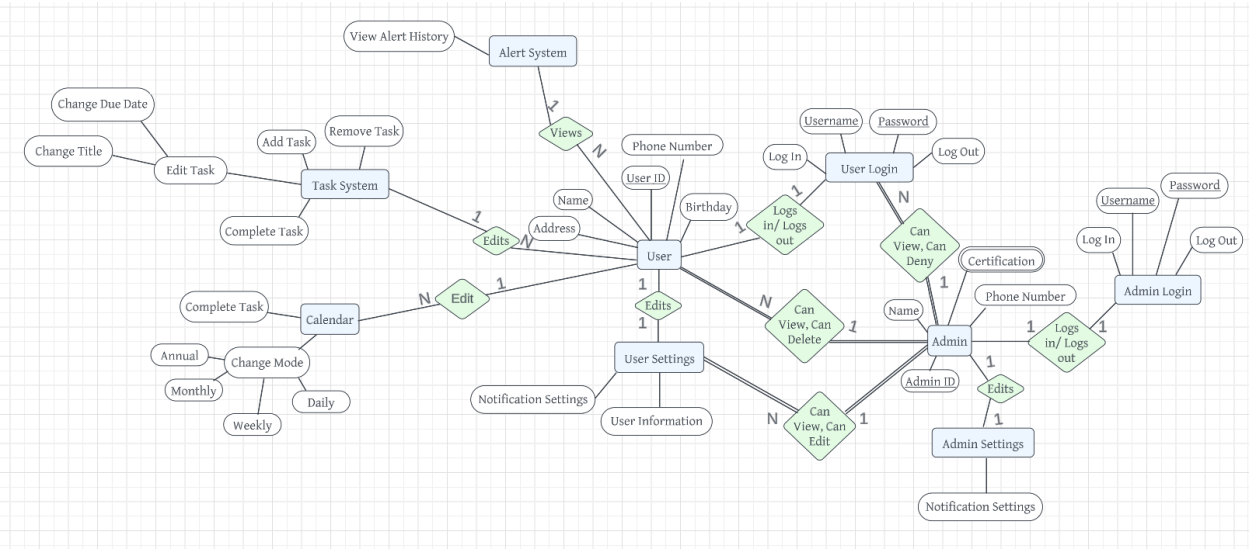
The "Admin" is also linked to the "User Information" use case via "Can View, Can Edit," explaining they have permission to access and modify user data. The "User Information" subsystem itself encompasses basic user details such as "User ID," "Name," "Address," "Phone Number," and "Birthday." This subsystem is associated with regular "User" entities, showing that admins can view and manipulate these user-specific details. Furthermore, the "Admin" has a unique "Can View, Can Delete" association with "User Settings," showcasing the ability to not only view but also delete user settings, a capability not typically given to standard users. The "Admin" also seems to have a specialized "Edits" relationship with "Admin Settings," which shows their capability to edit their settings. Additionally, “Certificate” is a multi-valued attribute.



Conceptual Model:

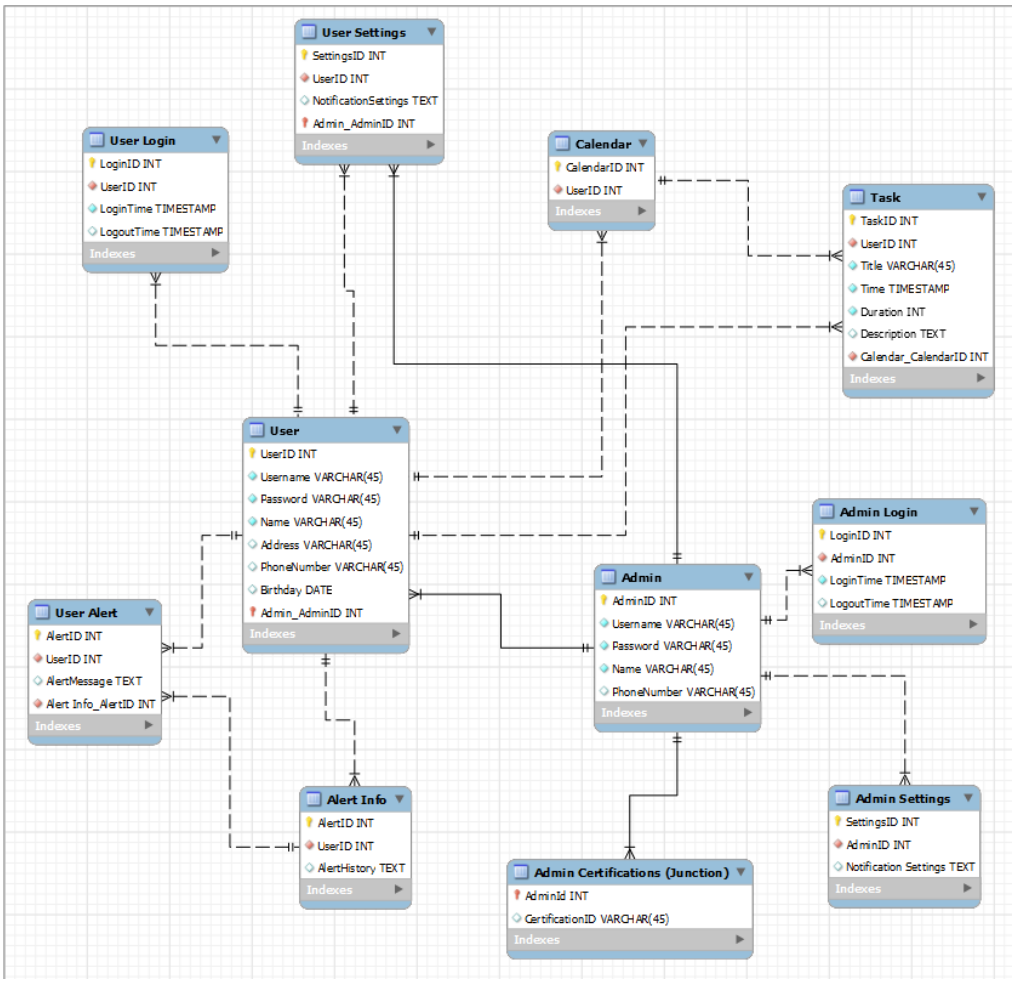
User Login/Admin Login: Both users and admins have separate login systems, but these systems are interconnected. Admins have the same basic login functionality as users but with additional privileges. User Settings/Admin Settings: Both regular users and admins can access and edit settings related to their accounts. However, admins have extended privileges, as indicated by the "Can View, Can Edit" and "Can View, Can Delete" associations. This shows that admins can not only modify their settings but also have the authority to view and alter or remove the settings of regular users. Notification Settings: This is a common feature between users and admins, allowing both to receive and configure notifications. Given that this appears under both "User Settings" and "Admin Settings," it shows a design where notification preferences can be managed at different levels of authority.

Task System/Calendar: Users can interact with tasks and the calendar, while admins seem to have oversight capabilities. This is inferred from the "Can View, Can Edit" and "Can View, Can Delete" connections from the admin to the user subsystems. Admins might not only manage tasks and calendar entries but also can delete or edit these entries. User Information: Admins have a clear connection to the "User Information" through the "Can View, Can Edit" relationship, indicating they can access and modify personal details of the users, such as "User ID," "Name," "Address," "Phone Number," and "Birthday." Alert System: While not directly connected to the "Admin" model, the "Alert System" is likely accessible to admins given their elevated access to user-related functionalities. They may have the ability to view and manage alerts and notifications that pertain to users. Edits: There are "Edits" relationships that connect the "User" to their settings and information, as well as the "Admin" to the admin settings. This explains that both users and admins can make changes to their respective information and settings within the system. Logs in/Logs out Both models include this feature, which shows that the system keeps track of both user and admin session activities.



EER Diagram:

My Enhanced Entity Relationship (EER) Diagram provides a detailed view of the database structure for my Task Management System involving users and administrators. Starting with the User table, it has a one-to-many relationship with multiple tables: User Settings: Each user has corresponding settings, indicating that a user can have multiple settings but each setting is associated with only one user. Calendar: Users can have multiple calendar entries, each uniquely identified within the Calendar table. User Login: This tracks the login and logout timestamps for users. A user can have multiple login records. User Alert: Indicates alerts specific to users, where a user can have multiple alerts. Task: Users are associated with tasks, where each task is related to one user, but a user can have multiple tasks. The Admin entity has a similar structure: Admin Login: Tracks login and logout activities of an admin. Admin Settings: Stores settings for each admin, with a one-to-many relationship. Admin Certifications (Function): It appears to represent a function rather than a table, potentially indicating certifications that an admin can issue or possess. The exact nature of the function is not clear from the diagram alone. Certifications are a multi-valued attribute and thus needed a junction table in the form of admin certifications, which is then connected to admin. Foreign keys that establish the relationships between tables, such as user IDs or alert information. The foreign keys create a relational structure that allows the database to maintain referential integrity.



Phase 3: Presentation and SQL Files:

Presentation:

Slide 2: Default Values

- A DEFAULT clause indicates a default value for a column.
- It can be a literal constant or an expression.
- Expression values are put in parentheses.
- For TIMESTAMP and DATETIME columns, CURRENT_TIMESTAMP can be used without parentheses.
- SERIAL DEFAULT VALUE is used for an integer column, acting as an alias for NOT NULL AUTO_INCREMENT UNIQUE.
- BLOB, TEXT, GEOMETRY, and JSON data types can only be assigned default values if they are written as expressions.

Slide 3: Rules for Expression Default Values

- Allowed constructs: Literals, built-in functions, and operators.
- Disallowed constructs: Subqueries, parameters, variables, stored functions, and loadable functions.
- An expression default cannot use a column that has an AUTO_INCREMENT attribute.
- An expression default value for one column can refer to other table columns, but references must be to columns that appear earlier in the table definition.
- This rule also applies to ALTER TABLE to reorder columns.

Slide 4: Altering Tables

- For CREATE TABLE ... LIKE and CREATE TABLE ... SELECT, the destination table uses default values from the original table.
- If an expression default value has a nondeterministic function, it cannot be used for statement-based replication.
- This includes INSERT and UPDATE.
- If binary logging is disabled, the statement will execute.
- If binary logging is enabled and binlog_format is set to STATEMENT, the statement will execute but a warning will be sent to the error log.
- If binlog_format is set to MIXED or ROW, the statement will execute.

Slide 5: Inserting Rows

- To insert a new row, the default value of a column with an expression value can be inserted by removing the column name or setting it to DEFAULT.

- The use of DEFAULT(col_name) to specify the default value for a named column can only be used for columns that have a literal default value.
- Not all storage engines allow expression values.
- ER_UNSUPPORTED_ACTION_ON_DEFAULT_VAL_GENERATED will occur.

Slide 6: Implicit Default Handling

- If there is no explicit DEFAULT value, MySQL will:
- Define the column with a DEFAULT NULL clause if the column can take NULL as a value.
- Define the column with no explicit DEFAULT clause if null cannot be used.
- For data entry into a NOT NULL column:
- In strict SQL mode, an error will occur.
- In non-strict mode, MySQL sets the column to the implicit default value for the column data type.

Slide 7: Implicit Defaults

- SHOW CREATE TABLE statement reveals columns with an explicit DEFAULT clause.
- Numeric types default to 0.
- Integer or floating-point types with AUTO_INCREMENT attribute default to the next value.
- Date and time types (except TIMESTAMP) default to the correct "zero" value.
- TIMESTAMP's default depends on the explicit_defaults_for_timestamp system variable.
- If enabled, TIMESTAMP defaults to the current date and time.
- Otherwise, it defaults to the current date and time if it's the first TIMESTAMP column.
- The default value for non-ENUM string types is the empty string.
- ENUM defaults to the first enumeration value.

Slide 8: Demo 1

```sql
CREATE DATABASE IF NOT EXISTS ProjectPhaseDefaultValuesDemo;
USE ProjectPhaseDefaultValuesDemo;

DROP TABLE IF EXISTS DefaultValueExample;

-- Create a table with default value specifications
CREATE TABLE DefaultValueExample (
    myInt INT DEFAULT 0,
    myVarchar VARCHAR(45) DEFAULT 'No Data',
    myFloat FLOAT DEFAULT 1.0,
    myDate DATE DEFAULT (CURRENT_DATE + INTERVAL 1 DAY),
    myTimestamp TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    myEnum ENUM('Value1', 'Value2', 'Value3') DEFAULT 'Value1',
    mySet SET('A', 'B', 'C') DEFAULT 'A,B'
);
```

Slide 9: Demo 2

```sql
USE ProjectPhaseDefaultValuesDemo;

-- Inserts a row with all of the default values
INSERT INTO DefaultValueExample () VALUES();

-- Inserts a new row with only 3 values specified, the rest will be default
INSERT INTO DefaultValueExample (myInt, myVarchar, myDate) VALUES (99, 'Data', '2023-05-05');

SELECT * FROM DefaultValueExample;

SELECT CONCAT(
    myInt, ', ',
    myVarchar, ', ',
    myFloat, ', ',
    myDate, ', ',
    myTimestamp, ', ',
    myEnum, ', (',
    mySet, ')'
) AS allColumnConcatenation
FROM DefaultValueExample;
```

Slide 10: Choosing the Right Type for a Column

- Optimal storage: Prefer the most precise data type in most cases.
- Example: Use MEDIUMINT UNSIGNED for values 1-99999 in an integer column.
- DECIMAL column calculations: Have a precision of 65 decimal digits for basic operations (+, -, *, and /).
- Speed over accuracy: Consider using the DOUBLE type when precision isn't critical.
- High precision option: Convert to a fixed-point type stored in a BIGINT for precision.
- Perform calculations with 64-bit integers and convert results back to floating-point values as needed.

