

Advanced Database Systems (QH0541)

BSc (Hons) Computer Science

Academic Year 2024-2025

Name: Andre Ramos

Tutor: Ajmal Gharib

Student ID: 10300471

Word count: 2000

Date: 04/02/2025

Contents

1. Introduction.....	2
2. Part 1	3
2.1 Business Case:.....	3
2.2 Seven Business Requirements:	3
2.3 Entity Relationship Diagram	4
2.4 ERD Justification	4
2.5 Referential Integrity Constraints	5
2.6 Table creation and data insertion	6
2.7 Testing and Validation:	12
2.8 Explanation of the Tables and Relationships:	14
3. Part 2	16
3.1 Triggers.....	16
3.2 Views.....	18
4. Part 3	22
Conclusion.....	38
References.....	39

1. Introduction

Warehousing operations rely heavily on efficient data management, where structured databases play a crucial role in tracking inventory, processing orders, and maintaining operational accuracy. This project was inspired by my experience working with warehouse datasets, which provided insight into the challenges of managing large volumes of data efficiently. Recognizing the need for a scalable and well-structured system, I developed a database solution tailored for Prime Warehousing Solutions to support efficient inventory tracking and future expansion.

The project focuses on designing and implementing a relational database using SQLite, ensuring data integrity, minimal redundancy, and optimized performance through Third Normal Form normalization. A comprehensive Entity-Relationship Diagram guided the development of a five-table relational schema, with key features such as triggers, views, and stored functions to automate essential warehouse operations. Additionally, XML handling and BLOB storage were incorporated to enhance data management capabilities.

By integrating multi-threading for parallel order processing and ensuring security through SQL Injection prevention and password hashing, this system provides a robust and scalable solution for warehouse data management. The following report outlines the design, implementation, and optimization of this database, highlighting how it improves efficiency, security, and scalability in warehouse operations.

2. Part 1

2.1 Business Case:

The database system is designed for a warehouse distribution centre. This centre handles storing, managing, and distributing products such as shoes, clothing, and accessories. The goal is to create a system that keeps track of stock levels, suppliers, employees, and orders. The system will help the warehouse run smoothly by solving common issues like stock going missing, delayed deliveries, and poor organisation of items in storage.

This system will focus on improving how stock is managed and tracked. It will also keep records of suppliers, employees, and orders to make operations more efficient. By doing this, the system will reduce problems like running out of stock, delays in fulfilling customer orders, and locating misplaced items in the warehouse.

2.2 Seven Business Requirements:

- **Stock Management:** The database should track how much stock is available for each product. It should give alerts when stock levels are low so the warehouse can reorder in time.
- **Order Tracking:** The system must record all incoming orders (stock deliveries from suppliers) and outgoing orders (shipments to customers). It should also track the status of each order, such as "pending," "delivered," or "delayed."
- **Supplier Details:** The database should store information about suppliers, including their names, contact details, and the types of products they provide. This will make it easier to contact suppliers and review their reliability.
- **Warehouse Storage:** Products should be assigned specific storage locations within the warehouse, like an aisle and shelf number. This will make it quicker to find items and easier to check if anything goes missing.
- **Employee Records:** The database should keep track of employees, including their names, roles, shifts, and tasks. This will help with assigning jobs like picking orders or carrying out stock checks.
- **Stock Auditing:** The system must identify when stock levels do not match what is expected, such as missing or misplaced items. This will help the warehouse fix issues before they affect customer orders.
- **Delivery Performance:** The database should record delivery times and flag any delays. This will help identify patterns and make improvements to keep customers satisfied.

2.3 Entity Relationship Diagram

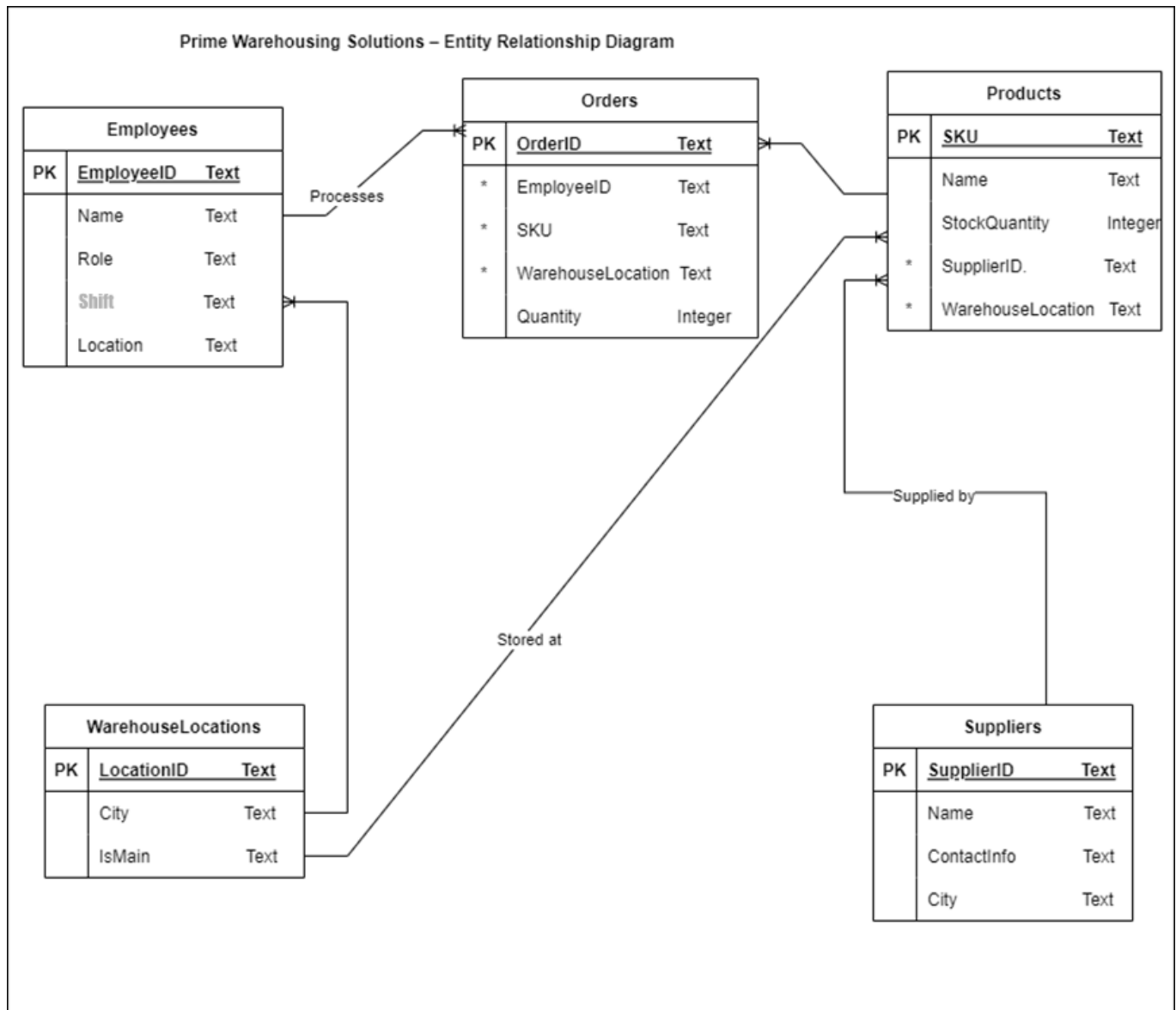


FIGURE 1 PRIME WAREHOUSING SOLUTIONS – ENTITY RELATIONSHIP DIAGRAM

2.4 ERD Justification

The Entity-Relationship Diagram (ERD) represents the database system for Prime Warehousing Solutions, a UK-based warehouse distribution centre. The system has been designed to meet the requirements of Third Normal Form (3NF), ensuring it is clear, efficient, and avoids unnecessary duplication of data. Each table focuses on a key area of the warehouse operations, and the relationships between them are structured to support the company's processes.

2.5 Referential Integrity Constraints

Referential integrity is enforced using foreign keys to link tables and maintain data consistency.

Orders Table:

EmployeeID → references Employees(EmployeeID) to ensure valid employees process orders.

SKU → references Products(SKU) to ensure valid products are ordered.

WarehouseLocation → references WarehouseLocations(LocationID) for valid locations.

Products Table:

SupplierID → references Suppliers(SupplierID) to ensure valid supplier data.

