Inventory Management System

Contribution

This project was collaboratively developed by:

- **Ekta Shukla (33.4%)**
- **Rithika Kavitha Suresh (33.3%)**
- **Praveenraj Seenivasan (33.3%)**

Project Overview

The **Inventory Management System** is a Python-based desktop application designed to handle and streamline inventory processes for various business entities, such as products, suppliers, orders, customers, sales, warehouses, and inventory stock. The system integrates a graphical user interface (GUI) built using Tkinter, and it connects to a MySQL database backend for efficient data storage and retrieval. The project supports fundamental database operations (CRUD) and includes advanced SQL queries to provide valuable insights through data analysis.

This README outlines the features, setup instructions, database requirements, and usage guidelines for the 5th deliverable of the project.

Key Features

- **User-Friendly GUI:** A Tkinter-based interface that simplifies user interaction with the database.
- **CRUD Operations:** Allows users to create, read, update, and delete records for various entities.
- **Advanced SQL Queries:** Supports complex SQL queries including set operations, aggregate functions, subqueries using the WITH clause, and OLAP (Online Analytical Processing) queries.
- **Data Integrity and Validation:** Ensures data consistency through referential integrity checks and input validation.
- **Visual Feedback:** Provides informative message boxes for operation success, errors, and data alerts.
- **Error Handling:** Handles common errors gracefully with user-friendly messages.

Prerequisites

Ensure the following are installed before running the application:

- Python 3.x
- MySQL Server
- Required Python libraries:

bash
Copy code
pip install mysql-connector-python
pip install pillow

Database Setup

1. **Database Configuration:**
    - Start the MySQL server and create a database named inventory_management:
    - Set up tables for entities such as Product, Supplier, Order, Order_Details, Customer, Sale, Sale_Details, Warehouse, and Inventory_Stock based on the provided Entity Relationship Diagram (ERD).
2. **MySQL Connection Parameters:**
    - Update the host, user, password, and database parameters in the Python script to match your MySQL configuration.

Application Setup

1. **Clone the Project:** Download or clone the project repository to your local machine.
2. **Install Python Dependencies:** Use the following command to install the required Python libraries:

bash
Copy code
pip install mysql-connector-python pillow

3. **Run the Application:** Execute the Python script to start the application:

bash
Copy code
python inventory_management.py

Interface Components

The GUI for the application consists of the following components:

1. Login Window:

- Prompts the user to enter their MySQL username and password for database connection.
- Provides success or error messages based on the connection status.

2. Main Window:

- **Title:** Displays "Inventory Management System".
- **Sections:**
    - **CRUD Operations Section:**
        - Buttons for Create, Read, Update, and Delete operations.
        - Each button triggers respective functions for interacting with the database.

CRUD Operations in the Inventory Management System

In this section, we'll describe the **CRUD operations** implemented in the project using Python, MySQL, and Tkinter. The CRUD operations (Create, Read, Update, and Delete) allow users to interact with the database effectively. Below is an overview of each operation, followed by how it is implemented in the system:

1. Create Operation

The **Create** operation allows users to insert new records into the specified table of the inventory_management database.

- **Implementation Details:**
    - The create_record function prompts the user to input the table name and its values for each column using dialog boxes.
    - It automatically fetches the table structure using the DESCRIBE query, asks for each column value, and inserts the new record using a dynamically constructed INSERT INTO SQL query.
    - The record is added to the database, and a confirmation message is displayed upon success.

```
def create_record():
    try:
        table = simpledialog.askstring("Input", "Enter the table name:")
        if not table:
            return
        cursor = db_connection.cursor()
        cursor.execute(f"DESCRIBE {table}")
        columns = cursor.fetchall()
        values = []
        for column in columns:
            column_name = column[0]
            value = simpledialog.askstring("Input", f"Enter value for {column_name}:")
            values.append(value)
        placeholders = ", ".join(["%s"] * len(values))
        columns_list = ", ".join([col[0] for col in columns])
        query = f"INSERT INTO {table} ({columns_list}) VALUES ({placeholders})"
        cursor.execute(query, values)
        db_connection.commit()
        messagebox.showinfo("Success", "Record created successfully!")
    except Error as e:
        messagebox.showerror("Error", f"Failed to create record: {e}")
```

2. Read Operation

The **Read** operation retrieves and displays records from the specified table.

- **Implementation Details:**
    - The read_records function asks the user for the table name and fetches all records using the SELECT * query.
    - The results are displayed in a separate window using a Treeview widget for better readability.
    - This function is useful for quickly viewing data in any table.

```
def read_records():
    try:
        table = simpledialog.askstring("Input", "Enter the table name to read from:")
        if not table:
            return
        cursor = db_connection.cursor()
        cursor.execute(f"SELECT * FROM {table}")
        records = cursor.fetchall()
        display_results(records, f"Records from {table}")
    except Error as e:
        messagebox.showerror("Error", f"Failed to read records: {e}")
```

3. Update Operation

The **Update** operation allows users to modify existing records in a table.