Slide 11: Demo 3

```sql
create database if not exists TheRightType;
use TheRightType;

DROP TABLE IF EXISTS TheRightType;

CREATE TABLE Example (
myInt int,
myTinyInt tinyint,
mySmallInt smallint,
myMediumInt mediumint,
myDecimal decimal,
myFloat float
);
insert into Example (myInt, myTinyInt, mySmallInt, myMediumInt, myDecimal, myFloat)
values (3359000, 0, 50, 43000, 33.2, 5.983),
(190, -2, 120, 45, -3.9, 10.254),
(495, 1, 7, 568, 276.4, -96.892);
```

Slide 12: Demo 4

```sql
-- Use the previously created database and table
USE TheRightType;

-- Query to demonstrate that MEDIUMINT is sufficient for storing values between 1 and 99999
SELECT * FROM Example WHERE myMediumInt BETWEEN 1 AND 99999;

-- Query to show calculations with FLOAT type columns
SELECT myFloat / 2 AS 'HalfFloat' FROM Example;

-- Query to demonstrate high precision with DECIMAL
SELECT myDecimal + 0.0000000001 AS 'HighPrecisionDecimal' FROM Example;

-- Query to demonstrate speed with FLOAT but loss of precision
SELECT myFloat + 0.0000000001 AS 'LowPrecisionFloat' FROM Example;

-- Query to demonstrate range of SMALLINT
SELECT * FROM Example WHERE mySmallInt BETWEEN -32768 AND 32767;

-- Query to demonstrate range of TINYINT
SELECT * FROM Example WHERE myTinyInt BETWEEN -128 AND 127;
```

Phase 4: Database Development

The ‘CREATE TABLE IF NOT EXISTS’ statement checks if the ‘User’ table already exists within the database. If it does not, the following table will be created. This code defines the structure for the ‘User’ table with various attributes related to user information. ‘UserID’ is an integer column that represents a unique identifier for each user. This column is set as the primary key of this table, which means it must be unique and there cannot be any NULL values. ‘Username’ is a variable character column that has a maximum length of 45 characters. It stores the display name of the user. ‘Password’ is a variable character column that has a maximum length of 45 characters. It stores the user’s password. ‘Name’ is a variable character column that has a maximum length of 45 characters. It stores the full name of the user. ‘Address’ is a variable character column that has a maximum length of 45 characters. It stores the user’s address. ‘PhoneNumber’ is a variable character column that has a maximum length of 45 characters. It stores the user’s phone number. ‘Birthday’ is a date-type column. It stores the user’s birthday. The columns ‘Username’, ‘Password’, and ‘Name’ have the ‘NOT NULL’ constraint, meaning these columns must always have a value in them.

```
-- Create the User table
CREATE TABLE IF NOT EXISTS User (
    UserID INT PRIMARY KEY,
    Username VARCHAR(45) NOT NULL,
    Password VARCHAR(45) NOT NULL,
    Name VARCHAR(45) NOT NULL,
    Address VARCHAR(45),
    PhoneNumber VARCHAR(45),
    Birthday DATE
);
```

Figure 5: User Table Code

The ‘CREATE TABLE IF NOT EXISTS’ statement checks if the ‘UserSettings’ table already exists within the database. If it does not, the following table will be created. This code defines the structure for the ‘UserSettings’ table that holds settings related to individual users. This column is set as the primary key of this table, which means it must be unique and there cannot be any NULL values. ‘SettingsID’ is an integer column that represents a unique identifier for each setting. ‘UserID’ is an integer column that is used to link settings to a user. This column is a foreign key, which means that each record in ‘UserSettings’ has a valid corresponding ‘UserID’ in the ‘User’ table. This establishes a many-to-one relationship between the ‘UserSettings’ table and the ‘User’ table. ‘Mode’ is a variable character column that has a maximum length of 45 characters. It stores the mode of the user settings (e.g., dark mode or light mode). ‘NotificationSettings’ is a text type column, which stores notification settings set by the user.

```
-- Create the User Settings table
CREATE TABLE IF NOT EXISTS UserSettings (
    SettingsID INT PRIMARY KEY,
    UserID INT,
    Mode VARCHAR(45),
    NotificationSettings TEXT,
    FOREIGN KEY (UserID) REFERENCES User(UserID)
);
```

Figure 6: User Settings Table Code

The ‘CREATE TABLE IF NOT EXISTS’ statement checks if the ‘Calendar’ table already exists within the database. If it does not, the following table will be created. This code defines the structure for the ‘Calendar’ table that holds calendar-related data for individual users. ‘CalendarID’ is an integer column that represents a unique identifier for each calendar entry. This column is set as the primary key of this table, which means it must be unique and there cannot be any NULL values. ‘UserID ’ is an integer column that is used to link a calendar to a user. This column is a foreign key, which means that each record in ‘Calendar’ has a valid corresponding ‘UserID’ in the ‘User’ table. This establishes a many-to-one relationship between the ‘Calendar’ table and the ‘User’ table.‘Mode’ is an integer column that represents the numerical code that indicates a specific mode of the calendar.

```
-- Create the Calendar table
CREATE TABLE IF NOT EXISTS Calendar (
    CalendarID INT PRIMARY KEY,
    UserID INT,
    Mode INT,
    FOREIGN KEY (UserID) REFERENCES User(UserID)
);
```

Figure 7: Calendar Table Code

The ‘CREATE TABLE IF NOT EXISTS Task’ table is created with several columns to store task-related information such as the title, description, and duration as well as having the calendar linked to the table. With the CalendarID entity linked to this table, the user is allowed to give their tasks a timetable with a due date and time. The UserID is also linked to this table as it allows each user to have their own, unique task table. The Title entity allows you to name a task that is up to 45 characters in length. This value can never be blank, therefore, each task created must have a title. The rest of the entities have data types that can be filled in by the user when they create the task.

```
-- Create the Task table
CREATE TABLE IF NOT EXISTS Task (
    TaskID INT PRIMARY KEY,
    UserID INT,
    Title VARCHAR(45) NOT NULL,
    Time TIMESTAMP,
    Duration INT,
    Description TEXT,
    Calendar_CalendarID INT,
    FOREIGN KEY (UserID) REFERENCES User(UserID),
    FOREIGN KEY (Calendar_CalendarID) REFERENCES Calendar(CalendarID)
);
```

Figure 8: Task Table Code

The ‘CREATE TABLE IF NOT EXISTS UserAlert’ allows the system to create user alerts for their FoxTask to-do’s. The AlertID is the primary key as it is the distinct value for this table and it is needed for the rest of the values. The UserID is an integer and is different for every user. The AlertMessage is a TEXT data type because when the user gets the alert, it has to be in text form so they can properly read it. Each alert will be different depending on the task which is why each alert has an AlertID, this is an integer and will change each alert the user gets.

```
-- Create the User Alert table
CREATE TABLE IF NOT EXISTS UserAlert (
    AlertID INT PRIMARY KEY,
    UserID INT,
    AlertMessage TEXT,
    Alert_Info_AlertID INT,
    FOREIGN KEY (UserID) REFERENCES User(UserID)
);
```

Figure 9: User Alert Table Code

The ‘CREATE TABLE IF NOT EXISTS UserLogin’ is a table with columns for a unique login identifier(LoginID), a user identifier(UserID) to link a user in another table with timestamps that allow you to record login and logout times. LoginID is the primary key as it uniquely identifies each row in the table. The UserID is an integer that will represent the user's identification throughout the whole process, this is linked to the other UserIDs in these tables. The LoginTime and LogoutTimes represent time stamps with dates and times when the user logs in or out of the app.

```
-- Create the User Login table
CREATE TABLE IF NOT EXISTS UserLogin (
    LoginID INT PRIMARY KEY,
    UserID INT,
    LoginTime TIMESTAMP,
    LogoutTime TIMESTAMP,
    FOREIGN KEY (UserID) REFERENCES User(UserID)
);
```

Figure 10: User Login Table Code

The ‘CREATE TABLE IF NOT EXISTS AlertInfo’ table has three columns, AlertID, UserID, and AlertHistory. These values just establish relationships with the other entities in the remaining diagrams. The foreign key ensures that the values in the “UserID” column of the “AlertInfo” table must exist in the “UserID” column of the User table which creates a link between the two tables. Once again, each user has their own unique ID and alert ID to allow the data to be separated from others.

```
-- Create the Alert Info table
CREATE TABLE IF NOT EXISTS AlertInfo (
    AlertID INT PRIMARY KEY,
    UserID INT,
    AlertHistory TEXT,
    FOREIGN KEY (UserID) REFERENCES User(UserID)
);
```

Figure 11: Alert Info Table Code

The ‘CREATE TABLE IF NOT EXISTS Admin’ allows the admins to log in and create a username, password, name, certification, and phone number for their profile. We say “IF NOT EXISTS” in this situation as it reduces redundancy and errors if the table was already created somewhere else. Each of these entities must be in the range of 0-45 characters or else it is not valid. It is optional for the admin to add their certification and phone number but the rest of the values must be filled as it is declared as NOT NULL. The foreign key establishes a relationship between UserID and AlertInfo. This ensures that the values of the UserID column of the AlertInfo table must exist in the UserID column of the user table, which also creates a link between these two tables.

```
-- Create the Admin table
CREATE TABLE IF NOT EXISTS Admin (
    AdminID INT PRIMARY KEY,
    Username VARCHAR(45) NOT NULL,
    Password VARCHAR(45) NOT NULL,
    Name VARCHAR(45) NOT NULL,
    Certification VARCHAR(45),
    PhoneNumber VARCHAR(45)
);
```

Figure 12: Admin Table Code

The ‘CREATE TABLE IF NOT EXISTS AdminLogin’ allows the admin to log in through their adminID and get a login ID with an integer and primary key. It also logs the specific time that the person logged into their account and then when they log out it keeps a stamp of when they logged out. Using the AdminID Key.

```
-- Create the Admin Login table
CREATE TABLE IF NOT EXISTS AdminLogin (
    LoginID INT PRIMARY KEY,
    AdminID INT,
    LoginTime TIMESTAMP,
    LogoutTime TIMESTAMP,
    FOREIGN KEY (AdminID) REFERENCES Admin(AdminID)
);
```

Figure 13: Admin Login Table Code

The "CREATE TABLE IF NOT EXISTS AdminSettings" table is designed to store various settings for administrators. The AdminID column makes a relationship with the "Admin" table, making sure that each setting is associated with an administrator. The table's primary key is the SettingsID column, which provides a unique identifier for each row in the table.

```
-- Create the Admin Settings table
CREATE TABLE IF NOT EXISTS AdminSettings (
    SettingsID INT PRIMARY KEY,
    AdminID INT,
    Mode VARCHAR(45),
    NotificationSettings TEXT,
    FOREIGN KEY (AdminID) REFERENCES Admin(AdminID)
);
```

Figure 14: Admin Settings Table Code

Phase 5.1: Loading data and performance enhancements:

Handling Foreign Key Constraints:

Part B: In my SQL code, I have included an example of an insertion error due to foreign key constraints. (Error below)

> Cannot add or update a child row: a foreign key constraint fails (`MaristTask`.`user settings, CONSTRAINT `usersettings_ibfk_1` FOREIGN KEY (`UserID`) REFERENCES `user` (`UserID`)) 0.000 sec

This error occurs when trying to insert data into a child table (UserSettings) with a foreign key (UserID) that does not exist in the parent table (User). (The line that caused the error is displayed below)

```
-- Line that would cause the  insertion error due to foreign key constraints
INSERT INTO UserSettings (SettingsID, UserID, Mode, NotificationSettings) VALUES
(11, 999, 'Dark', 'Email');--Primary Key Constraint Test
```

Part C: The method that we used to handle the issue of foreign key inconsistency was to set foreign key checks to 0 before the insertions and then set it back to 1 after the insertions. We used this method because it is useful for bulk data insertion where the order of data insertion might temporarily violate foreign key constraints, and our data insertion is done in bulk. This can be seen at the start of our data insertion code here:

```
1   use RedFoxTask;
2
3   SET FOREIGN_KEY_CHECKS=0;
```

And the end of our data insertion code here:

```
137     (8, 8, 'Light', 'Email, SMS'),
138     (9, 9, 'Dark', 'Email'),
139     (10, 10, 'Light', 'SMS');
140
141 •   SET FOREIGN_KEY_CHECKS=1;
```

Importing Data:

Our importing process involved inserting data into tables such as User, UserSettings, Calendar, Task, UserAlert, and several others. Each table received a variety of data, showcasing our database's capability to handle different data types, including VARCHAR, INT, DATE, TIMESTAMP, and TEXT.

Figure 1: Importing the data for the User and UserSettings tables

```
-- Optimized bulk insertions take a total time of 0.000 seconds (I assume this means a number
-- smaller than 0.000 but too small to be displayed with 3 signficant figures
Insert INTO User (UserID, Username, Password, Name, Address, PhoneNumber, Birthday) VALUES
(1, 'KevinR', 'KevinR1', 'Kevin Reiff', '123 Main St', '123-456-7890', '2002-10-23'),
(2, 'janedoe', 'pass456', 'Jane Doe', '124 Main St', NULL, '1992-02-02'),
(3, 'mikeb', 'mikepass', 'Mike Brown', '125 Main St', '123-456-7891', '1988-03-03'),
(4, 'emmagreen', 'emma1234', 'Emma Green', NULL, '123-456-7892', '1995-04-04'),
(5, 'samw', 'samspass', 'Sam Wilson', '126 Main St', '123-456-7893', '1991-05-05'),
(6, 'lucyp', 'lucyword', 'Lucy Parker', '127 Main St', '123-456-7894', '1989-06-06'),
(7, 'davidt', 'davidpwd', 'David Taylor', NULL, '123-456-7895', '1993-07-07'),
(8, 'sarahc', 'sarahpass', 'Sarah Connor', '128 Main St', '123-456-7896', '1994-08-08'),
(9, 'rickg', 'rick123', 'Rick Grimes', '129 Main St', NULL, '1996-09-09'),
(10, 'annak', 'anna456', 'Anna Klein', '130 Main St', '123-456-7897', '1990-10-10');
```

```
INSERT INTO UserSettings (SettingsID, UserID, NotificationSettings) VALUES
(1, 1, 'SMS'),
(2, 2, NULL),
(3, 3, 'SMS'),
(4, 4, NULL),
(5, 5, 'SMS'),
(6, 6, NULL),
(7, 7, NULL),
(8, 8, 'SMS'),
(9, 9, 'SMS'),
(10, 10, 'SMS');
```

The User table stores information about individual users, including their username, password, personal details, and contact information. As for the imported data, 10 distinct user records were inserted. The data demonstrates a range of different usernames, addresses, and birthdays.
Notably, some users do not have addresses or phone numbers, showcasing optional fields. The UserSettings table holds settings preferences for each user, like display mode and notification settings. The data shows variations and notification settings, including null values.

Figure 2: Importing the data for the Calendar, Task, and UserAlert tables

```
45 •  INSERT INTO Calendar (CalendarID, UserID, Mode) VALUES
46    (1, 1, 1),
47    (2, 2, 0),
48    (3, 3, 1),
49    (4, 4, 0),
50    (5, 5, 1),
51    (6, 6, 0),
52    (7, 7, 1),
53    (8, 8, 0),
54    (9, 9, 1),
55    (10, 10, 0);
56
57 •  INSERT INTO Task (TaskID, UserID, Title, Time, Duration, Description, Calendar_CalendarID) VALUES
58    (1, 1, 'Task 1', '2023-11-10 09:00:00', 60, 'Description for Task 1', 1),
59    (2, 1, 'Task 2', '2023-11-10 10:00:00', 30, 'Description for Task 2', 1),
60    (3, 2, 'Task 3', '2023-11-11 09:00:00', 200, 'Description for Task 3', 2),
61    (4, 2, 'Task 4', '2023-11-11 10:15:00', 1000, 'Description for Task 4', 2),
62    (5, 3, 'Task 5', '2023-11-12 09:00:00', 300, 'Description for Task 5', 3),
63    (6, 3, 'Task 6', '2023-11-12 10:00:00', 45, 'Description for Task 6', 3),
64    (7, 4, 'Task 7', '2023-11-13 09:00:00', 120, 'Description for Task 7', 4),
65    (8, 4, 'Task 8', '2023-11-13 10:00:00', 180, 'Description for Task 8', 4),
66    (9, 5, 'Task 9', '2023-11-14 09:00:00', 90, 'Description for Task 9', 5),
67    (10, 5, 'Task 10', '2023-11-14 10:15:00', 365, 'Description for Task 10', 10);
68
69 •  INSERT INTO UserAlert (AlertID, UserID, AlertMessage, Alert_Info_AlertID) VALUES
70    (1, 1, 'Alert Message 1', 1),
71    (2, 2, 'Alert Message 2', 2),
72    (3, 3, 'Alert Message 3', 3),
73    (4, 4, 'Alert Message 4', 4),
74    (5, 5, 'Alert Message 5', 5),
75    (6, 6, 'Alert Message 6', 6),
76    (7, 7, 'Alert Message 7', 7),
77    (8, 8, 'Alert Message 8', 8),
78    (9, 9, 'Alert Message 9', 9),
79    (10, 10, 'Alert Message 10', 10);
```

The Calendar table links users to their calendar entries, represented by a mode. The mode field demonstrates binary states, 0 and 1, for different calendar types. The Task table contains tasks associated with users, including title, duration, and description. The imported data showcases varying lengths and types of tasks, with different durations and descriptions. The UserAlert table contains alert messages for the users. The imported data shows that each record is a unique combination of an alert ID, a corresponding user ID, an alert message, and a

reference to an alert information ID. These records demonstrate the ability of the table to store personalized alerts for each user.