WarehouseLocation → references WarehouseLocations(LocationID) to ensure proper storage locations.

Why?

Foreign keys prevent invalid or orphaned data and ensure consistency between linked tables.

EmployeeID	Name	Role	Shift	Location
E001	Alice Johnson	HR Manager	Morning	Manchester Warehouse
E002	Bob Smith	Picker	Morning	Manchester Warehouse
E003	Charlie Brown	HR Assistant	Evening	Birmingham Warehouse
E004	Diana Carter	Picker	Morning	London Warehouse
E005	Edward Davis	HR Assistant	Evening	Glasgow Warehouse

FIGURE 2 EMPLOYEES TABLE

LocationID	City	IsMain
WH-1	Manchester	Yes
WH-2	Birmingham	No
WH-3	London	No
WH-4	Glasgow	No
WH-5	Liverpool	No

FIGURE 3 WAREHOUSELOCATIONS TABLE

SKU	Name	SupplierID	WarehouseLocation	StockQuantity
P001	Running Shoes	S002	WH-2	395
P002	Hooded Sweatshirt	S003	WH-3	390
P003	Waterproof Jacket	S004	WH-4	401
P004	Yoga Leggings	S005	WH-5	392
P005	Casual Trainers	S001	WH-1	398

<
>
Employees
WarehouseLocations
Products
Suppliers
Orders

FIGURE 4 PRODUCTS TABLE

SupplierID	Name	ContactInfo	City
S001	Global Footwear Ltd	info@globalfootwear.com	Manchester
S002	Urban Wear Supplies	sales@urbanwearsupplies	Birmingham
S003	Active Sports Gear Co.	support@activesportsgear	London
S004	Peak Outdoor Equipment	contact@peakoutdoor.com	Glasgow
S005	Performance Apparel Ltd	help@performanceapparel	Liverpool

<
>
Employees
WarehouseLocations
Products
Suppliers

FIGURE 5 SUPPLIERS TABLE

OrderID	EmployeeID	SKU	WarehouseLocation	Quantity
O001	E002	P002	WH-2	62
O002	E010	P003	WH-3	57
O003	E006	P004	WH-4	68
O004	E004	P005	WH-5	59
O005	E004	P006	WH-1	65

<
>
Employees
WarehouseLocations
Products
Suppliers
Orders

FIGURE 6 ORDERS TABLE

Flat File with Collected DataThe flat file in Excel organized data into separate sheets for Employees, Products, Suppliers, Warehouse Locations, and Orders. This structure ensured data normalisation and made it easier to migrate the data into SQLite tables.

2.6 Table creation and data insertion.

```
2 CREATE TABLE Suppliers (
3     SupplierID TEXT PRIMARY KEY,
4     Name TEXT NOT NULL,
5     ContactInfo TEXT NOT NULL,
6     City TEXT NOT NULL
7 );
```

FIGURE 7 CREATE QUERY SUPPLIERS

```
2 CREATE TABLE WarehouseLocations (  
3     LocationID TEXT PRIMARY KEY,  
4     City TEXT NOT NULL,  
5     IsMain TEXT NOT NULL  
6 );
```

FIGURE 8 CREATE QUERY WAREHOUSELOCATIONS

```
1 -- Create the Employees table  
2 CREATE TABLE Employees (  
3     EmployeeID TEXT PRIMARY KEY,  
4     Name TEXT NOT NULL,  
5     Role TEXT NOT NULL,  
6     Shift TEXT NOT NULL,  
7     Location TEXT NOT NULL  
8 );
```

FIGURE 9 CREATE QUERY EMPLOYEES

```
1 -- Create Orders Table  
2 CREATE TABLE Orders (  
3     OrderID TEXT PRIMARY KEY,  
4     EmployeeID TEXT,  
5     SKU TEXT,  
6     WarehouseLocation TEXT,  
7     Quantity INTEGER,  
8     FOREIGN KEY (EmployeeID) REFERENCES Employees(EmployeeID)  
9 );
```

FIGURE 10 CREATE QUERY ORDERS

```
1 CREATE TABLE Products (  
2     SKU TEXT PRIMARY KEY,  
3     Name TEXT NOT NULL,  
4     SupplierID TEXT NOT NULL,  
5     WarehouseLocation TEXT NOT NULL,  
6     StockQuantity INTEGER NOT NULL,  
7     FOREIGN KEY (SupplierID) REFERENCES Suppliers(SupplierID),  
8     FOREIGN KEY (WarehouseLocation) REFERENCES WarehouseLocations(LocationID)  
9 );
```

FIGURE 11 CREATE QUERY PRODUCTS

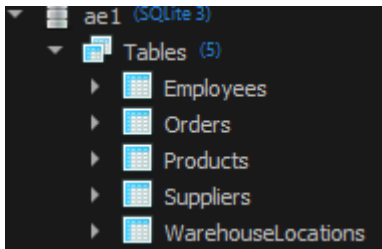


FIGURE 12 TABLES CREATED

```
1 -- Insert sample data into Employees
2 INSERT INTO Employees (EmployeeID, Name, Role, Shift, Location) VALUES
3 ('E001', 'Alice Johnson', 'Picker', 'Morning', 'Manchester'),
4 ('E002', 'Bob Smith', 'HR Assistant', 'Morning', 'Manchester'),
5 ('E003', 'Charlie Brown', 'Picker', 'Evening', 'Birmingham'),
6 ('E004', 'Diana Carter', 'HR Assistant', 'Morning', 'London'),
7 ('E005', 'Edward Davis', 'HR Assistant', 'Evening', 'Glasgow');
```

FIGURE 13 INSERT QUERY EMPLOYEES

```
1 -- Insert sample data into Products
2 INSERT INTO Products (SKU, Name, StockQuantity, SupplierID, WarehouseLocation) VALUES
3 ('P001', 'Running Shoes', 395, 'S001', 'WH-1'),
4 ('P002', 'Hooded Sweatshirt', 390, 'S002', 'WH-2'),
5 ('P003', 'Waterproof Jacket', 401, 'S003', 'WH-3'),
6 ('P004', 'Yoga Leggings', 392, 'S004', 'WH-4'),
7 ('P005', 'Casual Trainers', 398, 'S005', 'WH-5');
```

FIGURE 14 INSERT QUERY PRODUCTS

```
1 -- Insert sample data into Suppliers
2 INSERT INTO Suppliers (SupplierID, Name, ContactInfo, City) VALUES
3 ('S001', 'Global Footwear Ltd', 'info@globalfootwear.com', 'Manchester'),
4 ('S002', 'Urban Wear Supplies', 'sales@urbanwearsupplies.com', 'Birmingham'),
5 ('S003', 'Active Sports Gear Co.', 'support@activesportsgear.com', 'London'),
6 ('S004', 'Peak Outdoor Equipment', 'contact@peakoutdoor.com', 'Glasgow'),
7 ('S005', 'Performance Apparel Ltd', 'help@performanceapparel.com', 'Liverpool');
```

FIGURE 15 INSERT QUERY SUPPLIERS

```
1 -- Insert sample data into WarehouseLocations
2 INSERT INTO WarehouseLocations (LocationID, City, IsMain) VALUES
3 ('WH-1', 'Manchester', 'Yes'),
4 ('WH-2', 'Birmingham', 'No'),
5 ('WH-3', 'London', 'No'),
6 ('WH-4', 'Glasgow', 'No'),
7 ('WH-5', 'Liverpool', 'No');
```

FIGURE 16 INSERT QUERY WAREHOUSELOCATIONS

```
1 -- Insert sample data into Orders
2 INSERT INTO Orders (OrderID, EmployeeID, SKU, WarehouseLocation, Quantity) VALUES
3 ('0001', 'E001', 'P002', 'WH-2', 62),
4 ('0002', 'E002', 'P003', 'WH-3', 57),
5 ('0003', 'E003', 'P004', 'WH-4', 68),
6 ('0004', 'E004', 'P005', 'WH-5', 59),
7 ('0005', 'E005', 'P001', 'WH-1', 65);
```

FIGURE 17 INSERT QUERY ORDERS

```
1 INSERT INTO Products (SKU, Name, StockQuantity, SupplierID, WarehouseLocation)
2 VALUES ('P100', 'Invalid Product', 50, 'INVALID_SUPPLIER', 'WH-1');
```

Grid view Form view

Total rows loaded: 0

Status

[02:37:18] Error while executing SQL query on database 'ae1': FOREIGN KEY constraint failed