- **Implementation Details:**
    - The update_record function prompts the user to input the table name, the primary key value of the record to update, and the column to be updated.
    - It fetches the primary key of the table automatically and ensures that the specified column exists before updating.
    - The function checks for referential integrity, preventing updates that would violate foreign key constraints.
    - The SQL UPDATE query is constructed dynamically, and the record is updated in the database.

```
        # Get the column to update
        cursor.execute(f"DESCRIBE {table}")
        columns = [row[0] for row in cursor.fetchall()]
        column_list = "\n".join(columns)
        column_to_update = simpledialog.askstring("Input",
            f"Enter column name to update:\nAvailable columns:\n{column_list}")

        if not column_to_update or column_to_update not in columns:
            messagebox.showerror("Error", "Invalid column name")
            return

        # Get the new value
        new_value = simpledialog.askstring("Input", f"Enter new value for {column_to_update}:")
        if new_value is None:  # User clicked Cancel
            return

        # Construct and execute the update query
        query = f"UPDATE {table} SET {column_to_update} = %s WHERE {primary_key} = %s"
        cursor.execute(query, (new_value, id_value))
        db_connection.commit()

        if cursor.rowcount > 0:
            messagebox.showinfo("Success", "Record updated successfully!")
        else:
            messagebox.showinfo("Info", "No matching record found to update")

    except Error as e:
        if "foreign key constraint fails" in str(e):
            messagebox.showerror("Error",
                "Cannot update this record because it would violate referential integrity.\n"
                "The new value must exist in the referenced table.")
        else:
            messagebox.showerror("Error", f"Failed to update record: {e}")
        print(f"Error details: {e}")  # For debugging
```

4. Delete Operation

The **Delete** operation removes a record from the specified table based on the primary key.

- **Implementation Details:**
  - The delete_record function prompts the user to enter the table name and the primary key value of the record to delete.
  - It retrieves the primary key field dynamically and warns the user about potential foreign key constraints.
  - If a record is found, it is deleted using the DELETE FROM query. The function also handles errors related to foreign key violations.

```
def delete_record():
    try:
        # First ask for table name
        table = simpledialog.askstring("Input", "Enter the table name to delete from:")
        if not table:
            return

        # Handle reserved words in MySQL
        if table.lower() == "order":
            table = "`order`"  # Escape the reserved word

        # Get the primary key field for the selected table
        cursor = db_connection.cursor()
        cursor.execute(f"SHOW KEYS FROM {table} WHERE Key_name = 'PRIMARY'")
        primary_key = cursor.fetchone()[4]  # Get the primary key column name

        # First show existing records
        cursor.execute(f"SELECT * FROM {table}")
        records = cursor.fetchall()

        if not records:
            messagebox.showinfo("Info", "No records found in this table")
            return

        # Show records in a message box
        record_list = "\n".join([str(record) for record in records])
        messagebox.showinfo("Existing Records", f"Available records:\n{record_list}")

        # Ask for the ID value
        id_value = simpledialog.askstring("Input", f"Enter {primary_key} value to delete:")
        if not id_value:
            return
```

- **Advanced SQL Queries Section:**
  - Buttons for executing predefined complex queries:
    - **Top 5 Best-Selling Products**
    - **Monthly Sales Trend**
    - **Product Reorder Alert**
    - **Customer Ranking**
    - **Category Sales Analysis**
- **Exit Button:** Allows users to safely exit the application.

3. Results Display:

- Uses a Tkinter Treeview widget to display query results in a table format with scroll support.
- Includes columns with headers, formatted data, and a close button for easy navigation.

Advanced SQL Query Examples

The application supports a range of advanced SQL queries, including:

1. **Set Operations:**
   - Displays products that need urgent reordering or are top-selling:
2. **Aggregate Functions:**
   - Calculates the average monthly sales amount:
3. **Subquery with WITH Clause:**
   - Identifies top customers using a common table expression (CTE):

```
sql
WITH TotalPurchases AS (
  SELECT CID, SUM(Stotal_amount) AS TotalSpent
  FROM Sale
  GROUP BY CID
)
SELECT CID, TotalSpent
FROM TotalPurchases
WHERE TotalSpent > 1000;
```

4. **OLAP Analysis:**
    - Cumulative sales trend analysis using window functions:

sql

SELECT sale_date, Stotal_amount, SUM(Stotal_amount) OVER (ORDER BY sale_date) AS CumulativeSales

FROM Sale;

Error Handling

The application includes robust error handling mechanisms:

- **Connection Errors:** Provides clear feedback if the MySQL connection fails.
- **Invalid Input:** Alerts users to input errors or invalid column/table names.
- **Referential Integrity Violations:** Prevents update or delete actions that would violate foreign key constraints, with informative error messages.

Testing and Use Cases

The application has been tested with various use cases, including:

- **Managing Inventory:** Adding new products, updating stock levels, and deleting obsolete items.
- **Customer Analytics:** Ranking customers based on their total purchases.
- **Sales Trend Analysis:** Visualizing monthly sales trends and identifying top-selling products.
- **Stock Alerts:** Providing alerts for low-stock items requiring reorder.