Figure 3: Importing the data for the User login, AlertInfo, and Admin tables

```
81 •  INSERT INTO UserLogin (LoginID, UserID, LoginTime, LogoutTime) VALUES
82    (1, 1, '2023-11-10 08:00:00', '2023-11-10 17:00:00'),
83    (2, 2, '2023-11-10 09:00:00', '2023-11-10 18:00:00'),
84    (3, 3, '2023-11-11 08:30:00', '2023-11-11 17:30:00'),
85    (4, 4, '2023-11-11 09:30:00', '2023-11-11 18:30:00'),
86    (5, 5, '2023-11-12 08:00:00', '2023-11-12 17:00:00'),
87    (6, 6, '2023-11-12 09:00:00', '2023-11-12 18:00:00'),
88    (7, 7, '2023-11-13 08:30:00', '2023-11-13 17:30:00'),
89    (8, 8, '2023-11-13 09:30:00', '2023-11-13 18:30:00'),
90    (9, 9, '2023-11-14 08:00:00', '2023-11-14 17:00:00'),
91    (10, 10, '2023-11-14 09:00:00', '2023-11-14 18:00:00');
92
93 •  INSERT INTO AlertInfo (AlertID, UserID, AlertHistory) VALUES
94    (1, 1, 'History 1'),
95    (2, 2, 'History 2'),
96    (3, 3, 'History 3'),
97    (4, 4, 'History 4'),
98    (5, 5, 'History 5'),
99    (6, 6, 'History 6'),
100   (7, 7, 'History 7'),
101   (8, 8, 'History 8'),
102   (9, 9, 'History 9'),
103   (10, 10, 'History 10');
104
105 •  INSERT INTO Admin (AdminID, Username, Password, Name, Certification, PhoneNumber) VALUES
106   (1, 'admin1', 'adminpass1', 'Admin One', 'Certified', '111-222-3333'),
107   (2, 'admin2', 'adminpass2', 'Admin Two', 'Certified', '111-222-3334'),
108   (3, 'admin3', 'adminpass3', 'Admin Three', 'Certified', '111-222-3335'),
109   (4, 'admin4', 'adminpass4', 'Admin Four', NULL, '111-222-3336'),
110   (5, 'admin5', 'adminpass5', 'Admin Five', 'Certified', '111-222-3337'),
111   (6, 'admin6', 'adminpass6', 'Admin Six', NULL, '111-222-3338'),
112   (7, 'admin7', 'adminpass7', 'Admin Seven', 'Certified', '111-222-3339'),
113   (8, 'admin8', 'adminpass8', 'Admin Eight', 'Certified', '111-222-3340'),
114   (9, 'admin9', 'adminpass9', 'Admin Nine', NULL, '111-222-3341'),
115   (10, 'admin10', 'adminpass10', 'Admin Ten', 'Certified', '111-222-3342');
```

The UserLogin table tracks each user's login and logout activities. It includes LoginID, UserID, LoginTime, and LogoutTime. This query adds 10 entries, each representing a distinct user session. It captures the time each user logs in and out, providing a record of user activity within the system. The AlertInfo table is intended to store detailed historical information about alerts for each user. This section demonstrates the ability of the AlertInfo table to maintain a

history log for each alert. By inserting 10 distinct records, the table shows a relationship between users and their respective alert histories. The Admin table is designed to manage administrator accounts. This query populates the Admin table with 10 entries. Each entry represents an administrator, containing a unique combination of ID, username, password, name, certification status, and contact number. This showcases the table's capability to handle multiple admin profiles with distinct credentials and personal details.

Figure 4: Importing the data for the AdminLogin and AdminSettings tables:

```
INSERT INTO AdminLogin (LoginID, AdminID, LoginTime, LogoutTime) VALUES
(1, 1, '2023-11-10 08:00:00', '2023-11-10 17:00:00'),
(2, 2, '2023-11-10 09:00:00', '2023-11-10 18:00:00'),
(3, 3, '2023-11-11 08:30:00', '2023-11-11 17:30:00'),
(4, 4, '2023-11-11 09:30:00', '2023-11-11 18:30:00'),
(5, 5, '2023-11-12 08:00:00', '2023-11-12 17:00:00'),
(6, 6, '2023-11-12 09:00:00', '2023-11-12 18:00:00'),
(7, 7, '2023-11-13 08:30:00', '2023-11-13 17:30:00'),
(8, 8, '2023-11-13 09:30:00', '2023-11-13 18:30:00'),
(9, 9, '2023-11-14 08:00:00', '2023-11-14 17:00:00'),
(10, 10, '2023-11-14 09:00:00', '2023-11-14 18:00:00');

INSERT INTO AdminSettings (SettingsID, AdminID, NotificationSettings) VALUES
(1, 1, 'Email'),
(2, 2, 'SMS'),
(3, 3, 'Email, SMS'),
(4, 4, NULL),
(5, 5, 'Email'),
(6, 6, 'SMS'),
(7, 7, NULL),
(8, 8, 'Email, SMS'),
(9, 9, 'Email'),
(10, 10, 'SMS');
```

The AdminLogin table is designed to track the login and logout activities of administrators in the system. The insertion command above adds 10 distinct entries to the AdminLogin table. Each entry details a specific admin session, highlighting the precise times of logging in and out. This data is critical for understanding admin usage patterns and enhancing system security. The AdminSettings table manages the individual settings preferences for each administrator. This insertion operation populates the AdminSettings table with 10 records, each corresponding to an administrator. These records demonstrate the flexibility of the system in accommodating notification settings SMS or No SMS. This capability is essential for ensuring

that administrators can customize their working environment according to their preferences, thus improving efficiency and user experience.

Figure 5: Testing Data Constraints

```
14    -- Primary Key Constraint Test: Try to insert a record into the User table with an existing UserID.
15    INSERT INTO User (UserID, Username, Password, Name, Address, PhoneNumber, Birthday) VALUES
16    (1, 'newuser', 'newpass', 'New User', '131 Main St', '123-456-7898', '1991-11-11');
17
18    -- Foreign Key Constraint Test: Insert a record in UserSettings with a UserID that doesn't exist in the User table.
19    INSERT INTO UserSettings (SettingsID, UserID, Mode, NotificationSettings) VALUES
20    (11, 999, 'Dark', 'Email, SMS');
21
22    -- Data Type Constraint Test: Attempt to insert a non-date value into a date column in the User table
23    INSERT INTO User (UserID, Username, Password, Name, Address, PhoneNumber, Birthday) VALUES
24    (11, 'testuser', 'testpass', 'Test User', '132 Main St', '123-456-7899', 'not a date');
25
26    -- Unique Constraint Test: Assuming you have a unique constraint on a column (like Username in User table),
27    -- try inserting a duplicate username.
28    INSERT INTO User (UserID, Username, Password, Name, Address, PhoneNumber, Birthday) VALUES
29    (11, 'johndoe', 'uniquepass', 'Unique User', '133 Main St', '123-456-7800', '1992-12-12');
```

- The Primary Key Constraint Test: Attempts to insert a record into the User table with an existing UserID. As UserID is a primary key, it must be unique for each record. The test confirms that the database prevents duplication in primary key fields, thereby maintaining the uniqueness and integrity of each record. This test will provide this error:

Error Code: 1062. Duplicate entry '1' for key 'user.PRIMARY'

- Foreign Key Constraint Test: Here, the insertion into UserSettings includes a UserID that does not exist in the User table. This test verifies that the foreign key constraint is active and prevents the insertion of records with non-existent foreign key values, ensuring referential integrity between tables. This test provides this error:

Error Code: 1452. Cannot add or update a child row: a foreign key constraint fails (`redfoxtask`.`usersettings`, CONSTRAINT `usersettings_ibfk_1` FOREIGN KEY (`UserID`) REFERENCES `user` (`UserID`))

  - (It's a little hard to see the screenshot so here's a printout: Error Code: 1452. Cannot add or update a child row: a foreign key constraint fails (`MaristTask`.`user settings', CONSTRAINT `usersettings_ibfk_1` FOREIGN KEY (`UserID`) REFERENCES `user` (`UserID`)) 0.000 sec)

- Data Type Constraint Test: This command attempts to insert a non-date value into the date column (Birthday) in the User table. The test checks the enforcement of data type constraints, ensuring that each field in the database only accepts data of the correct type. This test provides this error:

Error Code: 1292. Incorrect date value: 'not a date' for column 'Birthday' at row 1

- Unique Constraint Test: This test tries to insert a duplicate Username into the User table, assuming a unique constraint on the Username column. It validates that the unique constraint is working correctly by preventing the duplication of values in specific fields, ensuring data uniqueness. This test provides this error:

```
Error Code: 1062. Duplicate entry 'johndoe' for key 'user.Username'
```

Figure 6: Table Verification Queries:

```
-- Lines that verify Tables Structures

SELECT * FROM User;
SELECT * FROM User WHERE Birthday > '1990-01-01' ORDER BY Birthday;
SELECT * FROM UserSettings;
SELECT User.Username, UserSettings.Mode FROM User INNER JOIN UserSettings ON User.UserID = UserSettings.UserID;
SELECT * FROM Calendar WHERE Mode = 1;
SELECT User.Name, Calendar.CalendarID FROM User RIGHT JOIN Calendar ON User.UserID = Calendar.UserID;
SELECT * FROM Task;
SELECT User.Name, Task.Title FROM User LEFT JOIN Task ON User.UserID = Task.UserID
UNION
SELECT User.Name, Task.Title FROM User RIGHT JOIN Task ON User.UserID = Task.UserID;
SELECT * FROM UserAlert ORDER BY AlertID DESC;
SELECT UserAlert.AlertMessage, User.Name FROM UserAlert INNER JOIN User ON UserAlert.UserID = User.UserID;
```

These queries all demonstrate that the tables have proper structure given that each of these queries can be performed successfully. This shows that data can be taken from each table and displayed/manipulated in a variety of ways, thus demonstrating the correct structure of these tables.

Insertion Optimization:

In the development of the RedFoxTask database, we applied optimization techniques to improve the efficiency of data insertion. This optimization is particularly important in a real-world scenario where databases often handle large volumes of data, necessitating efficient data management practices. We experimented with two different methods of data insertion: single-line insertions and bulk insertions.

Figure 7: Single Line Insertions: (This figure ended up being hard to see, so here's a link to it)

https://drive.google.com/file/d/1p4T9jIAkhUD6MU4LNT1Ut2-vhL_Yd44i/view?usp=drive_link

Initially, we inserted data into our tables one row at a time. This method, while straightforward, was inefficient, especially when dealing with large datasets. In our case, the single-line insertions took a total of approximately 0.016 seconds per insertion, as demonstrated in the sample code provided.

Figure 8: Bulk Insertions:

```
18      -- Optimized bulk insertions take a total time of 0.000 seconds (I assume this means a number
19      -- smaller than 0.000 but too small to be displayed with 3 signficant figures
20 •    Insert INTO User (UserID, Username, Password, Name, Address, PhoneNumber, Birthday) VALUES
21      (1, 'johndoe', 'password123', 'John Doe', '123 Main St', '123-456-7890', '1990-01-01'),
22      (2, 'janedoe', 'pass456', 'Jane Doe', '124 Main St', NULL, '1992-02-02'),
23      (3, 'mikeb', 'mikepass', 'Mike Brown', '125 Main St', '123-456-7891', '1988-03-03'),
24      (4, 'emmagreen', 'emma1234', 'Emma Green', NULL, '123-456-7892', '1995-04-04'),
25      (5, 'samw', 'samspass', 'Sam Wilson', '126 Main St', '123-456-7893', '1991-05-05'),
26      (6, 'lucyp', 'lucyword', 'Lucy Parker', '127 Main St', '123-456-7894', '1989-06-06'),
27      (7, 'davidt', 'davidpwd', 'David Taylor', NULL, '123-456-7895', '1993-07-07'),
28      (8, 'sarahc', 'sarahpass', 'Sarah Connor', '128 Main St', '123-456-7896', '1994-08-08'),
29      (9, 'rickg', 'rick123', 'Rick Grimes', '129 Main St', NULL, '1996-09-09'),
30      (10, 'annak', 'anna456', 'Anna Klein', '130 Main St', '123-456-7897', '1990-10-10');
```

To enhance performance, we then implemented bulk insertions. This method allows multiple rows to be inserted in a single command, significantly reducing the time spent on database communication overhead. The optimized bulk insertions demonstrated a remarkable improvement in execution time, taking a total of approximately 0.000 seconds. We're assuming that this means a number that is smaller than 0.000 but cannot be displayed within three significant figures.

Normalization Check:

Figure 9: Calendar Table:

```
CREATE TABLE IF NOT EXISTS Calendar (
    CalendarID INT PRIMARY KEY,
    UserID INT,
    Mode INT,
    FOREIGN KEY (UserID) REFERENCES User(UserID)
);
```

In the first image (before normalization), the Calendar table is defined with three columns: CalendarID, UserID, and Mode. The CalendarID column is designated as the primary key, meaning it uniquely identifies each row in the table. Additionally, there's a foreign key

constraint on the UserID column, which references the UserID column in a User table, enforcing referential integrity between the Calendar and User tables. The second image (after normalization) shows a revised Calendar table structure. Here, the primary key has changed from a single column (CalendarID) to a composite key that includes both CalendarID and UserID. The foreign key constraint remains the same.

The change from a single primary key to a composite primary key fixes the fact that the original table structure violated the second normal form (2NF). The 2NF requires that the table be in first normal form (1NF) and that all non-primary-key columns be fully functionally dependent on the primary key. In other words, no non-primary key column should depend on only a part of a composite primary key. In the pre-normalization table, if CalendarID was not unique across different users, the Mode column would only be partially dependent on the primary key since Mode could vary for the same CalendarID but a different UserID. This partial dependency is what violates 2NF. By changing the primary key to a composite key of both CalendarID and UserID, the normalization ensures that each combination of CalendarID and UserID uniquely identifies a row, and thus, the Mode is fully functionally dependent on the entire primary key. This change enforces 2NF by eliminating the partial dependency of non-primary key columns on the primary key.

```
CREATE TABLE IF NOT EXISTS Calendar (
    CalendarID INT,
    UserID INT,
    Mode INT,
    PRIMARY KEY (CalendarID, UserID),
    FOREIGN KEY (UserID) REFERENCES User(UserID)
);
```

Figure 10: Task Table

```
CREATE TABLE IF NOT EXISTS Task (
    TaskID INT PRIMARY KEY,
    UserID INT,
    Title VARCHAR(45) NOT NULL,
    Time TIMESTAMP,
    Duration INT,
    Description TEXT,
    Calendar_CalendarID INT,
    FOREIGN KEY (UserID) REFERENCES User(UserID),
    FOREIGN KEY (Calendar_CalendarID) REFERENCES Calendar(CalendarID)
);
```

In the first image above, which represents the pre-normalized state, the Task table has the following columns: TaskID, UserID, Title, Time, Duration, Description, and Calendar_CalendarID. The primary key is solely TaskID, and two foreign keys reference User(UserID) and Calendar(CalendarID) respectively. The first image violates the Second Normal Form (2NF). For a table to be in 2NF, it must be in First Normal Form (1NF), and all non-key attributes must be fully functionally dependent on the primary key. The table in the first image has composite data that could be associated with TaskID and UserID together, but not on TaskID alone (assuming UserID is part of the primary key as it seems to be a defining attribute of a task as well). This could potentially cause update anomalies if, for example, a user's details are duplicated across multiple tasks.