FIGURE 18 RELATIONSHIPS TESTING

```
1 SELECT
2     Orders.OrderID,
3     Orders.EmployeeID,
4     Employees.Name AS EmployeeName,
5     Employees.Role AS EmployeeRole,
6     Orders.SKU,
7     Products.Name AS ProductName,
8     Orders.WarehouseLocation,
9     Orders.Quantity
10 FROM Orders
11 JOIN Employees ON Orders.EmployeeID = Employees.EmployeeID
12 JOIN Products ON Orders.SKU = Products.SKU;
```

Grid view Form view

Total rows loaded: 5

	OrderID	EmployeeID	EmployeeName	EmployeeRole	SKU	ProductName	Warehouse	Quantity
1	O001	E001	Alice Johnson	Picker	P002	Hooded Sweatshirt	WH-2	62
2	O002	E002	Bob Smith	HR Assistant	P003	Waterproof Jacket	WH-3	57
3	O003	E003	Charlie Brown	Picker	P004	Yoga Leggings	WH-4	68
4	O004	E004	Diana Carter	HR Assistant	P005	Casual Trainers	WH-5	59
5	O005	E005	Edward Davis	HR Assistant	P001	Running Shoes	WH-1	65

FIGURE 19 FURTHER TESTING

The query in Figure 14 joins the Orders table with the Employees and Products tables using their foreign key relationships:

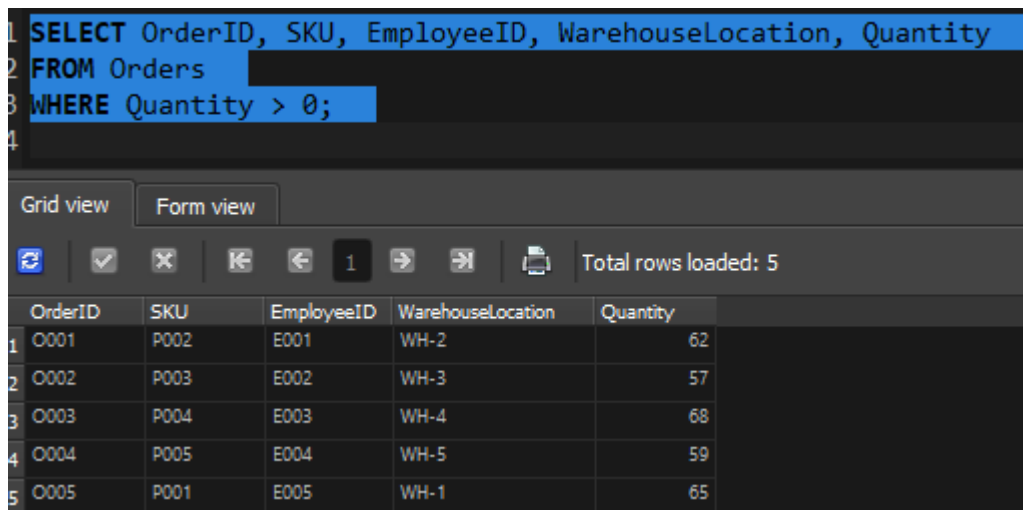
Orders.EmployeeID connects to Employees.EmployeeID to retrieve the employee's name and role.

Orders.SKU connects to Products.SKU to retrieve the product name.

By doing this, the query confirms that the foreign keys correctly link the tables.

FIGURE 20 QUERY BUSINESS RELATED TESTING

The query on figure 20 shows the total number of orders processed by each employee by joining the Employees and Orders tables using the EmployeeID. It uses COUNT to calculate orders and GROUP BY to group results by employee names. This confirms the relationship between the tables and helps track employee performance.



The screenshot shows a database query interface. At the top, a SQL query is entered in a text area:

```
1 SELECT OrderID, SKU, EmployeeID, WarehouseLocation, Quantity
2 FROM Orders
3 WHERE Quantity > 0;
4
```

Below the query area, there are tabs for "Grid view" and "Form view". The "Grid view" tab is selected. Below the tabs is a toolbar with various icons (refresh, check, close, back, forward, etc.) and a status bar that says "Total rows loaded: 5".

The results are displayed in a table with the following columns: OrderID, SKU, EmployeeID, WarehouseLocation, and Quantity. The table contains 5 rows of data:

	OrderID	SKU	EmployeeID	WarehouseLocation	Quantity
1	O001	P002	E001	WH-2	62
2	O002	P003	E002	WH-3	57
3	O003	P004	E003	WH-4	68
4	O004	P005	E004	WH-5	59
5	O005	P001	E005	WH-1	65

FIGURE 21 QUERY TO MONITORS STOCK QUANTITIES

The query on Figure 21 `SELECT OrderID, SKU, EmployeeID, WarehouseLocation, Quantity FROM Orders WHERE Quantity > 0;` demonstrates that stock levels are being tracked correctly.

It retrieves orders with valid quantities greater than zero, ensuring that only active and relevant stock entries are displayed.

Each order is linked to a product (SKU), the employee processing it (EmployeeID), and the warehouse location (WarehouseLocation).

This output confirms that the database successfully monitors stock quantities and their respective locations, aligning with the requirement for stock management and preventing missing or misplaced items.

2.7 Testing and Validation:

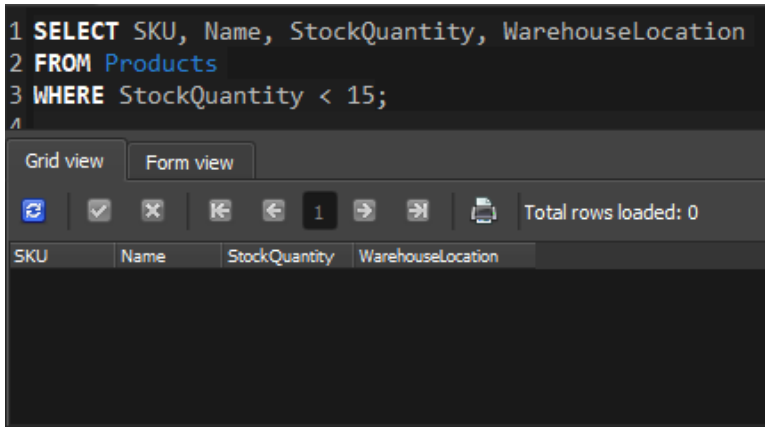


FIGURE 22 LOW STOCK QUERY

The query on Figure 22 demonstrates the successful execution of the query, showing products with stock levels below the specified threshold.

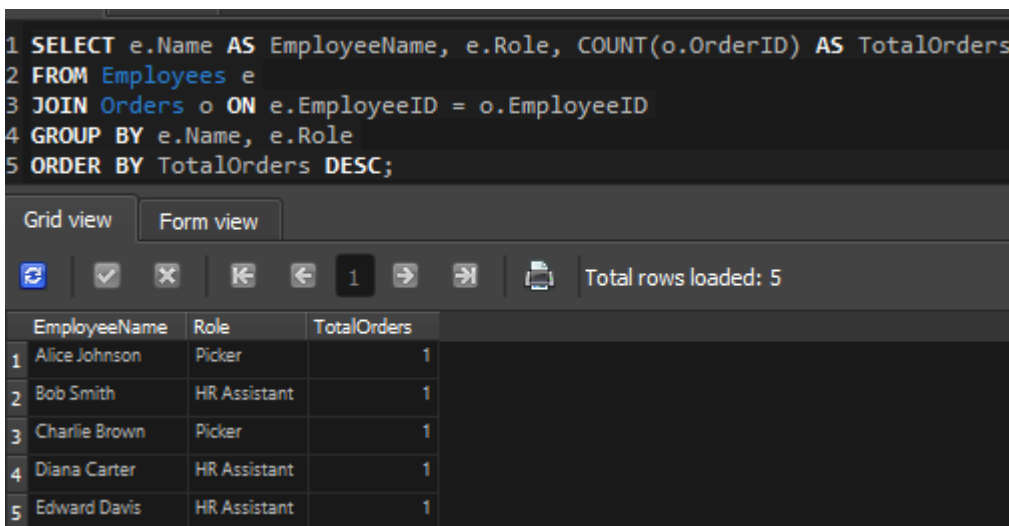


FIGURE 23 ORDERS PROCESSED BY QUERY

The query on Figure 23 demonstrates the successful execution of the query, showing which employees (pickers, packers, or supervisors) are actively managing orders.

```

1 SELECT p.SKU, p.Name, p.StockQuantity, w.City AS Warehouse
2 FROM Products p
3 JOIN WarehouseLocations w ON p.WarehouseLocation = w.LocationID;
4

