National Economics University

School of Advanced Education Program

Database Management Final Exam Report

Group 9

Hoang Anh Thu - 11226077

Pham Lam Ha - 11221962

Pham Thuy Dung - 11221461

Tran Thi Hoai Trang - 11226527

Supervisor: Tran Manh Hung

Ha Noi, May 30, 2025

Contents

List of Figures **3**

List of Tables **5**

1 Introduction **6**

2 Database design and implementation **7**

3 Advanced database objects **9**
3.1 VIEW 9
3.1.1 Overview of Current Production Orders 9
3.1.2 VIEW: Material Requirements for Active Production Orders . . 10
3.1.3 VIEW: Tracking Current Supplier Deliveries 11
3.2 TRIGGER: Automatically update product stock after production completion 12
3.3 User Defined Functions: Calculate material cost per product unit . . . 14
3.4 INDEX and Optimization Query 15
3.4.1 Use Case 1: Single-column Index on a Simple WHERE Condition 15
3.4.2 Use Case 2: Composite Index with WHERE Clause Containing Two or More Conditions 17
3.4.3 Use Case 3: Covering Index for SELECT + WHERE – Optimize query to avoid accessing base table 19
3.4.4 Use case 4: Index in JOIN 21
3.5 Stored Procedures 23
3.5.1 Inventory updates: 23
3.5.2 Automate production order creation include inventory update: . 25

4 Database Security and Administration **28**
4.1 Create roles for production managers, warehouse staff, and finance . . 28
4.2 Protect sensitive supplier and pricing data through permissions and encryption 30
4.3 Develop backup and recovery plans 31

5 Python Application Development **38**
5.1 Database connection 38
5.2 Operational Scripts 39
5.2.1 Inventory updates 39
5.2.2 Automate production order creation include inventory update: . 40
5.3 Reporting module 41
5.3.1 Monthly Completed Orders 41
5.3.2 Stock Quantity by Product Category 42
5.3.3 Material Unit Cost Distribution 42
5.3.4 Order Status Distribution 43

5.4 User Interface 44
5.4.1 Manufacturing Orders tab 44
5.4.2 Products Catalog tab 47
5.4.3 Raw materials tab 49
5.5 Suppliers tab 50

6 Conclusion **51**
6.1 Summary of Achievements: 51
6.2 Recommendations: 51

References **52**

List of Figures

2.1 ER diagram for the production workflow 7
2.2 Relational schema with PKs, FKs, and constraints 7

3.1 CREATE VIEW Current Production Orders 9
3.2 CREATE VIEW MaterialUsage_InProgressn 10
3.3 CREATE VIEW track_supplier_deli 11
3.4 CREATE TABLE logs 12
3.5 CREATE TRIGGER orders_AFTER_UPDATE 13
3.6 Practical testing of TRIGGER 13
3.7 CREATE FUNCTION MaterialCostPerUnit 14
3.8 Single-column Index on a Simple WHERE Condition 16
3.9 CREATE INDEX 16
3.10 Explain without INDEX 16
3.11 Analyze without INDEX 16
3.12 Explain with INDEX 16
3.13 Analyze with INDEX 16
3.14 Use Case 2 17
3.15 CREATE INDEX orders product status 17
3.16 index 18
3.17 Composite index is usable 18
3.18 Composite index 18
3.19 Composite index is not usable 19
3.20 EXPLAIN query: Composite index is not usable 19
3.21 Leftmost Prefix 19
3.22 Use case 3 20
3.23 Create index on StartDate column 20
3.24 Create covering index 20
3.25 Explain Composite index 20
3.26 EXPLAIN ANALYZE: Index in JOIN 1 21
3.27 QUERY PLAN: Hash join 21
3.28 EXPLAIN ANALYZE: Index in JOIN 2 22
3.29 QUERY PLAN: Nested loop join 22
3.30 Nested loop join 23
3.31 Procedure UpdateMaterialInventory 1 23
3.32 Procedure UpdateMaterialInventory 2 24
3.33 Procedure UpdateMaterialInventory 3 24
3.34 Store procedure: CreateProductionOrder 1 25
3.35 Store procedure: CreateProductionOrder 2 26