In the second image (Below), which represents the post-normalized state, the primary key is changed to include both TaskID and UserID, along with Calendar_CalendarID. This composite primary key ensures that each record is uniquely identified by the combination of these three fields, rather than just TaskID. This change addresses the 2NF violation by ensuring that all non-key attributes (Title, Time, Duration, Description) are now fully dependent on the entire primary key, not just part of it. By including both UserID and Calendar_CalendarID in the primary key, the second image's table schema also implicitly moves towards satisfying the Third Normal Form (3NF), which requires that all non-key attributes be not only fully functionally dependent on the primary key but also non-transitively dependent on the primary key.

```
CREATE TABLE IF NOT EXISTS Task (
    TaskID INT,
    UserID INT,
    Title VARCHAR(45) NOT NULL,
    Time TIMESTAMP,
    Duration INT,
    Description TEXT,
    Calendar_CalendarID INT,
    PRIMARY KEY (TaskID, UserID, Calendar_CalendarID),
    FOREIGN KEY (UserID) REFERENCES User(UserID),
    FOREIGN KEY (Calendar_CalendarID) REFERENCES Calendar(CalendarID)
);
```

Phase 5.2: Loading Data and Performance Enhancements

Flowchart 1:

The log-in chart begins with "Start" leading to "Login". A decision point then asks if "User information correct?". If "No", the flow returns to "Login" with the note "Invalid login, try again." If "Yes", it proceeds to "Display tasks and calendar page", and then to "Finish". This flowchart describes the path that the user will take upon reaching the landing ‘log in’ page and the actions that will be taken to log in.

Flowchart 2:

The create a new task chart begins with "Start" leading to "Display Calendar and task list". A decision point then asks if the user wants to "Create a new task?".If "No", another decision point asks "Edit task?". If "Yes" to editing, it moves to "Change due date or description", and then back to the "Task still in progress" point. If "Yes" to create a task, it proceeds to "Enter the description and due date". Then, another decision point asks if the "Task completed?". If "No", the flow leads to "Task still in progress". If "Yes", it leads to "Check off task and take off the list" and then to "Finish".

Flowchart 3:

The edit user settings chart begins with "Start", leading to "User settings". A decision point asks "Would you like to edit settings?". If "No", the flow moves to "Display calendar and task list", and then to "Finish". If "Yes", it moves to "Enter login information". A decision point follows asking if "Login credentials correct?". If "No", the flow returns to "Invalid login, try again." If "Yes", it proceeds to "Change user info, notification settings", and then to "Finish".

Flow charts:

Views Implementation:

View 1: Login page

The login page will have a simple form asking for the username and password. Once the user inputs their credentials and submits the form, the system will check against the LoginView to validate the credentials. The LoginView will pull the necessary credentials data from the User table. The application logic will use this view to match the input from the user against the usernames and hashed passwords stored in the database. This view is updatable and thus supports insertions, deletions etc.

```
CREATE VIEW `LoginView` AS
SELECT UserID, Username, Password
FROM User;
```

View 2: Task Management Page

The SimpleTaskManagementView provides a streamlined interface for managing tasks within a specific system. It focuses on the essential aspects of task management by displaying key details of tasks, such as their ID, title, time, duration, and description. This view is extracted solely from the Task table, making it straightforward and efficient for task-related operations. Key functionalities enabled by this view include: Viewing Task Details: Users can easily see important information about their tasks, such as the task title, scheduled time, duration, and a brief description. This makes it simple to get an overview of tasks at a glance. Creating New Tasks: Users can add new tasks to their list. While the view itself is limited to displaying tasks, it can be used in conjunction with insert operations on the Task table to facilitate the addition of new tasks. Editing Existing Tasks: The view allows for modifications to existing tasks. Users can update the title, time, duration, or description of a task. This is particularly useful for adjusting task details as plans change or more information becomes available.T ask Management within a Single Table Context: Since the view is based on a single table (Task), it offers a focused and simplified interface for task management, avoiding the complexities of dealing with multiple related tables and joins. This view is updatable and thus supports insertions, deletions etc.

```
CREATE VIEW `SimpleTaskManagementView` AS
SELECT TaskID, Title, Time, Duration, Description
FROM Task;
```

View 3: User Settings Page

The user settings page will provide a form where users can edit their settings such as notification preferences. After making changes, the user can save the settings which will be updated in the UserSettingsView. The UserSettingsView is a simple representation of the UserSettings table. It will be used to fetch the current settings for display in the form and update them as per the user's changes. This view is updatable and thus supports insetions, deletions etc.

```
CREATE VIEW `UserSettingsView` AS
SELECT UserID, Mode, NotificationSettings
FROM UserSettings;
```

Phase 6.1 Application Development:

Connection to Database:

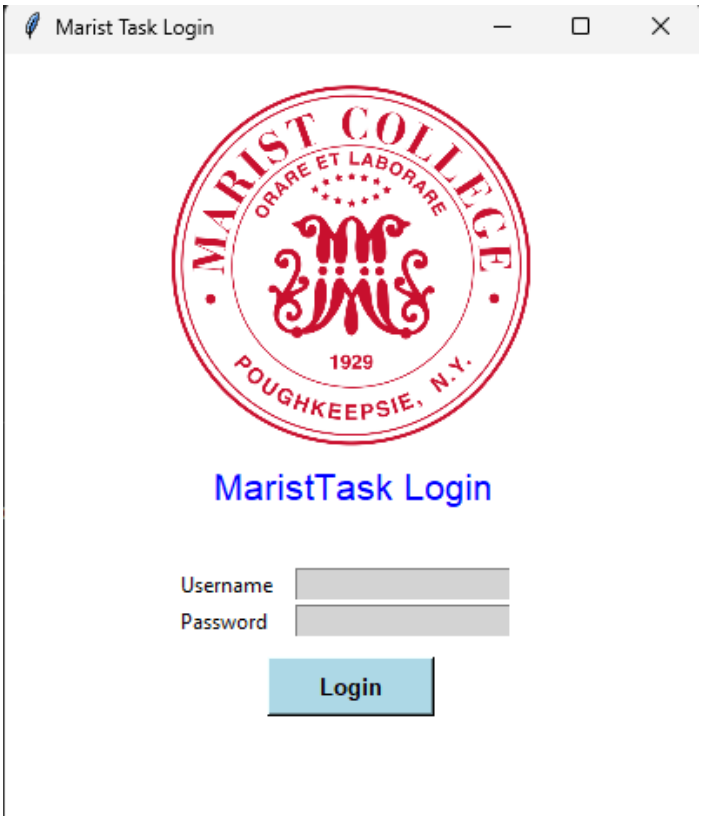
My code connects to a MySQL database using the MySQL.connector Python library. At the beginning of my script, I import the mysql.connector module. This module is part of a Python driver from Oracle that allows me to connect to MySQL databases. The mysql. connector. connect() function is used to establish a connection with my MySQL database. I provide it with the necessary credentials and information such as the host, user, password, and database name. Here, host="localhost" means my MySQL server is running on my local machine. user="root" and passwd="root" are the username and password for the database. database="MaristTask" specifies the name of the database I want to connect to.

```
1   import mysql.connector
2   from tkinter import *
3   from tkinter import messagebox
4   import os
5   from Functions import History, Encrypt, WelcomePage
6
7   # Connect to the database
8   db = mysql.connector.connect(
9       host="localhost",
10      user="root",
11      passwd="root",
12      database="MaristTask"
13  )
```

Login Page:

The code provided below is for a login page interface designed using Python's Tkinter library, integrated with MySQL for user authentication. The program first establishes a connection to a MySQL database named "MaristTask", using predefined credentials. It then defines a function check_login to authenticate users. This function checks whether the provided username and password (encrypted by the Encrypt.encrypt_password function) match either a regular user or an admin in the database, returning the user's role and details upon successful authentication. Upon a successful login attempt, the open_welcome_page function is triggered, closing the current window and opening a welcome page with user details. The login attempt (whether successful or not) is recorded using the History.record_login function.

The GUI is created using Tkinter widgets. It features a window titled "Marist Task Login" with a white background, displaying a logo and a welcome text. The login form includes fields for the username and password, and a customized login button styled with specific text and background colors, font settings, border thickness, and size. The login function is bound to the login button and the 'Enter' key for user convenience. The application enters a GUI loop using mainloop to keep the window responsive. Overall, the code integrates backend database operations with a user-friendly front-end interface, providing secure and efficient user authentication and navigation.



Phase06 > LoginPage.py > ...

```python
import mysql.connector
from tkinter import *
from tkinter import messagebox
import os
from Functions import History, Encrypt, WelcomePage

# Connect to the database
db = mysql.connector.connect(
    host="localhost",
    user="root",
    passwd="root",
    database="MaristTask"
)

# Function to check login credentials and determine role
def check_login(username, input_password):
    cursor = db.cursor()
    encrypted_input_password = Encrypt.encrypt_password(input_password)

    # Check if user is a regular user
    user_query = "SELECT UserID, Name, Password FROM User WHERE Username = %s"
    cursor.execute(user_query, (username,))
    user_result = cursor.fetchone()
```

```
24
25      if user_result and encrypted_input_password == user_result[2]:
26          return True, 'user', user_result[0], user_result[1]
27
28      # Check if user is an admin
29      admin_query = "SELECT AdminID, Name, Password FROM Admin WHERE Username = %s"
30      cursor.execute(admin_query, (username,))
31      admin_result = cursor.fetchone()
32
33      if admin_result and encrypted_input_password == admin_result[2]:
34          return True, 'admin', admin_result[0], admin_result[1]
35
36      return False, None, None, None
37
38  # Function to open the welcome page
39  def open_welcome_page(role, user_id, name, db):
40      window.destroy()  # Close the login window
41      WelcomePage.show_welcome_page(role, user_id, name, db)  # Open the welcome page with additional details
42
43  # Function to handle the login button click
44  # Function to handle the login button click
45  def on_login_click(event=None):
46      username = entry_username.get()
47      password = entry_password.get()
48      login_success, role, user_id, name = check_login(username, password)
49
50      if login_success:
51          messagebox.showinfo("Login Success", f"You have successfully logged in as {role}!")
52          is_admin = role == 'admin'
53          History.record_login(db, user_id, is_admin)  # Record the login time
54          open_welcome_page(role, user_id, name, db)  # Pass the 'db' argument here
55      else:
56          messagebox.showwarning("Login Failed", "Incorrect username or password")
57
58
59
60
61  # Create the main window
62  window = Tk()
63  window.title("Marist Task Login")
64  window.geometry("400x450")  # Adjust the size as needed
65  window.configure(bg='white')  # Set the background color to white
66
67  # Define the relative path to the images directory
68  # Note that 'Images' is capitalized as per your folder structure
69  image_path = os.path.join(os.path.dirname(__file__), 'Images')
70
71  # Load the logo image using a relative path
72  logo_image_path = os.path.join(image_path, 'logo.png')  # 'logo.png' is the image file in the Images directory
73  logo_image = PhotoImage(file=logo_image_path)
74  logo_image = logo_image.subsample(2, 2)  # Adjust the subsample factors as needed
75
76  # Use the scaled image in the Label widget
77  logo_label = Label(window, image=logo_image, bg='white')  # Set the background color to white
78  logo_label.pack(side=TOP, pady=(10, 0))
79
```

```
78  logo_label.pack(side=TOP, pady=(10, 0))
79
80  # Welcome label
81  welcome_text = Label(window, text="MaristTask Login", fg="blue", bg='white', font=("Helvetica", 16))
82  welcome_text.pack(side=TOP, pady=(0, 20))  # Adjust padding as needed
83
84  # Create and place the login form
85  form_frame = Frame(window, bg='white')  # Set the background color to white
86  form_frame.pack(side=TOP, pady=10)
87
88  label_username = Label(form_frame, text="Username", bg='white')
89  label_username.grid(row=0, column=0, sticky=W)
90
91  entry_username = Entry(form_frame, bg='#D3D3D3')
92  entry_username.grid(row=0, column=1, padx=10)
93
94  label_password = Label(form_frame, text="Password", bg='white')
95  label_password.grid(row=1, column=0, sticky=W)
96
97  entry_password = Entry(form_frame, show="*", bg='#D3D3D3')
98  entry_password.grid(row=1, column=1, padx=10)
99
100 # Modified login button with new properties
101 button_login = Button(form_frame, text="Login", command=on_login_click,
102                       fg="black",  # Text color
103                       bg="#ADD8E6",  # Background color
104                       font=("Helvetica", 10, "bold"),  # Font settings (size, weight)
105                       highlightthickness=4,  # Border thickness
106                       width=10, height=1)  # Button size
107 button_login.grid(row=2, column=0, columnspan=2, pady=10)
108
109
110 # Bind the enter key to the login function
111 window.bind('<Return>', on_login_click)
112
113 # Start the GUI loop
114 window.mainloop()
115
```

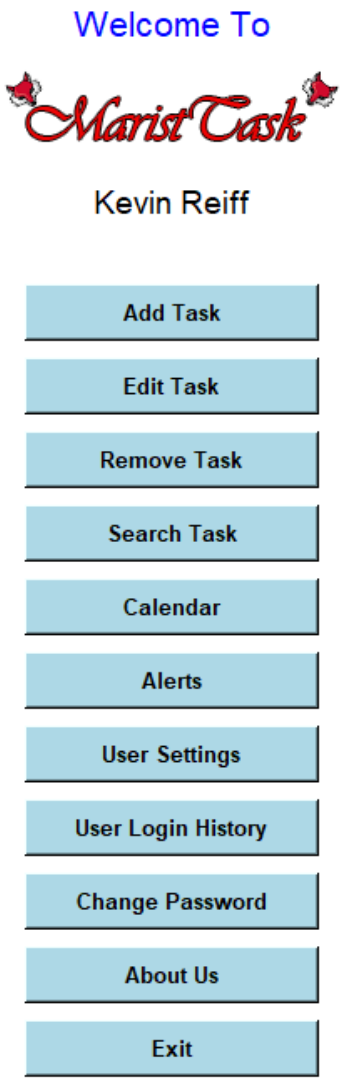
Main Menu Page:

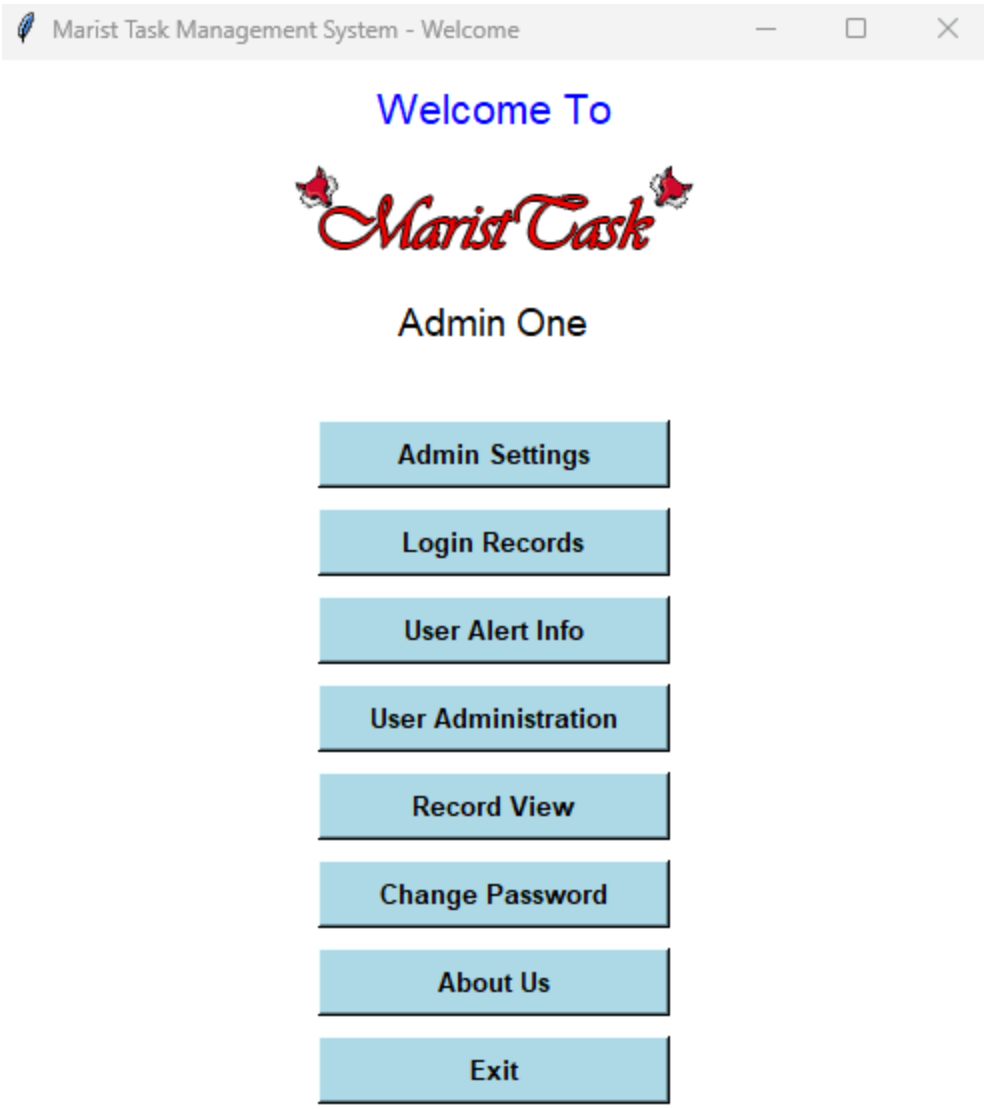
This code defines a Main Menu page for the "Marist Task Management System," using the Tkinter library in Python. The function show_welcome_page is designed to create a main window with a user interface tailored to the role of the logged-in user (admin or regular user). The window's title is set, and its size varies depending on the user's role—smaller for admins and larger for regular users. The background color is set to white, and the user is greeted with a welcome label displaying their name.

The UI features a series of buttons organized in a frame, with a common style defined for all buttons. These buttons are linked to various functionalities like task management (add, edit, remove, search tasks), calendar, alerts, settings, and history. For admins, additional buttons like 'Admin Settings', 'Login Records', 'User Alert Info', 'User Administration', and 'Record View' are

provided. A 'Change Password' button is available for all users. An 'Exit' button is included to close the application, and upon closing, a message is displayed thanking the user for using the software, and the logout time is recorded in the history.

The interface adapts based on the user's role, displaying different sets of buttons and functionalities. For regular users, there's also a check for alerts when the window is opened. The GUI loop starts at the end of the function, initiating the user interface. This design demonstrates a dynamic, role-based UI approach in a desktop application.




Marist Task Management System - Welcome
Welcome To
Marist Task
Admin One
Admin Settings
Login Records
User Alert Info
User Administration
Record View
Change Password
About Us
Exit

```python
from tkinter import Tk, Label, Button, Frame, messagebox
from Functions import ChangePassword, UserAdministration, AddTask, EditTask, RemoveTask, SearchTask, Calendar, Alerts, Settings, History, Records

def show_welcome_page(role, user_id, name, db):
    # Create the main window
    welcome_window = Tk()
    welcome_window.title("Marist Task Management System - Welcome")

    # Adjust the size based on the role
    if role == 'admin':
        welcome_window.geometry("500x450")  # Smaller size for admin
    else:
        welcome_window.geometry("600x750")  # Larger size for regular user

    welcome_window.configure(bg='white')  # Set the background color to white

    # Welcome label with user's name
    welcome_label = Label(welcome_window, text=f"Welcome to MaristTask, {name}!",
                          fg="blue", bg='white', font=("Helvetica", 16))
    welcome_label.pack(pady=10)

    # Display user ID (for debugging or reference)
    user_id_label = Label(welcome_window, text=f"User ID: {user_id}",
                          fg="red", bg='white', font=("Helvetica", 12))
    user_id_label.pack(pady=5)

    # Button frame
    button_frame = Frame(welcome_window, bg='white')
    button_frame.pack(pady=20)

    # Define a common button style
    button_style = {'fg': "black", 'bg': "#ADD8E6", 'font': ("Helvetica", 10, "bold"),
                    'highlightthickness': 4, 'width': 20, 'height': 1}

    def on_close():
        """ Function to handle window close event """
        is_admin = role == 'admin'
        History.record_logout(db, user_id, is_admin)  # Record the logout time
        messagebox.showinfo("Thank You", "Thank you for using the MaristTask software!")
        welcome_window.destroy()  # Close the window

    welcome_window.protocol("WM_DELETE_WINDOW", on_close)  # Bind the close event

    # Role-specific features and buttons
    if role == 'admin':
        # Admin-specific buttons
        btn_admin_settings = Button(button_frame, text="Admin Settings", **button_style,
                                    command=lambda: Settings.settings_window(db, user_id, role))
        btn_admin_settings.grid(row=0, column=0, padx=10, pady=5)
```

```
50
51          btn_admin_login_history = Button(button_frame, text="Login Records", **button_style,
52                                           command=lambda: History.show_admin_login_records(db))
53          btn_admin_login_history.grid(row=1, column=0, padx=10, pady=5)
54
55          btn_user_alert_info = Button(button_frame, text="User Alert Info", **button_style,
56                                       command=lambda: Alerts.show_user_alert_info(db))
57          btn_user_alert_info.grid(row=2, column=0, padx=10, pady=5)
58
59          btn_user_admin = Button(button_frame, text="User Administration", **button_style,
60                                  command=lambda: UserAdministration.user_administration(db))
61          btn_user_admin.grid(row=3, column=0, padx=10, pady=5)
62
63          btn_record_view = Button(button_frame, text="Record View", **button_style,
64                                   command=lambda: Records.record_view(db))
65          btn_record_view.grid(row=4, column=0, padx=10, pady=5)
66
67      else:
68          # User-specific buttons
69          btn_add_task = Button(button_frame, text="Add Task", **button_style,
70                                command=lambda: AddTask.add_task(db, user_id))
71          btn_add_task.grid(row=0, column=0, padx=10, pady=5)
72
73          btn_edit_task = Button(button_frame, text="Edit Task", **button_style,
74                                 command=lambda: EditTask.edit_task(db, user_id))
75          btn_edit_task.grid(row=1, column=0, padx=10, pady=5)
76
77          btn_remove_task = Button(button_frame, text="Remove Task", **button_style,
78                                   command=lambda: RemoveTask.remove_task(db, user_id))
79          btn_remove_task.grid(row=2, column=0, padx=10, pady=5)
80
81          btn_search_task = Button(button_frame, text="Search Task", **button_style,
82                                   command=lambda: SearchTask.search_task(db, user_id))
83          btn_search_task.grid(row=3, column=0, padx=10, pady=5)
84
85          default_calendar_id = 1
86          btn_calendar = Button(button_frame, text="Calendar", **button_style,
87                                command=lambda: Calendar.show_calendar(db, user_id, default_calendar_id))
88          btn_calendar.grid(row=4, column=0, padx=10, pady=5)
89
90          btn_alerts = Button(button_frame, text="Alerts", **button_style,
91                              command=lambda: Alerts.show_user_alerts(db, user_id))
92          btn_alerts.grid(row=5, column=0, padx=10, pady=5)
93
94          btn_user_settings = Button(button_frame, text="User Settings", **button_style,
95                                     command=lambda: Settings.settings_window(db, user_id, role))
96          btn_user_settings.grid(row=6, column=0, padx=10, pady=5)
```

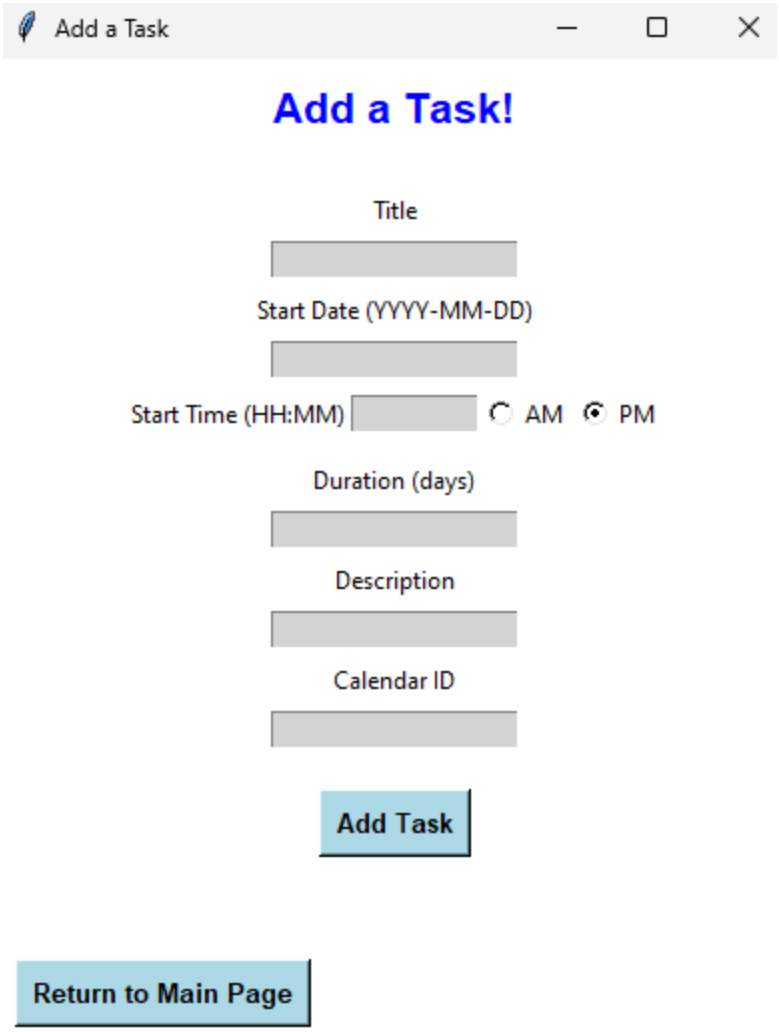
```
        btn_user_settings.grid(row=6, column=0, padx=10, pady=5)

        btn_user_login_history = Button(button_frame, text="User Login History", **button_style,
                                        command=lambda: History.show_login_history(db, user_id, False))
        btn_user_login_history.grid(row=7, column=0, padx=10, pady=5)

    # Change Password Button (for all users)
    btn_change_password = Button(button_frame, text="Change Password", **button_style,
                                 command=lambda: ChangePassword.change_password(db, user_id, role))
    btn_change_password.grid(row=8 if role == 'admin' else 9, column=0, padx=10, pady=5)

    # Exit button
    btn_exit = Button(button_frame, text="Exit", command=on_close, **button_style)
    btn_exit.grid(row=9 if role == 'admin' else 10, column=0, padx=10, pady=5)

    # Check for alerts (if user)
    if role != 'admin':
        Alerts.welcome_alerts(db, user_id)

    # Start the GUI loop
    welcome_window.mainloop()

```

Action Pages:

Figure 1: Add Task

The code below defines a Python function add_task that creates a graphical user interface (GUI) for adding tasks to a database, using Tkinter for the GUI components and MySQL for database operations. The function is designed to be part of a larger application where each user can add tasks to their calendar. The GUI window, titled "Add a Task", contains several input fields for task details: title, start date, start time (with AM/PM selection), duration (in days), description, and a calendar ID. These fields are implemented using labels and entry widgets, with a special frame for the start time and AM/PM radio buttons. The start date and time inputs are validated and formatted to fit the required database format. An error message is displayed if the date or time format is invalid.

The user can submit the task by clicking the "Add Task" button or pressing the Enter key. The submission process involves checking that all fields are filled, executing an SQL query to insert the task into the database, and handling any potential errors. On successful addition, a success message is shown, and the add task window is closed. There's also a "Return to Main Page" button for closing the window without adding a task.


Add a Task
Add a Task!
Title
Start Date (YYYY-MM-DD)
Start Time (HH:MM)
AM
PM
Duration (days)
Description
Calendar ID
Add Task
Return to Main Page

```python
import mysql.connector
from tkinter import Radiobutton, Tk, Label, Entry, Button, messagebox, StringVar, Frame
import datetime

def add_task(db, user_id):
    # Function to handle the task submission process.
    def submit_task(event=None):
        # Retrieve values from the form fields.
        title = entry_title.get()
        start_date = entry_start_date.get()
        start_time = entry_start_time.get() + " " + am_pm.get()
        duration = entry_duration.get()
        description = entry_description.get()
        calendar_id = entry_calendar_id.get()

        # Attempt to format and validate the start date and time.
        try:
            formatted_start_time = datetime.datetime.strptime(f"{start_date} {start_time}", '%Y-%m-%d %I:%M %p')
            formatted_start_time_str = formatted_start_time.strftime('%Y-%m-%d %H:%M:%S')
        except ValueError:
            messagebox.showerror("Error", "Invalid date or time format")
            return

        # Check if all form fields are filled.
        if not title or not start_date or not start_time or not duration or not description or not calendar_id:
            messagebox.showerror("Error", "All fields are required")
            return

        # Create a database cursor for executing SQL queries.
        cursor = db.cursor()
        # SQL query to insert a new task into the database.
        insert_query = """
            INSERT INTO Task (UserID, Title, Time, Duration, Description, Calendar_CalendarID)
            VALUES (%s, %s, %s, %s, %s, %s)
        """
        try:
            # Execute the insert query with user-provided values.
            cursor.execute(insert_query, (user_id, title, formatted_start_time_str, int(duration) * 1440, description, calendar_id))
            db.commit()
            messagebox.showinfo("Success", "Task added successfully")
            add_window.destroy()  # Close the window after successful task addition.
        except mysql.connector.Error as err:
            messagebox.showerror("Error", f"An error occurred: {err}")
        finally:
            cursor.close()

    # Set up the GUI window for adding a task.
    add_window = Tk()
    add_window.title("Add a Task")
    add_window.geometry("400x500")
    add_window.configure(bg='white')

    label_main_title = Label(add_window, text="Add a Task!", bg='white', fg='blue', font=("Helvetica", 16, "bold"))
    label_main_title.pack(pady=(10, 20))

    # Widgets for task title input.
    label_title = Label(add_window, text="Title", bg='white')
    label_title.pack(pady=(5, 5))
    entry_title = Entry(add_window, bg='#D3D3D3')
    entry_title.pack()

    # Widgets for task start date input.
    label_start_date = Label(add_window, text="Start Date (YYYY-MM-DD)", bg='white')
    label_start_date.pack(pady=5)
    entry_start_date = Entry(add_window, bg='#D3D3D3')
    entry_start_date.pack()
```