```

Grid view Form view

Total rows loaded: 5

	SKU	Name	StockQuantity	Warehouse
1	P001	Running Shoes	395	Manchester
2	P002	Hooded Sweatshirt	390	Birmingham
3	P003	Waterproof Jacket	401	London
4	P004	Yoga Leggings	392	Glasgow
5	P005	Casual Trainers	398	Liverpool

FIGURE 24 PRODUCTS STORED BY WAREHOUSELOCATION QUERY

The query on Figure 24 demonstrates the successful execution of the query, this validates that each product is properly linked to its warehouse location, ensuring accurate tracking and storage.

```

1 SELECT p.SKU, p.Name AS ProductName, s.Name AS SupplierName, s.ContactInfo
2 FROM Products p
3 JOIN Suppliers s ON p.SupplierID = s.SupplierID;
4

```

Grid view Form view

Total rows loaded: 5

	SKU	ProductName	SupplierName	ContactInfo
1	P001	Running Shoes	Global Footwear Ltd	info@globalfootwear.com
2	P002	Hooded Sweatshirt	Urban Wear Supplies	sales@urbanwearsupplies.com
3	P003	Waterproof Jacket	Active Sports Gear Co.	support@activesportsgear.com
4	P004	Yoga Leggings	Peak Outdoor Equipment	contact@peakoutdoor.com
5	P005	Casual Trainers	Performance Apparel Ltd	help@performanceapparel.com

FIGURE 25 PRODUCT DETAILS QUERY

The query on Figure 25 demonstrates the successful execution of the query, the database accurately tracks suppliers and their products, facilitating efficient restocking.

2.8 Explanation of the Tables and Relationships:

Normalisation Justification

The database follows Third Normal Form, ensuring efficient data organisation, eliminating redundancy, and maintaining integrity. The normalisation process can be broken down as follows:

First Normal Form – Ensuring Atomicity

- Each table has a primary key that uniquely identifies records.
- All attributes contain atomic values (no multiple values in a single field).
- Example: The Orders table ensures that each row represents one order per product. If an order includes multiple products, each product is recorded as a separate row instead of storing multiple product names in a single field.

Second Normal Form – Eliminating Partial Dependencies

- The database eliminates partial dependencies, meaning that all non-key attributes depend entirely on the primary key.
- Example: Initially, the Orders table may have stored product details such as ProductName and SupplierName. However, since SupplierName depends on SupplierID (not OrderID), these details were moved to the Products and Suppliers tables.
- The Orders table now only stores OrderID, EmployeeID, SKU, and Quantity, while ProductName and SupplierName are retrieved through foreign key relationships.

Third Normal Form – Eliminating Transitive Dependencies

- Transitive dependencies are removed, meaning that non-key attributes do not depend on other non-key attributes.
- Example: Instead of storing WarehouseLocation in the Orders table (which depends on ProductID rather than OrderID), the WarehouseLocations table stores this data separately. Orders reference ProductID, which in turn references WarehouseLocation in the Products table.
- This structure ensures data integrity by avoiding redundant updates. If a product moves to a different warehouse, only one update is needed in the Products table, rather than updating every order record.

Final Justification of 3NF:

- The Orders table only stores order-specific details (OrderID, EmployeeID, SKU, Quantity, etc.).
- The Products table only stores product-specific details (SKU, ProductName, SupplierID, etc.), preventing redundancy.
- Foreign keys link related data (EmployeeID in Orders, SupplierID in Products), ensuring consistency without duplication.
- This structure eliminates redundancy, ensures data integrity, and allows efficient querying.

Entity Relationships and Data Integrity

The relationships between tables are established using foreign keys, ensuring data consistency across different business functions. This structure follows best practices for referential integrity, preventing data duplication and improving maintainability.

Employees and Orders

Employees are responsible for fulfilling orders, so the database connects the Orders table to the Employees table. This ensures that every order is linked to the employee who manages it, such as a picker or packer.

- One employee can manage multiple orders, but each order is assigned to one specific employee.

Orders and Products

Each order involves a specific product, so the Orders table is connected to the Products table.

- Example: An order might be for a pair of running shoes or a gym bag.
- This relationship makes it easy to track what products are being picked, packed, or shipped in each order.

Products and Suppliers

Every product in the warehouse comes from a supplier. Instead of duplicating supplier details in the Products table, the database links Products to the Suppliers table.

- This structure prevents redundancy: if a supplier changes contact details, only one update is needed in the Suppliers table, rather than updating every product.

Products and WarehouseLocations

Products are stored in specific warehouse locations, so the Products table is connected to the WarehouseLocations table.

- Example: Some products might be stored in the Manchester warehouse, while others are in Birmingham.
- This relationship ensures that the system can track where each product is located and helps manage inventory effectively.

WarehouseLocations and Employees

Employees are assigned to specific warehouses, so the Employees table is linked to the WarehouseLocations table.

- Example: A picker in Glasgow is assigned to that specific warehouse, and their assigned location is stored in the database.
- This helps ensure efficient work allocation and prevents employees from being assigned to warehouses where they are not based.

Foreign Key Implementation & Referential Integrity

The database ensures data consistency by using foreign keys, preventing orphan records, and enforcing relational constraints:

- The Orders table references Employees using EmployeeID, ensuring that every order is linked to a valid employee.
- The Products table references Suppliers using SupplierID, avoiding duplicated supplier data.
- The Orders table references Products using SKU, ensuring that every order is linked to a valid product.

3. Part 2

3.1 Triggers

Before implementing the triggers, it was necessary to utilize the existing Orders and Stock tables to address the business requirements for inventory management. The triggers were created to ensure stock levels are automatically updated when orders are modified or deleted.

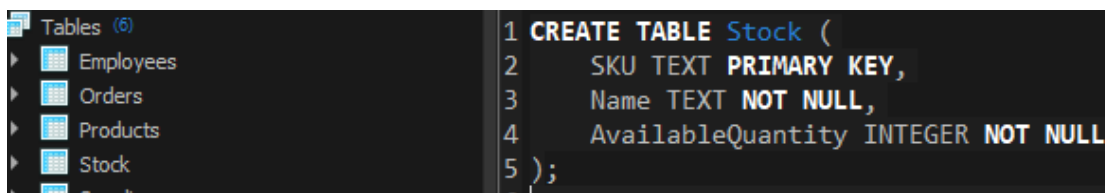


FIGURE 26 QUERY CREATE STOCK TABLE

The Stock table in Figure 22 manages inventory levels through the AvailableQuantity field, which reflects the current stock for each item. The ItemID serves as the primary key, linking to the existing Orders table to ensure referential integrity. Changes in the Orders table, such as updates or deletions, trigger automatic updates to the Stock table, ensuring accurate and synchronized inventory management.

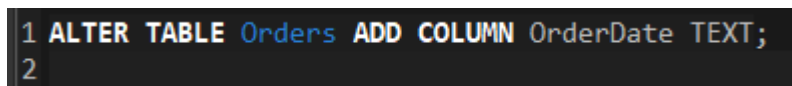


FIGURE 27 QUERY ALTER ORDERS TABLE

Trigger 1: Update Trigger

This trigger updates the Stock table whenever the Quantity of an order in the **Orders** table is updated.

```
1 CREATE TRIGGER update_stock_on_order_update
2 AFTER UPDATE OF Quantity ON Orders
3 FOR EACH ROW
4 BEGIN
5     -- Update Stock table based on SKU
6     UPDATE Stock
7     SET AvailableQuantity = AvailableQuantity + (OLD.Quantity - NEW.Quantity)
8     WHERE SKU = NEW.SKU;
9
10    -- Log the action into Stock_Log table
11    INSERT INTO Stock_Log (Action, SKU, QuantityDifference)
12    VALUES ('Order Updated', NEW.SKU, OLD.Quantity - NEW.Quantity);
13 END;
```

FIGURE 28 QUERY CREATE TRIGGER 1

The trigger is executed after the Quantity field in the Orders table is updated.

The difference between the OLD.Quantity and NEW.Quantity is calculated.