4.1 Create user 28
4.2 Grant privileges 28

4.3 Grant permission 29
4.4 Insert data 30
4.5 Insert data 31
4.6 Output 31
4.7 SELECT with AES_DECRYPT statement 31
4.8 Output 32
4.9 Create backup script file 32
4.10 Run script file 33
4.11 Backup files 33
4.12 Open app Task Scheduler 33
4.13 Click create basic task 33
4.14 Name the task 34
4.15 Choose time 34
4.16 Select start date 35
4.17 Start a program 35
4.18 Click browse 36
4.19 Click finish 36
4.20 Click "Task Scheduler Library" 37

5.1 Import mysql 38
5.2 Import mysql 38
5.3 Establish connection 39
5.4 Monthly completed order 41
5.5 Stock Quantity by Product Category 42
5.6 Material Unit Cost Distribution 43
5.7 Order Status Distribution 43
5.8 Manufacturing Orders tab 45
5.9 Search functionality 45
5.10 All fields are required 46
5.11 Success: Manufacturing order created successfully 46
5.12 New manufacturing order appears on the treeview (status = in progress) 46
5.13 Codes for processing the manufacturing order 47
5.14 Insufficient stock (Order O0202, Product P015) 47
5.15 Products catalog tab 47
5.16 Bill of materials interface 48
5.17 Raw materials tab 49
5.18 Successfully purchased material (M021) 49
5.19 Suppliers tab 50

List of Tables

2.1 Overview of Main Relationships Among Production Entities 8

3.1 Index_Order_Table 15

1 Introduction

- **Project Overview and Objectives**

This project aims to design and implement an integrated production and inventory management system tailored for manufacturing and supply chain operations. The system is intended to streamline the entire workflow, from managing production orders, tracking materials and components, to updating stock quantities automatically based on order completion.

Key objectives include improving data accuracy, enhancing real-time visibility of production status, and enabling better decision-making through timely reporting and alerts. Ultimately, the project strives to reduce operational inefficiencies, optimize resource allocation, and ensure a seamless flow of materials from suppliers to production and delivery.

- **Importance of Production and Inventory Management**

Efficient production and inventory management are fundamental to the success of manufacturing businesses.

Proper coordination of production schedules with inventory levels helps prevent costly delays, avoid stock shortages, and minimize excess inventory holding costs. By closely monitoring production orders and stock movements, companies can respond swiftly to changes in demand, maintain product quality, and improve customer satisfaction.

Moreover, automated inventory updates and accurate record-keeping reduce human errors and administrative overhead. Good management practices in this area contribute directly to lowering operational costs, improving cash flow, and gaining a competitive advantage in the market.