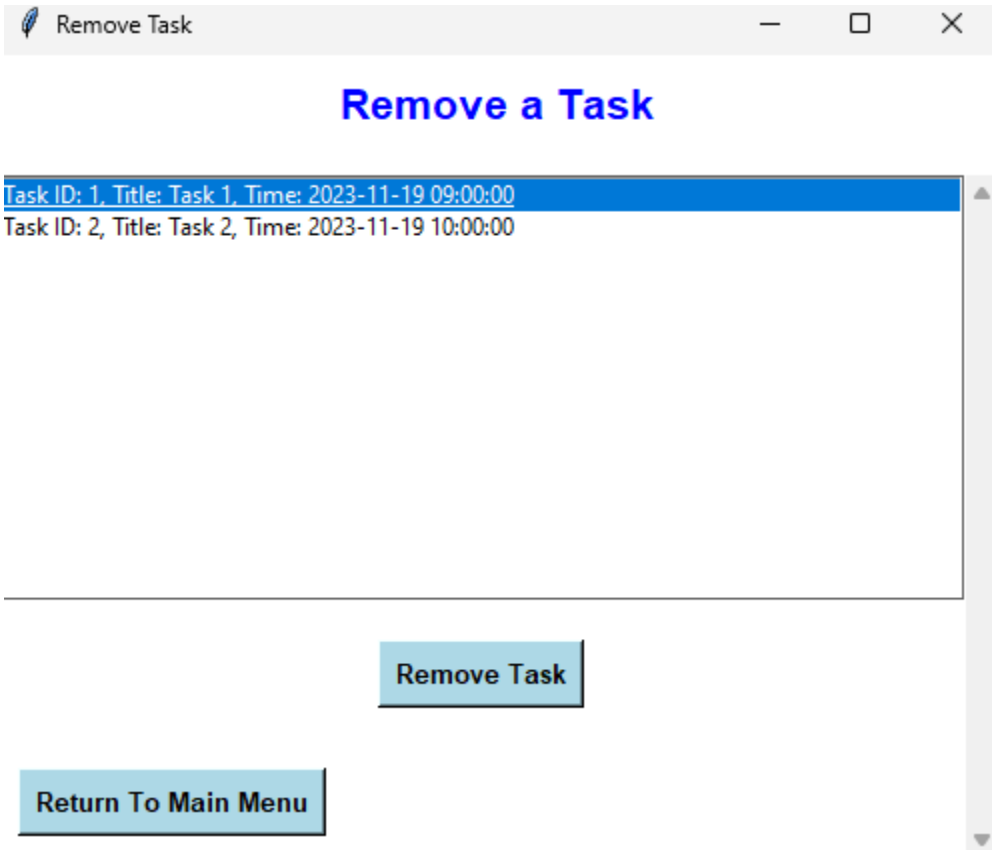
```
    label_start_date.pack(pady=5)
    entry_start_date = Entry(add_window, bg='#D3D3D3')
    entry_start_date.pack()

    # Frame for time input and AM/PM selection.
    time_frame = Frame(add_window, bg='white')
    time_frame.pack(pady=5)

    label_start_time = Label(time_frame, text="Start Time (HH:MM)", bg='white')
    label_start_time.pack(side="left")
    entry_start_time = Entry(time_frame, bg='#D3D3D3', width=10)
    entry_start_time.pack(side="left")

    # Radio buttons for AM/PM selection.
    am_pm = StringVar(value="AM")
    rb_am = Radiobutton(time_frame, text="AM", variable=am_pm, value="AM", bg='white')
    rb_am.pack(side="left")
    rb_pm = Radiobutton(time_frame, text="PM", variable=am_pm, value="PM", bg='white')
    rb_pm.pack(side="left")

    # Widgets for task duration input.
    label_duration = Label(add_window, text="Duration (days)", bg='white')
    label_duration.pack(pady=5)
    entry_duration = Entry(add_window, bg='#D3D3D3')
    entry_duration.pack()

    # Widgets for task description input.
    label_description = Label(add_window, text="Description", bg='white')
    label_description.pack(pady=5)
    entry_description = Entry(add_window, bg='#D3D3D3')
    entry_description.pack()

    # Widgets for calendar ID input.
    label_calendar_id = Label(add_window, text="Calendar ID", bg='white')
    label_calendar_id.pack(pady=5)
    entry_calendar_id = Entry(add_window, bg='#D3D3D3')
    entry_calendar_id.pack()

    # Button to submit the task information.
    button_submit = Button(add_window, text="Add Task", fg="black", bg="#ADD8E6",
                           font=("Helvetica", 10, "bold"), highlightthickness=4, command=submit_task)
    button_submit.pack(pady=20)

    # Button to close the add task window.
    button_return = Button(add_window, text="Return to Main Page", fg="black", bg="#ADD8E6",
                           font=("Helvetica", 10, "bold"), highlightthickness=4, command=add_window.destroy)
    button_return.place(x=10, y=450)

    # Enable submission of the task by pressing the Enter key.
    add_window.bind('<Return>', submit_task)

    add_window.mainloop()
```

Figure 2: Remove Task

This code describes a Python application for managing tasks, with a focus on a feature that allows users to remove tasks. It uses mysql. connector to interact with a MySQL database and tkinter for the graphical user interface (GUI). The central functionality is encapsulated in the remove_task function, which creates a window where tasks associated with a given user ID can be viewed and selectively removed. This function has two inner functions: refresh_task_list and select_task_to_remove. refresh_task_list queries the database to retrieve tasks for the specified user, displaying them in a Listbox widget with an associated scrollbar for navigation. Each task's

ID, title, and time are displayed, and the IDs are stored in a list for further operations. The select_task_to_remove function allows the user to select a task from the list and delete it from the database after confirmation, with error handling for any issues during the deletion process.

The GUI is composed of a main window titled "Remove Task", a title label, a task display Listbox with a scrollbar, and buttons for removing a selected task and returning to the main menu. The layout is designed for ease of use, with clear labels and a simple, intuitive interface. The application emphasizes user interaction and confirmation for task removal, ensuring a user-friendly experience with necessary precautions against accidental deletions.



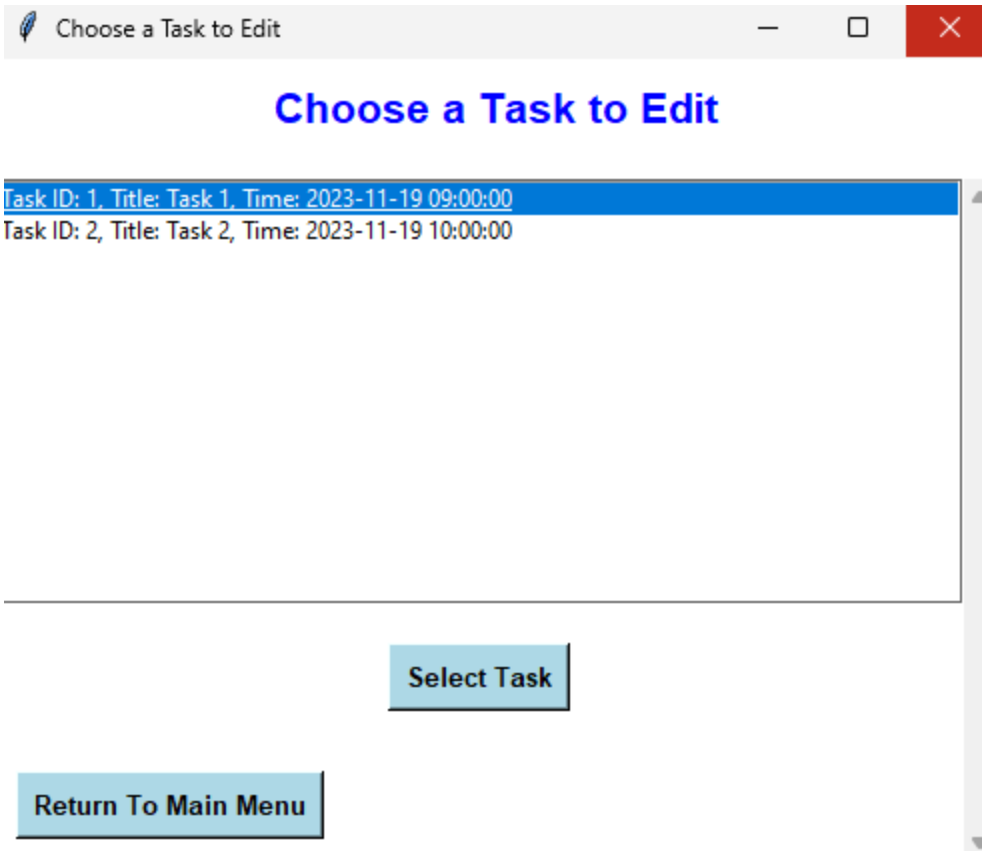
```python
import mysql.connector
from tkinter import Tk, Label, Button, Frame, Listbox, Scrollbar, messagebox

def remove_task(db, user_id):
    # Function to update the task list in the GUI
    def refresh_task_list():
        # Clear existing items in the listbox
        listbox_tasks.delete(0, 'end')
        # Clear the list that tracks task IDs
        task_ids.clear()
        # Create a database cursor for querying
        cursor = db.cursor()
        # Retrieve tasks specific to the given user ID
        cursor.execute("SELECT TaskID, Title, Time FROM Task WHERE UserID = %s", (user_id,))
        # Populate the listbox with tasks and store their IDs
        for task_id, title, time in cursor:
            listbox_tasks.insert("end", f"Task ID: {task_id}, Title: {title}, Time: {time}")
            task_ids.append(task_id)
        # Close the cursor to free database resources
        cursor.close()

    # Function to handle task removal
    def select_task_to_remove():
        # Get the selected item's index from the listbox
        selected_index = listbox_tasks.curselection()
        # Check if an item is selected
        if not selected_index:
            messagebox.showwarning("Warning", "Please select a task to remove")
            return
        # Retrieve the task ID corresponding to the selected item
        task_id = task_ids[selected_index[0]]

        # Ask for confirmation before deleting the task
        if messagebox.askyesno("Confirm Deletion", "Are you sure you want to delete this task?"):
            cursor = db.cursor()
            delete_query = "DELETE FROM Task WHERE TaskID = %s"
            try:
                # Execute the deletion query
                cursor.execute(delete_query, (task_id,))
                db.commit()
                messagebox.showinfo("Success", "Task removed successfully")
                # Refresh the list to reflect the deletion
                refresh_task_list()
            except mysql.connector.Error as err:
                messagebox.showerror("Error", f"An error occurred: {err}")
            finally:
                # Always close the cursor after the operation
                cursor.close()

    # Create the main window using Tkinter
    main_window = Tk()
    main_window.title("Remove Task")
    main_window.geometry("500x400")
    main_window.configure(bg='white')

    # Setup the title label
    label_main_title = Label(main_window, text="Remove a Task", bg='white', fg='blue', font=("Helvetica", 16, "bold"))
    label_main_title.pack(pady=(10, 20))

    # Initialize a scrollbar for the listbox
```

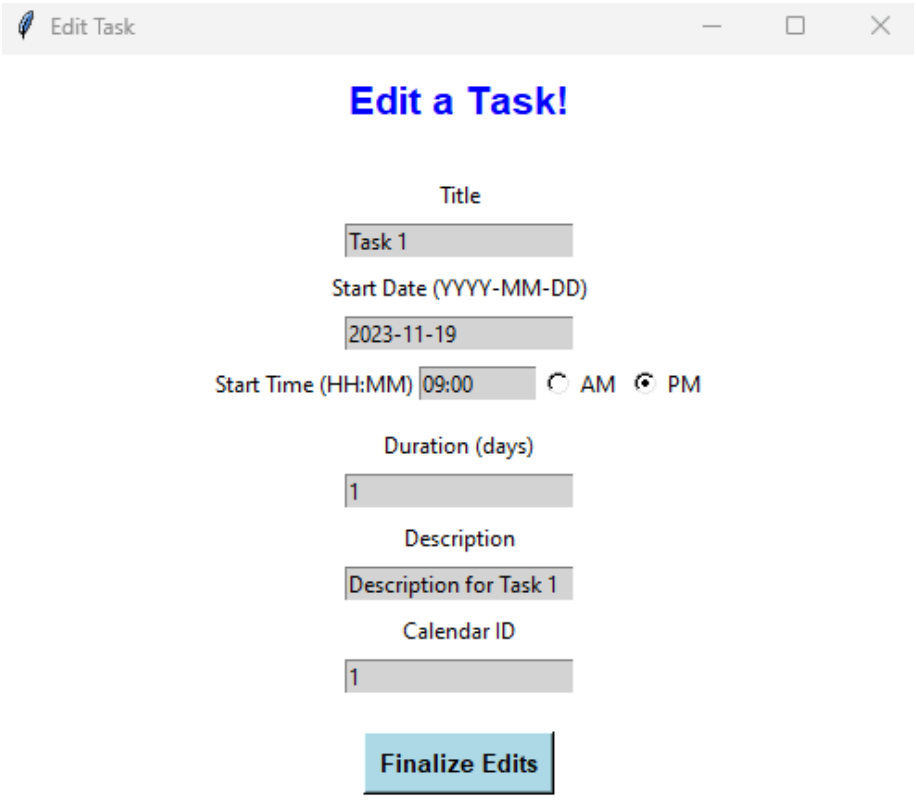
```
    # Initialize a scrollbar for the listbox
    scrollbar = Scrollbar(main_window)
    scrollbar.pack(side="right", fill="y")

    # Create a listbox to display tasks and attach the scrollbar to it
    listbox_tasks = Listbox(main_window, yscrollcommand=scrollbar.set)
    task_ids = []  # List to store task IDs
    refresh_task_list()  # Populate the listbox with tasks

    listbox_tasks.pack(fill="both", expand=True)
    scrollbar.config(command=listbox_tasks.yview)

    # Button to trigger task removal
    button_remove = Button(main_window, text="Remove Task", fg="black", bg="#ADD8E6",
                           font=("Helvetica", 10, "bold"), highlightthickness=4, command=select_task_to_remove)
    button_remove.pack(pady=20)

    # Button to return to the main menu
    button_return = Button(main_window, text="Return To Main Menu", fg="black", bg="#ADD8E6",
                           font=("Helvetica", 10, "bold"), highlightthickness=4, command=main_window.destroy)
    button_return.pack(side='left', padx=10, pady=10)

    # Start the Tkinter event loop
    main_window.mainloop()
```

Figure 3: Edit Task

The below code defines a Python application for editing tasks in a database, using Tkinter for the graphical user interface and MySQL.connector for database operations. The program is designed for a user to select and edit tasks from a database associated with their user ID. The main function, edit_task, initializes the user interface and provides functionalities for listing, selecting, and editing tasks. It consists of several nested functions: refresh_task_list: This function populates a list box with tasks retrieved from the database. Each task includes its ID, title, and time. This list aids users in selecting a task to edit. select_task_to_edit: Triggered when a user selects a task from the list. It opens a new window for editing the chosen task. If no task is selected, it shows a warning message. edit_selected_task: This function creates an interface for editing the selected task. It displays current task details in various entry fields, including title, start date and time, duration, description, and calendar ID. The start time is formatted for ease of editing, and the duration is converted from minutes to days. finalize_edits: This function saves the edited task back to the database. It validates and formats the new start time and converts the duration back to minutes. If successful, it updates the task in the database and refreshes the task list; otherwise, it displays an error message.

The main window (main_window) contains a list box for displaying tasks, a scrollbar for navigation, and buttons for selecting a task to edit and returning to the main menu. The edit window (edit_window) offers fields for editing the task's details and a button to finalize the edits.