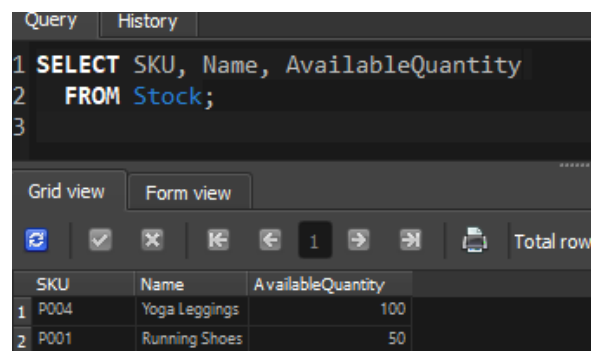
This difference is added (or subtracted) from the AvailableQuantity in the Stock table.

If the previous quantity was 5 (OLD.Quantity) and is updated to 8 (NEW.Quantity), the stock decreases by three.

Testing the Trigger

```
1 INSERT INTO Stock (SKU, Name, AvailableQuantity)
2 VALUES ('P004', 'Yoga Leggings', 100), ('P001', 'Running Shoes', 50);
3
```

FIGURE 29 QUERY INSERT INTO STOCK



The screenshot shows a database query tool interface. At the top, there are tabs for 'Query' and 'History'. Below the tabs, a SQL query is entered: `1 SELECT SKU, Name, AvailableQuantity` and `2 FROM Stock;`. Below the query editor, there are tabs for 'Grid view' and 'Form view'. The 'Grid view' is selected, and it displays a table with 3 columns: 'SKU', 'Name', and 'AvailableQuantity'. The table contains 2 rows of data.

	SKU	Name	AvailableQuantity
1	P004	Yoga Leggings	100
2	P001	Running Shoes	50

FIGURE 30 QUERY TRIGGER TESTING

Trigger 2: Delete Trigger

This trigger adjusts the Stock table when a record in the **Orders** table is deleted (e.g., an order is cancelled, or items are returned).

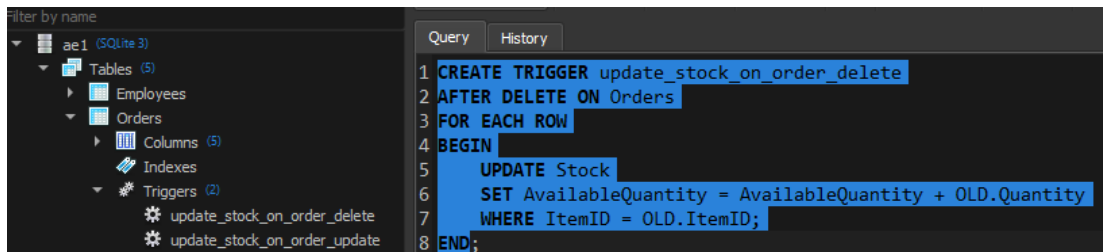


FIGURE 31 QUERY CREATE TRIGGER 2

The trigger is executed after a record in the Orders table is deleted.

The trigger retrieves the Quantity of the deleted order (OLD.Quantity).

This value is added back to the AvailableQuantity in the Stock table.

If an order for five items is deleted, the stock increases by five.

3.2 Views

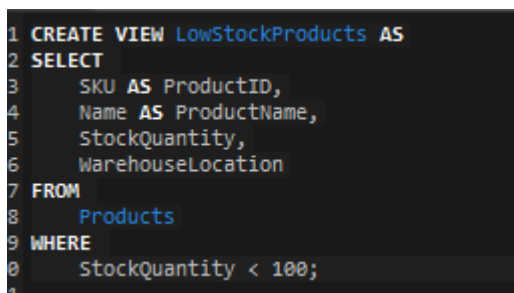


FIGURE 32 CREATE VIEW QUERY

The query in figure 32 creates a view that helps identify products that have stock below a certain threshold. This aligns with the business requirement for stock management and alerts when stock levels are low.

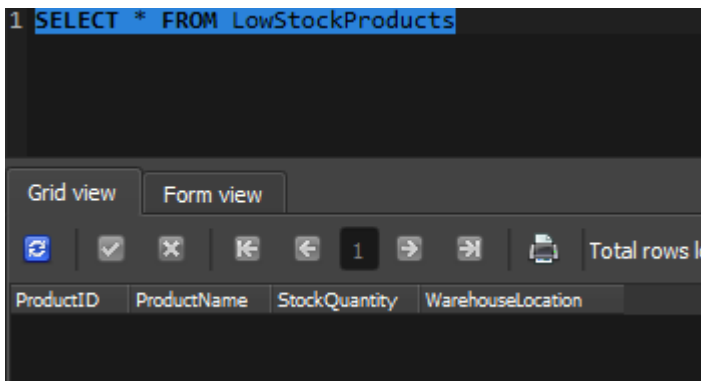


FIGURE 33 VIEW QUERY

This query in Figure 33 shows the view runs successfully there is no results as the stock in all locations is above one hundred.

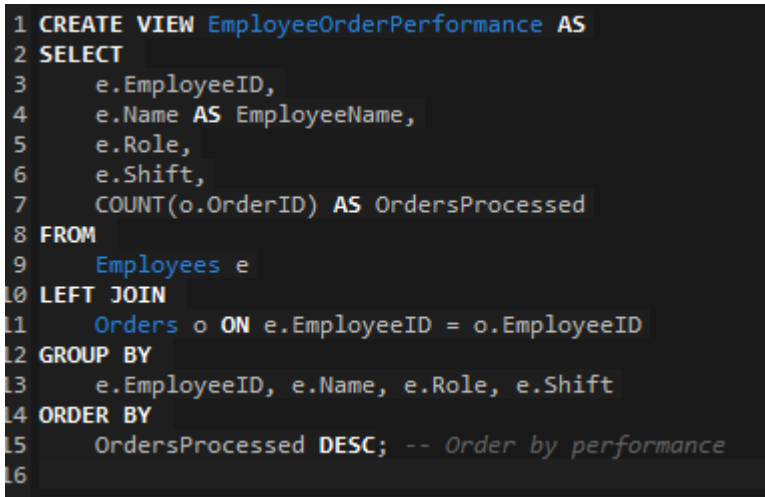


FIGURE 34 CREATE VIEW QUERY

This query in the Figure 33 creates a view that calculates the number of orders processed by each employee and includes their role and shift. This aligns with the business requirement to track employee performance.

```
1 SELECT * FROM EmployeeOrderPerformance
```

Grid view
Form view

1

Total rows loaded: 7

	EmployeeID	EmployeeName	Role	Shift	OrdersProcessed
1	E001	Alice Johnson	Picker	Morning	1
2	E002	Bob Smith	HR Assistant	Morning	1
3	E003	Charlie Brown	Picker	Evening	1
4	E004	Diana Carter	HR Assistant	Morning	1
5	E005	Edward Davis	HR Assistant	Evening	1
6	E000	Madara	CEO	Morning	0
7	E034	Wilson Melo	Control Room Assistant	Day	0

FIGURE 35 VIEW QUERY

The screenshot in Figure 35 shows the view being displayed the expected results that shows employee performance metrics.

The login and employee management system are designed to enhance security and streamline operations for Prime Warehousing Solutions. The system allows only authorised users, such as the Control Room Assistant, to add new employees to the database. This ensures data integrity and aligns with the business requirement of efficient staff management.

List the features included:

- Login System: Validates user credentials using hashed passwords stored in the SQLite database.
- Employee Management: Enables the Control Room Assistant to add new employees through a modern GUI interface.
- Security Features:
 - Passwords are hashed using SHA-256 for secure storage.
 - Error messages are displayed for invalid inputs or duplicate entries.
- User-Friendly GUI: Built using Tkinter, providing a simple and interactive interface.

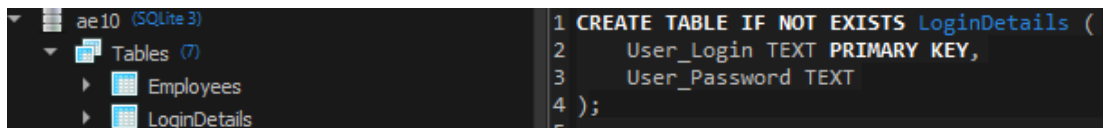


FIGURE 36 CREATE LOGINDETAILS QUERY

The query in Figure 32 was used to create the LoginDetails table, which stores user login credentials.