Choose a Task to Edit
Choose a Task to Edit
Task ID: 1, Title: Task 1, Time: 2023-11-19 09:00:00
Task ID: 2, Title: Task 2, Time: 2023-11-19 10:00:00
Select Task
Return To Main Menu


Edit Task
Edit a Task!
Title
Task 1
Start Date (YYYY-MM-DD)
2023-11-19
Start Time (HH:MM) 09:00
AM
PM
Duration (days)
1
Description
Description for Task 1
Calendar ID
1
Finalize Edits

```python
import mysql.connector
from tkinter import Radiobutton, Tk, Label, Entry, Button, messagebox, StringVar, Frame, Listbox, Scrollbar, Toplevel
import datetime

def edit_task(db, user_id):
    # Function to refresh the task list displayed in the main window.
    def refresh_task_list():
        listbox_tasks.delete(0, 'end')  # Clear existing items in the listbox.
        task_ids.clear()  # Clear the task ID list.
        cursor = db.cursor()  # Create a new cursor for database operations.
        cursor.execute("SELECT TaskID, Title, Time FROM Task WHERE UserID = %s", (user_id,))
        # Populate the listbox with tasks and store their IDs for later use.
        for task_id, title, time in cursor:
            listbox_tasks.insert("end", f"Task ID: {task_id}, Title: {title}, Time: {time}")
            task_ids.append(task_id)
        cursor.close()

    # Function to handle the selection of a task to edit.
    def select_task_to_edit():
        selected_index = listbox_tasks.curselection()  # Get the index of the selected task.
        if not selected_index:  # Check if a task is selected.
            messagebox.showwarning("Warning", "Please select a task to edit")
            return
        task_id = task_ids[selected_index[0]]  # Retrieve the ID of the selected task.
        edit_window = Toplevel(main_window)  # Create a new top-level window for editing.
        edit_selected_task(edit_window, task_id)  # Call function to handle editing.

    # Function to edit the selected task.
    def edit_selected_task(edit_window, task_id):
        cursor = db.cursor()
        # Fetch existing details of the selected task from the database.
        cursor.execute("SELECT Title, Time, Duration, Description, Calendar_CalendarID FROM Task WHERE TaskID = %s", (task_id,))
        task = cursor.fetchone()
        cursor.close()

        # Unpack the task details.
        title, time, duration, description, calendar_id = task
        # Format the start date and time for display.
        start_date = time.strftime('%Y-%m-%d')
        start_time = time.strftime('%I:%M %p')
        duration_days = duration // 1440  # Convert duration from minutes to days.

        # Function to finalize and save the edits to the task.
        def finalize_edits():
            # Fetch new task details from the entry fields.
            new_title = entry_title.get()
            new_start_date = entry_start_date.get()
            new_start_time = entry_start_time.get() + " " + am_pm.get()
            new_duration_days = entry_duration.get()
            new_description = entry_description.get()
            new_calendar_id = entry_calendar_id.get()

            # Attempt to format and validate the new start time.
            try:
                formatted_start_time = datetime.datetime.strptime(f"{new_start_date} {new_start_time}", '%Y-%m-%d %I:%M %p')
                formatted_duration = int(new_duration_days) * 1440  # Convert days back to minutes.
            except ValueError:
                messagebox.showerror("Error", "Invalid date or time format")
                return

            # Prepare the SQL query for updating the task.
```

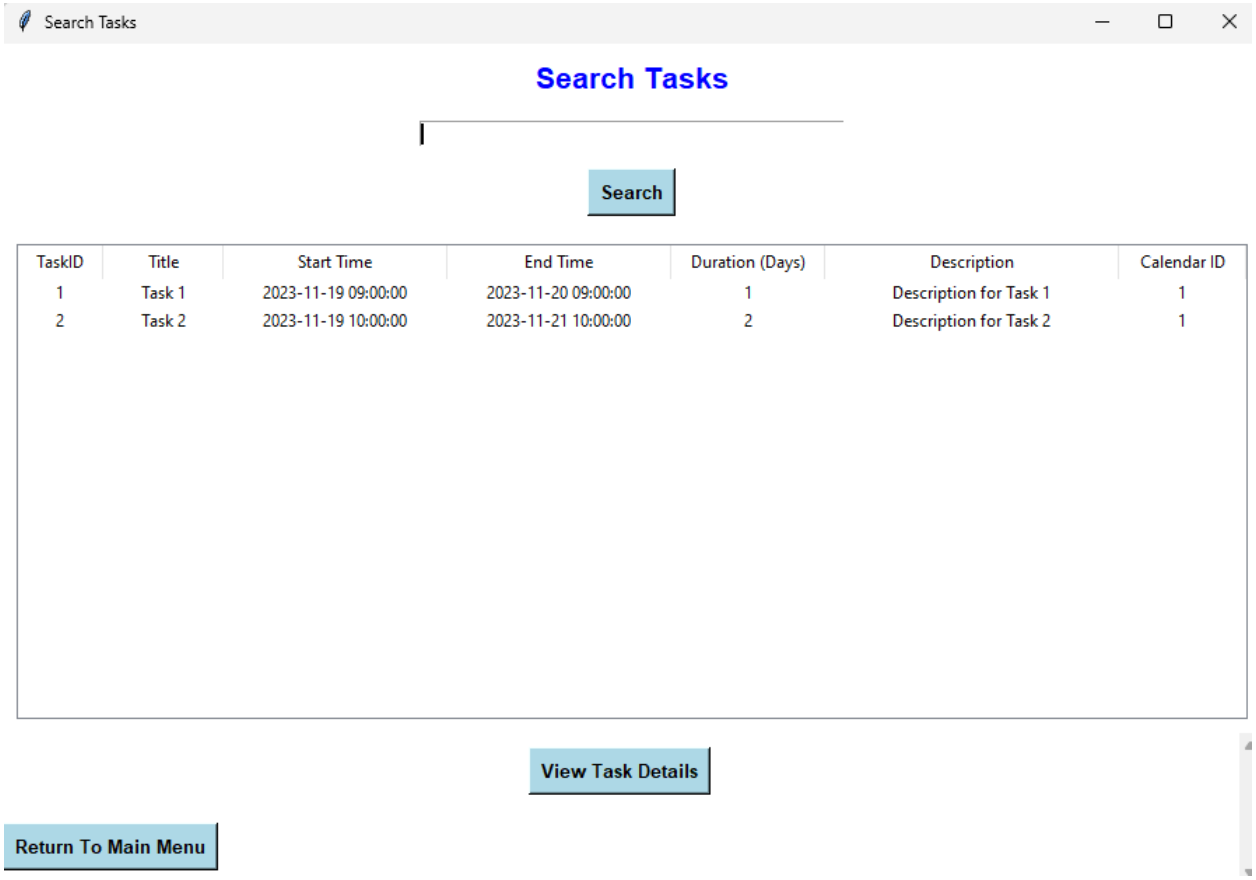
```
            # Prepare the SQL query for updating the task.
            update_query = """
                UPDATE Task SET Title = %s, Time = %s, Duration = %s, Description = %s, Calendar_CalendarID = %s
                WHERE TaskID = %s
            """
            try:
                cursor = db.cursor()
                # Execute the update query with the new task details.
                cursor.execute(update_query, (new_title, formatted_start_time, formatted_duration, new_description, int(new_calendar_id), task_id))
                db.commit()  # Commit the changes to the database.
                messagebox.showinfo("Success", "Task updated successfully")
                edit_window.destroy()  # Close the edit window.
                refresh_task_list()  # Refresh the task list to show updated details.
            except mysql.connector.Error as err:
                messagebox.showerror("Error", f"An error occurred: {err}")
            finally:
                cursor.close()

        # Setting up the edit window layout with entry fields for task details.
        edit_window.title("Edit Task")
        edit_window.geometry("500x500")
        edit_window.configure(bg='white')

        # Main title caption
        label_main_title = Label(edit_window, text="Edit a Task!", bg='white', fg='blue', font=("Helvetica", 16, "bold"))
        label_main_title.pack(pady=(10, 20))

        # Task Title
        label_title = Label(edit_window, text="Title", bg='white')
        label_title.pack(pady=(5, 5))
        entry_title = Entry(edit_window, bg='#D3D3D3')
        entry_title.insert(0, title)
        entry_title.pack()

        # Task Start Date
        label_start_date = Label(edit_window, text="Start Date (YYYY-MM-DD)", bg='white')
        label_start_date.pack(pady=5)
        entry_start_date = Entry(edit_window, bg='#D3D3D3')
        entry_start_date.insert(0, start_date)
        entry_start_date.pack()

        # Time Frame
        time_frame = Frame(edit_window, bg='white')
        time_frame.pack(pady=5)

        # Task Start Time
        label_start_time = Label(time_frame, text="Start Time (HH:MM)", bg='white')
        label_start_time.pack(side="left")
        entry_start_time = Entry(time_frame, bg='#D3D3D3', width=10)
        entry_start_time.insert(0, start_time.split()[0])
        entry_start_time.pack(side="left")

        # AM/PM Selection
        am_pm = StringVar(value=start_time.split()[1])
        Radiobutton(time_frame, text="AM", variable=am_pm, value="AM", bg='white', command=lambda: am_pm.set("AM")).pack(side="left")
        Radiobutton(time_frame, text="PM", variable=am_pm, value="PM", bg='white', command=lambda: am_pm.set("PM")).pack(side="left")

        # Task Duration in Days
        label_duration = Label(edit_window, text="Duration (days)", bg='white')
        label_duration.pack(pady=5)
```

```
        entry_duration = Entry(edit_window, bg='#D3D3D3')
        entry_duration.insert(0, duration_days)
        entry_duration.pack()

        # Task Description
        label_description = Label(edit_window, text="Description", bg='white')
        label_description.pack(pady=5)
        entry_description = Entry(edit_window, bg='#D3D3D3')
        entry_description.insert(0, description)
        entry_description.pack()

        # Calendar ID
        label_calendar_id = Label(edit_window, text="Calendar ID", bg='white')
        label_calendar_id.pack(pady=5)
        entry_calendar_id = Entry(edit_window, bg='#D3D3D3')
        entry_calendar_id.insert(0, calendar_id)
        entry_calendar_id.pack()

        # Finalize Edits button
        button_finalize = Button(edit_window, text="Finalize Edits", fg="black", bg="#ADD8E6",
                                 font=("Helvetica", 10, "bold"), highlightthickness=4, command=finalize_edits)
        button_finalize.pack(pady=20)

    # Main window for task selection
    main_window = Tk()
    main_window.title("Choose a Task to Edit")
    main_window.geometry("500x400")
    main_window.configure(bg='white')

    # Title caption
    label_main_title = Label(main_window, text="Choose a Task to Edit", bg='white', fg='blue', font=("Helvetica", 16, "bold"))
    label_main_title.pack(pady=(10, 20))

    # Listbox with tasks
    scrollbar = Scrollbar(main_window)
    scrollbar.pack(side="right", fill="y")

    listbox_tasks = Listbox(main_window, yscrollcommand=scrollbar.set)
    task_ids = []
    refresh_task_list()

    listbox_tasks.pack(fill="both", expand=True)
    scrollbar.config(command=listbox_tasks.yview)

    # Select Task button
    button_select = Button(main_window, text="Select Task", fg="black", bg="#ADD8E6",
                           font=("Helvetica", 10, "bold"), highlightthickness=4, command=select_task_to_edit)
    button_select.pack(pady=20)

    def return_to_main_menu():
        main_window.destroy()

    button_return = Button(main_window, text="Return To Main Menu", fg="black", bg="#ADD8E6",
                           font=("Helvetica", 10, "bold"), highlightthickness=4, command=return_to_main_menu)
    button_return.pack(side='bottom', anchor='w', padx=10, pady=10)  # Placed at bottom left

    main_window.mainloop()
```

Figure 4: Search Task

This code defines a search_task function for a task management application, utilizing a MySQL database and a graphical user interface built with Tkinter in Python. The primary purpose of the function is to allow users to search for tasks based on various criteria and display detailed information about these tasks. The search_task function, designed for a specific user (user_id), incorporates several key features:

- Search Functionality: Users can input search criteria in a text entry field. The search encompasses multiple fields in the Task table, such as title, description, calendar ID, task ID, time, and duration. The results are fetched from the database and displayed in a Treeview widget, allowing users to view summarized task details.
- Task Detail Display: Clicking on a task in the search results opens a new window displaying detailed information about the task. This includes the task's title, start and end times, duration (converted from minutes to days), description, and associated calendar ID.
- Data Handling and Display: The application connects to a MySQL database to retrieve task data based on the search query. It uses functions to calculate the end time of tasks based on their duration and start time and employs a Treeview with a scrollbar for result display.
- User Interface Components: The Tkinter library is used to create a user-friendly interface, including labels, entry fields, buttons, and frames. There's a main window titled "Search Tasks," where users can input search terms, view results, and access detailed task information.
- Navigation and Layout: The layout is organized with clear labels and buttons for searching tasks, viewing task details, and returning to the main menu. The application is designed to be interactive and easy to navigate.


Search Tasks
Search Tasks
Search
TaskID Title Start Time End Time Duration (Days) Description Calendar ID
1 Task 1 2023-11-19 09:00:00 2023-11-20 09:00:00 1 Description for Task 1 1
2 Task 2 2023-11-19 10:00:00 2023-11-21 10:00:00 2 Description for Task 2 1
View Task Details
Return To Main Menu

```python
import mysql.connector
from tkinter import Tk, Label, Entry, Button, Frame, Toplevel, messagebox
from tkinter.ttk import Treeview, Scrollbar
import datetime

def search_task(db, user_id):
    # Function to calculate the end time of a task based on its start time and duration
    def calculate_end_date(start_time, duration):
        end_time = start_time + datetime.timedelta(minutes=duration)
        return end_time

    # Function to display detailed information of a specific task in a new window
    def display_task_details(task_id):
        # Creating a new top-level window for task details
        details_window = Toplevel(main_window)
        details_window.title("Task Information")
        details_window.geometry("400x500")
        details_window.configure(bg='white')

        # Querying the database for task details using the task ID
        cursor = db.cursor()
        cursor.execute("SELECT Title, Time, Duration, Description, Calendar_CalendarID FROM Task WHERE TaskID = %s", (task_id,))
        task = cursor.fetchone()
        cursor.close()

        # Checking if task details are found and displaying them
        if task:
            title, time, duration, description, calendar_id = task
            duration_days = duration // 1440  # Converting duration from minutes to days
            end_date = calculate_end_date(time, duration)

            # Creating and configuring frames and labels to display task details
            header_frame = Frame(details_window, bg='white')
            header_frame.pack(pady=(10, 20), padx=10, fill='x')
            Label(header_frame, text="Task Information", bg='white', fg='blue', font=("Helvetica", 16, "bold")).pack(side='left')

            details_frame = Frame(details_window, bg='white')
            details_frame.pack(pady=(5, 10), padx=10, fill='x')

            # Labels and values for each task detail
            labels = ["Title", "Start Time", "End Time", "Duration (days)", "Description", "Calendar ID", "Task ID"]
            values = [title, time.strftime('%Y-%m-%d %H:%M'), end_date.strftime('%Y-%m-%d %H:%M'), str(duration_days), description, str(calendar_id), str(task_id)]

            # Displaying each label and its corresponding value
            for i in range(len(labels)):
                row_frame = Frame(details_frame, bg='white')
                row_frame.pack(fill='x', pady=2)
                Label(row_frame, text=f"{labels[i]}:", width=15, anchor='w', bg='white', font=("Helvetica", 10, "bold")).pack(side='left')
                Label(row_frame, text=values[i], width=25, anchor='w', bg='white', font=("Helvetica", 10)).pack(side='left')

    # Function to execute search based on user input and display results
    def search():
        # Getting user input for the search
        query = search_entry.get()

        # Preparing and executing the search query
        cursor = db.cursor()
        search_query = """
            SELECT TaskID, Title, Time, Duration, Description, Calendar_CalendarID FROM Task
            WHERE UserID = %s AND
            (Title LIKE %s OR
             Description LIKE %s OR
             Calendar_CalendarID LIKE %s OR
             TaskID LIKE %s OR
             Time LIKE %s OR
```

```
                Time LIKE %s OR
                Duration LIKE %s OR
                ADDDATE(Time, INTERVAL Duration MINUTE) LIKE %s)
            """
            cursor.execute(search_query, (user_id, f'%{query}%', f'%{query}%', f'%{query}%', f'%{query}%', f'%{query}%', f'%{query}%', f'%{query}%'))
            tasks = cursor.fetchall()
            cursor.close()

            # Clearing previous search results from the treeview
            for i in tree.get_children():
                tree.delete(i)

            # Populating the treeview with the search results
            for task in tasks:
                task_id, title, start_time, duration, description, calendar_id = task
                duration_days = duration // 1440  # Convert minutes to days
                end_time = calculate_end_date(start_time, duration)
                tree.insert("", "end", values=(task_id, title, start_time, end_time, duration_days, description, calendar_id))

        # Setting up the main window for task search
        main_window = Tk()
        main_window.title("Search Tasks")
        main_window.geometry("900x600")
        main_window.configure(bg='white')

        # Configuring and placing search bar, entry, and button in the main window
        Label(main_window, text="Search Tasks", bg='white', fg='blue', font=("Helvetica", 16, "bold")).pack(pady=(10, 10))
        search_entry = Entry(main_window, width=50)
        search_entry.pack(pady=5)
        Button(main_window, text="Search", command=search, bg="#ADD8E6", fg="black", font=("Helvetica", 10, "bold"), highlightthickness=4).pack(pady=10)

        # Configuring and adding a treeview for displaying search results
        tree = Treeview(main_window, columns=("TaskID", "Title", "Start Time", "End Time", "Duration (Days)", "Description", "Calendar ID"), show='headings')
        tree.heading("TaskID", text="TaskID")
        tree.heading("Title", text="Title")
        tree.heading("Start Time", text="Start Time")
        tree.heading("End Time", text="End Time")
        tree.heading("Duration (Days)", text="Duration (Days)")
        tree.heading("Description", text="Description")
        tree.heading("Calendar ID", text="Calendar ID")
        tree.column("TaskID", width=50, anchor='center')
        tree.column("Title", width=75, anchor='center')
        tree.column("Start Time", width=150, anchor='center')
        tree.column("End Time", width=150, anchor='center')
        tree.column("Duration (Days)", width=100, anchor='center')
        tree.column("Description", width=200, anchor='center')
        tree.column("Calendar ID", width=80, anchor='center')
        tree.pack(expand=True, fill='both', padx=10, pady=10)
```