4. Part 3

```

1 import hashlib
2
3 def hash_password(password):
4     return hashlib.sha256(password.encode()).hexdigest()
5
6 # Hash Wilson Melo's password
7 hashed_password = hash_password("Pr1m3$0lu710n$!")
8 print(hashed_password)
9

```

```

C:\Users\andre\PycharmProjects\QH0541AE1\.venv\Scripts\python.exe
4464d0e9df62fec64b93ec181098e5a7dcbd220ec368cbbf68dc4fddb12b8d0d

Process finished with exit code 0

```

FIGURE 37 HASH PASSWORD METHOD

The Python code in Figure 37 demonstrates the function used to hash passwords using the SHA-256 algorithm. This ensures that passwords are not stored in plaintext, enhancing system security.

```

1 INSERT INTO LoginDetails (User_Login, User_Password)
2 VALUES ('wilson.melo', '4464d0e9df62fec64b93ec181098e5a7dcbd220ec368cbbf68dc4fddb12b8d0d');
3

```

FIGURE 38 INSERTING NEW USER AND HASHED PASSWORD

User_Login	User_Password
wilson.melo	4464d0e9df62fec64b93ec181098e5a7dcbd220ec368cbbf68dc4fddb12b8d0d

FIGURE 39 HASHED PASSWORD INSERTED

```
# Add employee function
def add_employee(employee_id, name, role, shift, location): 1 usage

    conn = sqlite3.connect(DB_NAME)
    cursor = conn.cursor()

    try:
        cursor.execute( sql: """
            INSERT INTO Employees (EmployeeID, Name, Role, Shift, Location)
            VALUES (?, ?, ?, ?, ?)
        """, parameters: (employee_id, name, role, shift, location))
        conn.commit()
        messagebox.showinfo( title: "Success", message: f"Employee {name} added successfully!")
    except sqlite3.IntegrityError:
        messagebox.showerror( title: "Error", message: "EmployeeID must be unique!")
    finally:
        conn.close()
```

FIGURE 40 EMPLOYEE FUNCTION

The Python code in Figure 40 is used to insert a new employee into the Employees table.

```
# Validate login function
def validate_login(username, input_password): 1 usage

    conn = sqlite3.connect(DB_NAME)
    cursor = conn.cursor()

    cursor.execute( sql: "SELECT User_Password FROM LoginDetails WHERE User_Login=?", parameters: (username,))
    result = cursor.fetchone()

    conn.close()

    if result:
        stored_password = result[0]
        if hash_password(input_password) == stored_password:
            messagebox.showinfo( title: "Login", message: "Login successful!")
            if username == "wilson.melo":
                open_control_room_interface()
        else:
            messagebox.showerror( title: "Login", message: "Invalid password!")
    else:
        messagebox.showerror( title: "Login", message: "Username not found!")
```

FIGURE 41 FUNCTION TO SECURE THE LOGIN

The Python code in Figure 41 is used to validate user login credentials. It compares the hashed version of the input password with the stored hashed password in the LoginDetails table.


```
# Main Login GUI
def open_login_window(): 1 usage
    def handle_login():
        username = username_var.get()
        password = password_var.get()
        validate_login(username, password)

    global root
    root = Tk()
    root.title("Login System")

    Label(root, text="Control Room Login").grid(row=0, column=0, columnspan=2, pady=10)

    Label(root, text="Username:").grid(row=1, column=0, sticky="e")
    username_var = StringVar()
    Entry(root, textvariable=username_var).grid(row=1, column=1)

    Label(root, text="Password:").grid(row=2, column=0, sticky="e")
    password_var = StringVar()
    Entry(root, textvariable=password_var, show="*").grid(row=2, column=1)

    Button(root, text="Login", command=handle_login).grid(row=3, column=0, columnspan=2, pady=10)

    root.mainloop()
```

FIGURE 42 LOGIN FOR GUI

The Python code in Figure 38 demonstrates the implementation of a Tkinter based login interface. It allows users to input their credentials and manages login validation.

```
# Open Control Room Interface
def open_control_room_interface(): 1 usage

def submit_employee():
    emp_id = emp_id_var.get()
    emp_name = emp_name_var.get()
    emp_role = emp_role_var.get()
    emp_shift = emp_shift_var.get()
    emp_location = emp_location_var.get()

    if not (emp_id and emp_name and emp_role and emp_shift and emp_location):
        messagebox.showerror(title="Error", message="All fields are required!")
    else:
        add_employee(emp_id, emp_name, emp_role, emp_shift, emp_location)

control_window = Toplevel(root)
control_window.title("Control Room Assistant")

Label(control_window, text="Add New Employee").grid(row=0, column=0, columnspan=2, pady=10)

Label(control_window, text="Employee ID:").grid(row=1, column=0, sticky="e")
emp_id_var = StringVar()
Entry(control_window, textvariable=emp_id_var).grid(row=1, column=1)

Label(control_window, text="Name:").grid(row=2, column=0, sticky="e")
emp_name_var = StringVar()
Entry(control_window, textvariable=emp_name_var).grid(row=2, column=1)

Label(control_window, text="Role:").grid(row=3, column=0, sticky="e")
emp_role_var = StringVar()
Entry(control_window, textvariable=emp_role_var).grid(row=3, column=1)

Label(control_window, text="Shift (e.g., Morning/Evening):").grid(row=4, column=0, sticky="e")
emp_shift_var = StringVar()
Entry(control_window, textvariable=emp_shift_var).grid(row=4, column=1)

Label(control_window, text="Warehouse Location:").grid(row=5, column=0, sticky="e")
emp_location_var = StringVar()
Entry(control_window, textvariable=emp_location_var).grid(row=5, column=1)

Button(control_window, text="Add Employee", command=submit_employee).grid(row=6, column=0, columnspan=2, pady=10)
```

FIGURE 43 CONTROL ROOM GUI

```
Label(control_window, text="Name:").grid(row=2, column=0, sticky="e")
emp_name_var = StringVar()
Entry(control_window, textvariable=emp_name_var).grid(row=2, column=1)

Label(control_window, text="Role:").grid(row=3, column=0, sticky="e")
emp_role_var = StringVar()
Entry(control_window, textvariable=emp_role_var).grid(row=3, column=1)

Label(control_window, text="Shift (e.g., Morning/Evening):").grid(row=4, column=0, sticky="e")
emp_shift_var = StringVar()
Entry(control_window, textvariable=emp_shift_var).grid(row=4, column=1)

Label(control_window, text="Warehouse Location:").grid(row=5, column=0, sticky="e")
emp_location_var = StringVar()
Entry(control_window, textvariable=emp_location_var).grid(row=5, column=1)

Button(control_window, text="Add Employee", command=submit_employee).grid(row=6, column=0, columnspan=2, pady=10)
```

FIGURE 44 CONTROL ROOM GUI

The GUI shown in Figure 39 and 40 provides a simple interactive interface for the Control Room Assistant to add new employees to the database. It includes fields for entering employee details such as Employee ID, Name, Role, Shift, and Warehouse Location. Upon submission, the data is validated and added to the Employees table.

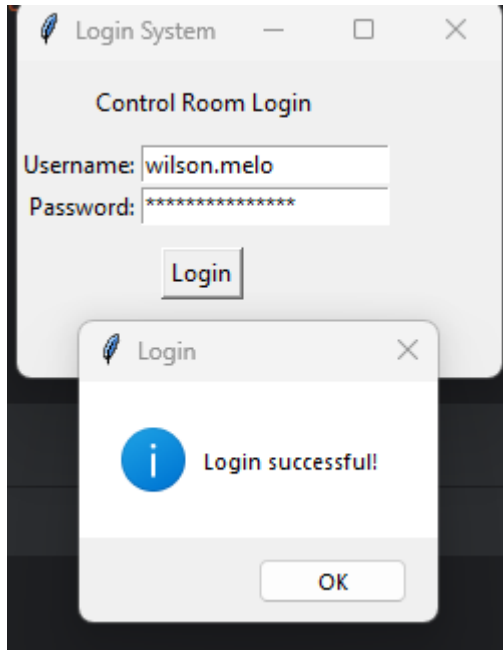


FIGURE 45 LOG IN WORKING

The screenshot in Figure 41 shows a successful login to the Control Room Assistant interface.

Login : wilson.melo

Password : Pr1m3\$0lu710n\$!

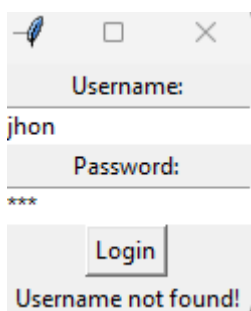


FIGURE 46 WRONG USERNAME

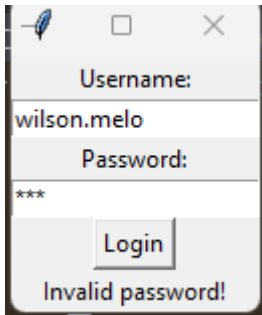


FIGURE 47 WRONG PASSWORD

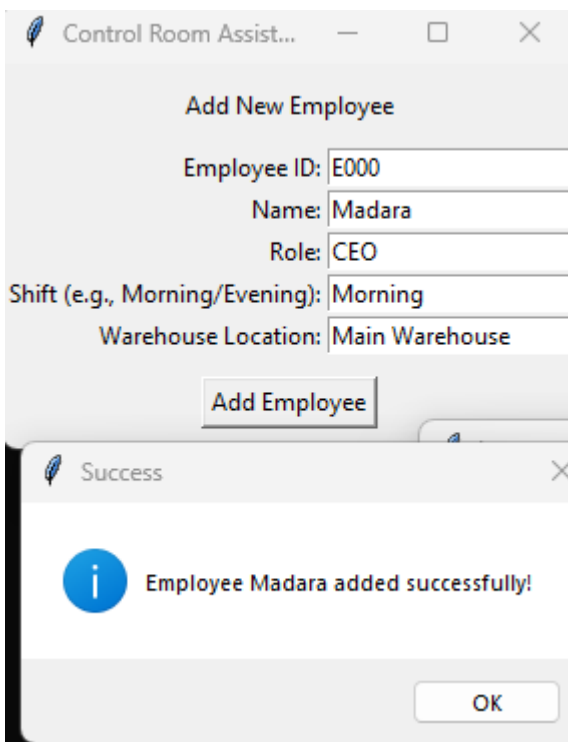


FIGURE 48 EMPLOYEE ADDED

The screenshot in figure 42 shows the user successfully adding a new employee into the system.

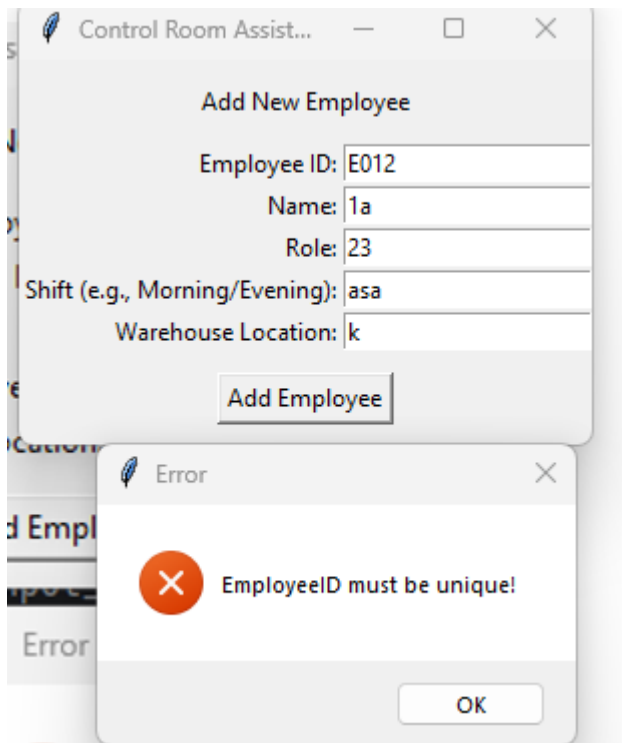


FIGURE 49 UNIQUE EMPLOYEEID

In the screenshot in Figure 45 the user attempted to add EmployeeID twice as a result an error message pop-up once he tries to add the Employee.

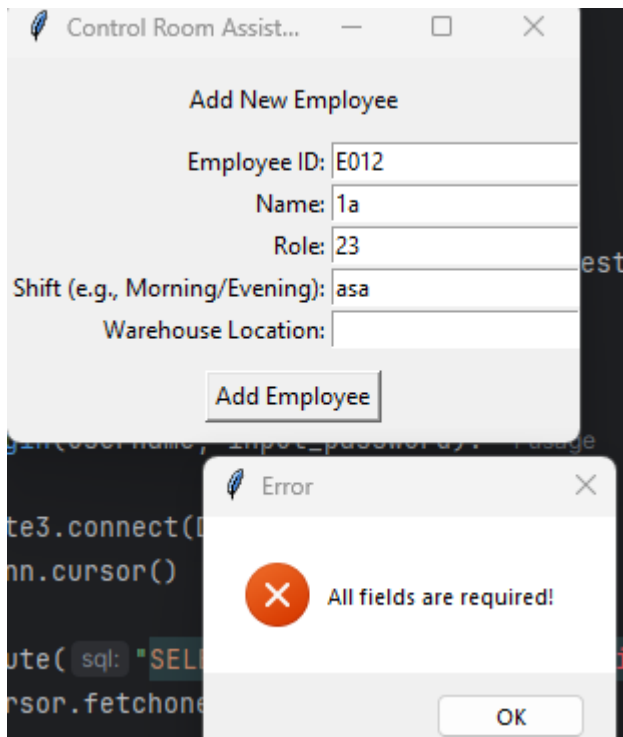


FIGURE 50 EMPLOYEE ADDING ATTEMPT

In this screenshot the user tries to attempt to add a new employee, but all fields are required as the dataset does not accept null values.

XML

Python Script for Exporting Orders to XML

```
import sqlite3
import xml.etree.ElementTree as ET

# Connect to SQLite
conn = sqlite3.connect("ae1.db") # Replace with your database file
cursor = conn.cursor()

# Extract data from Orders table
cursor.execute("SELECT OrderID, SKU, Quantity FROM Orders")
orders = cursor.fetchall()
```

```
# Create the root XML element
root = ET.Element("Orders")

# Loop through database rows and create XML structure
for order in orders:
    order_element = ET.SubElement(root, "Order")
    ET.SubElement(order_element, "OrderID").text = str(order[0])
    sku_element = ET.SubElement(order_element, "SKU")
    sku_element.set("type", "Product") # Add attribute type="Product"
    sku_element.text = str(order[1])
    ET.SubElement(order_element, "Quantity").text = str(order[2])

# Function to add indentation for readability
def prettify(elem, level=0):
    indent = "\n" + " " * level
    if len(elem):
        if not elem.text or not elem.text.strip():
            elem.text = indent + " "
        if not elem.tail or not elem.tail.strip():
            elem.tail = indent
        for subelem in elem:
            prettify(subelem, level + 1)
        if not elem.tail or not elem.tail.strip():
            elem.tail = indent
    else:
        if level and (not elem.tail or not elem.tail.strip()):
            elem.tail = indent

# Apply formatting to the XML
prettify(root)

# Save the XML file
tree = ET.ElementTree(root)
with open("orders.xml", "wb") as xml_file:
    tree.write(xml_file, encoding="utf-8", xml_declaration=True)
```

```
print("XML file 'orders.xml' created successfully!")
```

This Python script extracts order data from an SQLite database and converts it into an XML format. It uses the `xml.etree.ElementTree` library to structure the data, ensuring proper indentation for readability. The script also saves the formatted XML content to a file named `orders.xml`, which can be used for external processing and data exchange.

Results of python script:

```
1  <?xml version='1.0' encoding='utf-8'?>
2  <Orders>
3    <Order>
4      <OrderID>0001</OrderID>
5      <SKU type="Product">P002</SKU>
6      <Quantity>62</Quantity>
7    </Order>
8    <Order>
9      <OrderID>0002</OrderID>
10     <SKU type="Product">P003</SKU>
11     <Quantity>57</Quantity>
12   </Order>
13   <Order>
14     <OrderID>0003</OrderID>
15     <SKU type="Product">P004</SKU>
16     <Quantity>68</Quantity>
17   </Order>
18   <Order>
19     <OrderID>0004</OrderID>
20     <SKU type="Product">P005</SKU>
21     <Quantity>59</Quantity>
22   </Order>
23   <Order>
24     <OrderID>0005</OrderID>
25     <SKU type="Product">P001</SKU>
26     <Quantity>65</Quantity>
27   </Order>
28 </Orders>
```

FIGURE 51 FORMATTED XML OUTPUT OF ORDERS

This XML output represents the structured order data retrieved from the SQLite database. Each `<Order>` element contains an `<OrderID>`, an `<SKU>` with an attribute `type="Product"`, and a `<Quantity>`. The hierarchical structure ensures that the data is well-organized and readable for further processing.