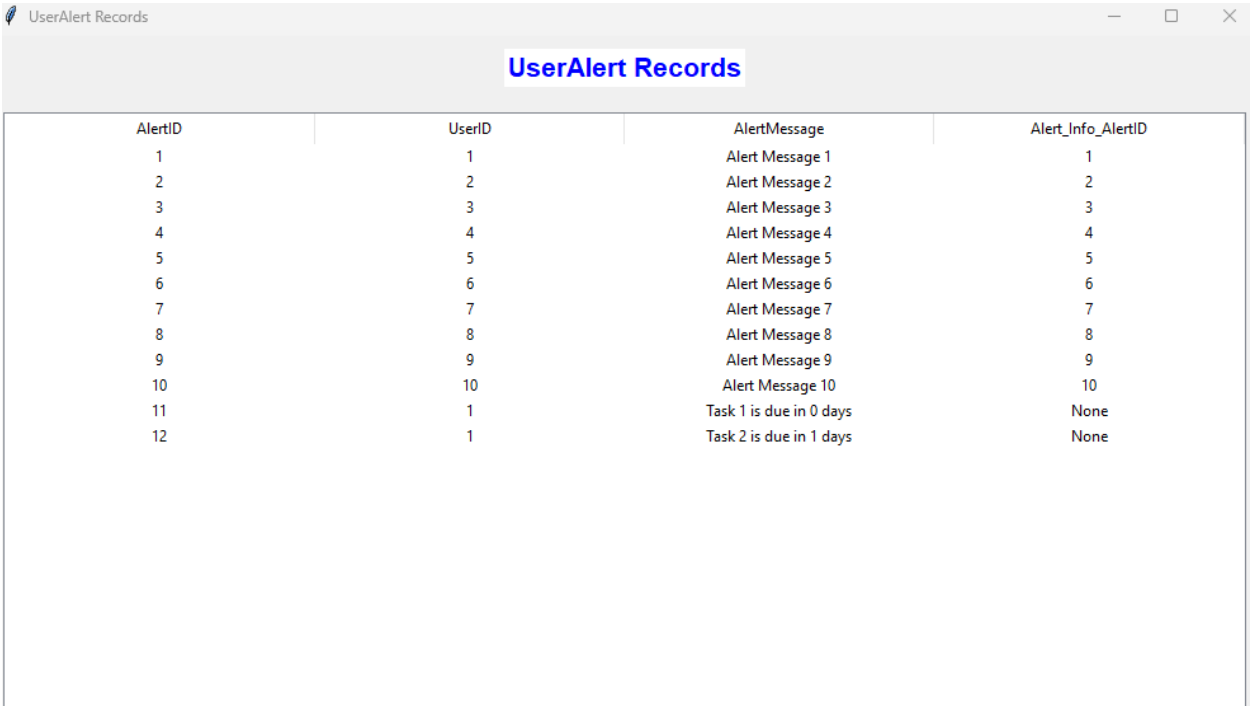
```
    tree.pack(expand=True, fill='both', padx=10, pady=10)

    # Adding a scrollbar for the treeview
    scrollbar = Scrollbar(main_window, command=tree.yview)
    tree.configure(yscrollcommand=scrollbar.set)
    scrollbar.pack(side='right', fill='y')

    # Button to view detailed information of a selected task
    Button(main_window, text="View Task Details", command=lambda: display_task_details(tree.item(tree.selection())['values'][0]), bg="#ADD8E6", fg="black", font=("Helvetica", 10, "bold"), highlightthickness=4).pack(pady=10)

    # Button to return to the main menu
    Button(main_window, text="Return To Main Menu", command=main_window.destroy, bg="#ADD8E6", fg="black", font=("Helvetica", 10, "bold"), highlightthickness=4).pack(side='left', pady=10)

    # Initiating the main window's event loop
    main_window.mainloop()

```

Phase 6.2:

Figure 5: Print All Data

The below code implements a graphical user interface (GUI) using Python's Tkinter library to display data from a MySQL database in a user-friendly manner. The main function, record_view(db), creates a window where users can select different database tables to view their records. Upon running, the record_view function opens a main window titled "Record View," offering a selection of database tables like 'User', 'UserSettings', 'Calendar', and others, each with their specific columns like 'UserID', 'Username', 'Password', etc. For each table, there's a button that, when clicked, opens a new window displaying all records from that table. This is achieved through the nested function show_table_data(table_name, columns), which creates a Toplevel window with a Treeview widget for tabular data display. Each column in the Treeview is configured with headings and specific widths.

The GUI layout is user-friendly, with clear labels and button styles configured for better visual appeal. The buttons allow viewing of table records and also include a "Return To Main Menu" button for easy navigation. The code uses SQL queries to fetch data from the specified tables in the MySQL database and displays this data in an organized and interactive format, making it accessible even to those not familiar with SQL or command-line database operations.




Record View
Which Records would you like to view?
User
UserSettings
Calendar
Task
UserAlert
UserLogin
AlertInfo
Admin
Admin Login
Admin Settings
Return To Main Menu

```python
import mysql.connector
from tkinter import Tk, Label, Button, Toplevel
from tkinter.ttk import Treeview
from tkinter import Scrollbar

def record_view(db):
    # Function to display table data in a new window
    def show_table_data(table_name, columns):
        # Function to fetch and populate data into the treeview
        def populate_data():
            cursor = db.cursor()
            cursor.execute(f"SELECT * FROM {table_name}")  # SQL query to fetch all records from the specified table
            records = cursor.fetchall()  # Fetching all records from the table
            for row in records:
                tree.insert('', 'end', values=row)  # Inserting each record into the treeview
            cursor.close()  # Closing the cursor

        # Creating a new window to display the data of a specific table
        data_window = Toplevel()
        data_window.title(f"{table_name} Records")
        data_window.geometry("1000x600")

        # Displaying a label at the top of the data window
        Label(data_window, text=f"{table_name} Records", bg='white', fg='blue', font=("Helvetica", 16, "bold")).pack(pady=(10, 10))

        # Setting up the treeview for displaying table data
        tree = Treeview(data_window, columns=columns, show='headings')
        for col in columns:
            tree.heading(col, text=col)  # Configuring column headers
            tree.column(col, width=100, anchor='center')  # Setting column width and alignment

        tree.pack(expand=True, fill='both', padx=10, pady=10)  # Placing the treeview in the window

        # Adding a scrollbar to the treeview
        scrollbar = Scrollbar(data_window, command=tree.yview)
        tree.configure(yscrollcommand=scrollbar.set)  # Linking the scrollbar to the treeview
        scrollbar.pack(side='right', fill='y')  # Placing the scrollbar

        populate_data()  # Populating the treeview with data

    # Main window for selecting which table records to view
    record_window = Tk()
    record_window.title("Record View")
    record_window.geometry("600x600")
    record_window.configure(bg='white')
```

```python
    record_window.configure(bg='white')

    # Main label for the record window
    Label(record_window, text="Which Records would you like to view?", bg='white', fg='blue', font=("Helvetica", 16, "bold")).pack(pady=(10, 10))

    # Button style configuration
    button_style = {'fg': "black", 'bg': "#ADD8E6", 'font': ("Helvetica", 10, "bold"), 'highlightthickness': 4, 'width': 20, 'height': 1}

    # Dictionary containing table names and their respective columns
    tables = {
        'User': ['UserID', 'Username', 'Password', 'Name', 'Address', 'PhoneNumber', 'Birthday'],
        'UserSettings': ['SettingsID', 'UserID', 'NotificationSettings'],
        'Calendar': ['CalendarID', 'UserID'],
        'Task': ['TaskID', 'UserID', 'Title', 'Time', 'Duration', 'Description', 'Calendar_CalendarID'],
        'UserAlert': ['AlertID', 'UserID', 'AlertMessage', 'Alert_Info_AlertID'],
        'UserLogin': ['LoginID', 'UserID', 'LoginTime', 'LogoutTime'],
        'AlertInfo': ['InfoID', 'AlertID', 'UserID', 'AlertHistory'],
        'Admin': ['AdminID', 'Username', 'Password', 'Name', 'Certification', 'Address', 'PhoneNumber', 'Birthday'],
        'Admin Login': ['LoginID', 'AdminID', 'LoginTime', 'LogoutTime'],
        'Admin Settings': ['SettingsID', 'AdminID', 'NotificationSettings']
    }

    for i, (table_name, columns) in enumerate(tables.items()):
        Button(record_window, text=table_name, **button_style, command=lambda name=table_name, cols=columns: show_table_data(name, cols)).pack(pady=5)

    Button(record_window, text="Return To Main Menu", **button_style, command=record_window.destroy).pack(side='left', padx=10, pady=10)

    record_window.mainloop()
```

Phase 7: Conclusion:

Through this project, I have gained comprehensive insights into database management, particularly in designing robust and efficient systems. My journey began with developing external and conceptual models, which enlightened me on the intricate interplay between various database features and the essential components required for a functional database. The subsequent creation of Entity-Relationship (ER) and Enhanced Entity-Relationship (EER) diagrams further deepened my understanding of database architecture, particularly in handling multi-variable attributes and many-to-many relationships. This phase also enhanced my knowledge of cardinality. The practical aspect of this learning came with the actual construction of database tables. This stage was pivotal in understanding the complexities involved in table creation, including considerations like primary and foreign keys, and the principles of normalization. Importing and optimizing data, along with normalization checks, offered me valuable experience in data management. The development of the Graphical User Interface (GUI) marked the most difficult part of this learning process. This phase required synthesizing all the theoretical knowledge into a user-friendly interface, a challenging yet rewarding endeavor. It demanded significant time and effort, more than any other assignment I've undertaken, and thus imparted a profound appreciation for the intricacies of database development.

To enhance this project further, several additional features could be integrated. Firstly, a customizable calendar within the GUI would significantly enhance user experience as it would allow the users to make the calendar different colors and themes to provide better visual stimulation. However, limitations with the Tkinter framework precluded its implementation in the current version. Secondly, incorporating a feature to track and display a history of completed tasks would provide users with valuable insights into their productivity patterns. Another beneficial enhancement would be the introduction of customizable alert settings, allowing users to determine the frequency and lead time of task-due notifications. Lastly, integrating email notifications for impending tasks could significantly improve user engagement and task management efficiency, providing reminders even when users are not actively using the Task Management System (TMS). These enhancements, I believe, would substantially elevate the application's functionality and user experience.

References:

Atlassian. “Jira: Issue & Project Tracking Software.” Atlassian, www.atlassian.com/software/jira. Accessed 30 Sep. 2023.

Dancuk, Milica. “How to Create a Table in Mysql {and Display Data}.” Knowledge Base by PhoenixNAP, 3 Aug. 2022, phoenixnap.com/kb/how-to-create-a-table-in-mysql.

Dylvannus, Cassandra I. “MySQL Python Connector.” Python Mysql, www.w3schools.com/python/python_mysql_getstarted.asp. Accessed 27 Nov. 2023.

Fitzpatrick, Martin. “Create Python GUI with Tkinter.” Python GUIs, Python GUIs, 13 Sept. 2023, www.pythonguis.com/tutorials/create-gui-tkinter/.

“Normal Forms in DBMS.” GeeksforGeeks, GeeksforGeeks, 6 Nov. 2023, www.geeksforgeeks.org/normal-forms-in-dbms/.

Mara, Evelynn. “Python Gui - Tkinter.” GeeksforGeeks, GeeksforGeeks, 9 Nov. 2023, www.geeksforgeeks.org/python-gui-tkinter/.

Redoran, Mira H. “Getting Started with Mysql Python Connector.” MySQL Tutorial, 25 Sept. 2023, www.mysqltutorial.org/getting-started-mysql-python-connector/.

Sirus, Amara d. “Python and Mysql Database: A Practical Introduction.” Real Python, Real Python, 12 Mar. 2023, realpython.com/python-mysql/.

“A To-Do List to Organize Your Work & Life.” Todoist, todoist.com/. Accessed 30 Sep. 2023.

Torrez, Edward I. “Database Mangement Table Creation.” MySQL Create Table Statement, www.w3schools.com/mysql/mysql_create_table.asp. Accessed 27 Nov. 2023.

“Trello Brings All Your Tasks, Teammates, and Tools Together.” Trello, trello.com/home. Accessed 30 Sep. 2023.

Umbra, Skarlett. “Modern GUI Using Tkinter.” Medium, Medium, 4 Sept. 2022, medium.com/@fareedkhandev/modern-gui-using-tkinter-12da0b983e22.

Void, James E. “Python GUI Programming with Tkinter.” Real Python, Real Python, 30 Jan. 2023, realpython.com/python-gui-tkinter/.

“DBMS SQL View - Javatpoint.” Www.Javatpoint.Com, www.javatpoint.com/dbms-sql-view. Accessed 17 Nov. 2023.

“Database Views.” IBM, www.ibm.com/docs/en/control-desk/7.6.1.2?topic=structure-views. Accessed 07 Nov. 2023.