BLOB

A table called EmployeePhotos was created, where:

- EmployeeID is the primary key.
- Photo is a BLOB column that stores image data.

```
CREATE TABLE EmployeePhotos (  
    EmployeeID TEXT PRIMARY KEY,  
    Photo BLOB  
);
```

FIGURE 52 SQL CODE TO CREATE THE TABLE



FIGURE 53 EMPLOYEE_PHOTO

```
import sqlite3

# Function to Convert Image to Binary
def convert_to_binary(filename):
    with open(filename, "rb") as file:
        return file.read()

# Connect to SQLite
conn = sqlite3.connect("ae1.db")
cursor = conn.cursor()

# Insert Image into Database
employee_id = "E034" # Now stored as TEXT
image_data = convert_to_binary("employee_photo.jpg") # Change to your image file

cursor.execute("INSERT INTO EmployeePhotos (EmployeeID, Photo) VALUES (?, ?)", (employee_id, image_data))

# Commit and Close
conn.commit()
conn.close()

print(f"Image stored successfully for EmployeeID: {employee_id}")
```

FIGURE 54 PYTHON SCRIPT FOR INSERTING AN IMAGE INTO SQLITE

The Python script in Figure 53 reads an image file, converts it into binary format, and stores it in an SQLite database using the BLOB (Binary Large Object) data type. The script ensures that images are securely stored within the database rather than as external file paths.

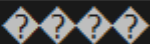
EmployeeID	Photo
E034	

FIGURE 55 IMAGE SUCCESSFULLY INSERTED INTO SQLITE

The SQL query in image 54 confirms that the image was successfully inserted into the EmployeePhotos table. The table now contains a binary representation of the image associated with an EmployeeID.

```
import sqlite3

# Function to Write Binary Data Back to an Image File
def write_to_file(binary_data, output_filename):
    with open(output_filename, "wb") as file:
        file.write(binary_data)

# Connect to SQLite
conn = sqlite3.connect("ae1.db")
cursor = conn.cursor()

# Retrieve Image
employee_id = "E034" # String format for EmployeeID
cursor.execute("SELECT Photo FROM EmployeePhotos WHERE EmployeeID = ?", (employee_id,))
image_data = cursor.fetchone()[0] # Fetch Binary Data

# Save the Image File
write_to_file(image_data, "retrieved_photo.jpg")

# Close Connection
conn.close()

print(f"Image retrieved and saved as 'retrieved_photo.jpg' for EmployeeID: {employee_id}")
```

FIGURE 56 PYTHON SCRIPT FOR RETRIEVING AN IMAGE FROM SQLITE

This Python script in figure 55 extracts the binary image data from SQLite, converts it back into an image file, and saves it to disk as retrieved_photo.jpg. This ensures that stored images can be retrieved and used whenever needed.



FIGURE 57 DISPLAYING THE RETRIEVED IMAGE

This screenshot in the figure 57 shows the retrieved_photo.jpg image opened, demonstrating that the image is identical to the original and was stored and retrieved without data loss.

Innovation chosen Threads

```
import threading
import sqlite3
import random
import time

# Thread class for inserting orders
class OrderInsertThread(threading.Thread):
    def __init__(self, thread_id, db_name, orders):
        threading.Thread.__init__(self)
        self.thread_id = thread_id
        self.db_name = db_name
        self.orders = orders

    def run(self):
        conn = sqlite3.connect(self.db_name)
        cursor = conn.cursor()

        # Insert each order in this threads batch
        for order in self.orders:
            cursor.execute("""
                INSERT INTO Orders (OrderID, EmployeeID, SKU, WarehouseLocation, Quantity, OrderDate)
                VALUES (?, ?, ?, ?, ?, ?)
            """, order)

        conn.commit()
        conn.close()
        print(f"Thread {self.thread_id} finished inserting orders!")

# Sample Order Data
orders_data = [
    ("O201", "E001", "P005", "WH-1", random.randint(1, 20), "2024-05-10"),
    ("O202", "E002", "P002", "WH-2", random.randint(1, 20), "2024-02-10"),
    ("O203", "E003", "P007", "WH-3", random.randint(1, 20), "2024-12-10"),
    ("O204", "E004", "P003", "WH-1", random.randint(1, 20), "2024-06-10"),
    ("O205", "E005", "P006", "WH-2", random.randint(1, 20), "2025-01-10"),
    ("O206", "E000", "P001", "WH-3", random.randint(1, 20), "2024-09-10"),
    ("O207", "E034", "P004", "WH-1", random.randint(1, 20), "2024-02-10"),
    ("O208", "E0023", "P008", "WH-2", random.randint(1, 20), "2024-04-10")
]

# Split orders into 2 batches
batch1 = orders_data[:4] # First half of orders
batch2 = orders_data[4:] # Second half of orders

# Create and Start Threads
threads = []
t1 = OrderInsertThread(1, "ae1.db", batch1)
t2 = OrderInsertThread(2, "ae1.db", batch2)
```

```
t1.start()
t2.start()

t1.join()
t2.join()

print("All orders inserted successfully using threading!")
```

This script uses threading to insert multiple orders into the SQLite database simultaneously. By creating separate threads for different batches of orders, the warehouse system can process transactions faster, ensuring efficient inventory management.

```
Thread 1 finished inserting orders!
Thread 2 finished inserting orders!
All orders inserted successfully using threading!

Process finished with exit code 0
```

FIGURE 58 THREAD 1 AND 2 FINISHED INSERTING ORDERS

	OrderID	Employee	SKU	Warehou	Quantity	OrderDate
1	O001	E001	P002	WH-2	62	NULL
2	O002	E002	P003	WH-3	57	NULL
3	O003	E003	P004	WH-4	68	NULL
4	O004	E004	P005	WH-5	59	NULL
5	O005	E005	P001	WH-1	65	NULL
6	O201	E001	P005	WH-1	1	2024-05-10
7	O202	E002	P002	WH-2	2	2024-02-10
8	O203	E003	P007	WH-3	8	2024-12-10
9	O204	E004	P003	WH-1	1	2024-06-10
10	O205	E005	P006	WH-2	15	2025-01-10
11	O206	E000	P001	WH-3	12	2024-09-10
12	O207	E034	P004	WH-1	18	2024-02-10
13	O208	E0023	P008	WH-2	4	2024-04-10

FIGURE 59 ORDERS SUCCESSFULLY INSERTED IN PARALLEL

The figure 58 shows the Orders table in SQLite after multi-threaded insertion. Orders are stored efficiently, demonstrating how parallel processing enhances database operations.

Benefits of Using Threading in Warehouse System:

Advantage	Why It Matters in Warehousing
Faster Order Processing	Multiple orders are inserted simultaneously, improving efficiency.
Better System Performance	Reduces waiting time when managing large data.
Real-World Relevance	Used in warehouse automation where multiple orders are processed at the same time.
Innovation Feature	Enhances the project, fulfilling the assessment's advanced feature requirement.

It is important to remember that SQLite handles multi-threading through built-in write locks, ensuring only one thread writes at a time. This prevents data inconsistencies when updating stock during parallel order processing.

Conclusion

This assessment successfully demonstrates the implementation of an advanced database system for Prime Warehousing Solutions, integrating SQLite, Python, XML, BLOB storage, and multi-threading. The database follows Third Normal Form to ensure data integrity and efficiency, while triggers, views, and functions automate key business processes. The system includes security features such as password hashing and SQL Injection prevention, ensuring data protection and reliability. The multi-threading innovation enhances performance by processing orders in parallel, reducing system delays, this project presents a well-structured, scalable, and secure database system, effectively addressing warehouse management challenges. With further improvements, such as enhanced stock validation and query optimization, the system could be scaled for real-world enterprise use.

References

Solent University (2024) *QH0541 Advanced Database Systems*.

Appendices



Prime%20Warehousing%20Solutions.xls



ae1.db



AE1GUI.py



AE1Tkinter.py



BLOB.py



hash.py



Threads.py



xml_module.py