2 Database design and implementation

For more information, visit Database

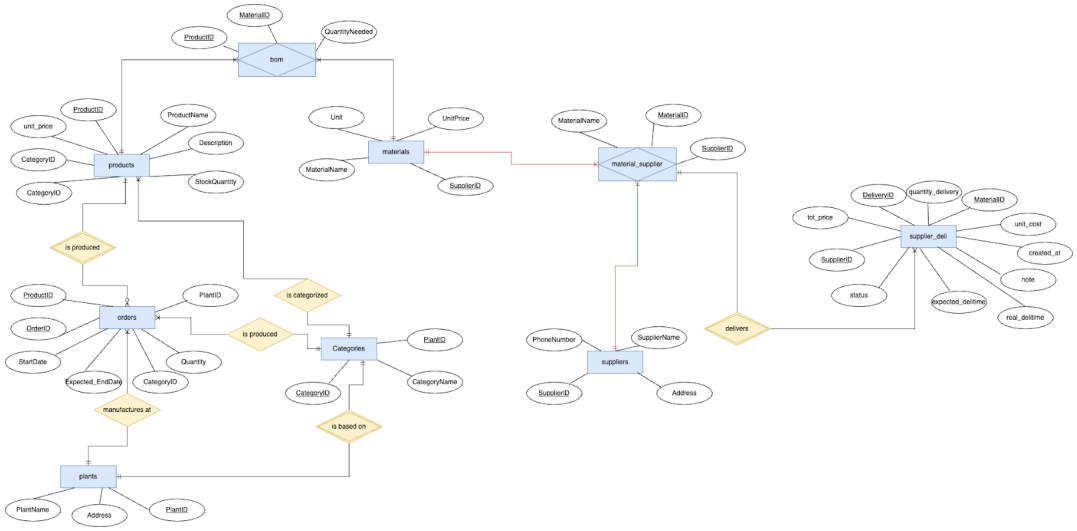


Figure 2.1: ER diagram for the production workflow

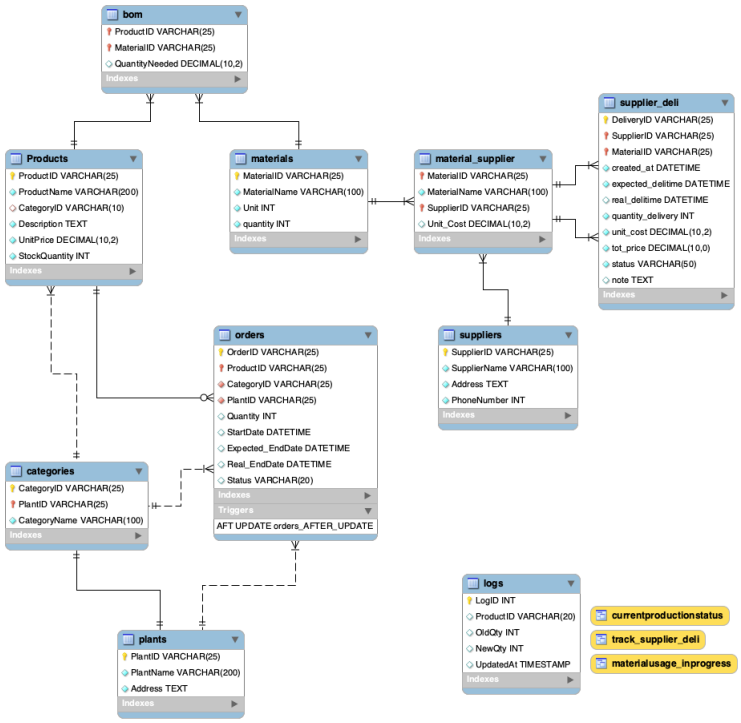


Figure 2.2: Relational schema with PKs, FKs, and constraints

The table below shows the main relationships between different entities.

Entity 1	Entity 2	Relationship description	Connectivity (Cardinality)	Participation (Mandatory/ Optional)
products	orders	One product can be in many orders; each order has exactly one product. There may exist products that do not appear in any order.	products (1, N) orders	(1,1) products, (0,N) orders
products	categories	Each product belongs to only one category. Each category can include many products.	products (N, 1) categories	(1,N) products, (1,1) categories
products	bom	Many-to-many relationship between products and materials through bom. A product may require multiple materials.	(N, M)	(1,1) products, (1,N) bom
materials	bom	Many-to-many relationship between materials and products through bom. A material can be used in multiple products.	(N, M)	(1,1) materials, (1,N) bom
materials	material_ supplier	Many-to-many relationship between materials and suppliers through material_supplier. Each material can be supplied by multiple suppliers.	(N, M)	(1,1) materials, (1,N) material_supplier
suppliers	material_ supplier	Many-to-many relationship between materials and suppliers through material_supplier. Each supplier can supply multiple materials.	(1, N)	(1,1) suppliers, (1,N) material_supplier
material_ supplier	supplier_deli	One supplier can have many deliveries.	(1, N)	(1,1) material_supplier, (1,N) supplier_deli
categories	plants	Each category is based on one plant. Each plant can manufacture one category.	categories (1, 1) plants	(1,1) categories, (1,1) plants
orders	plants	Each order is manufactured at one plant. Each plant can manufacture many orders.	orders (N, 1) plants	(1,N) orders, (1,1) plants

Table 2.1: Overview of Main Relationships Among Production Entities

For more information, visit DBMS Final

The relational schema defines the structure of the database used for managing production, inventory, orders, and supplier information. It organizes data into 9 related tables with defined keys to ensure data integrity and efficient querying, 1 logs table for tracking changes and audit purposes.

The database includes approximately 200 rows in the Orders table, reflecting active and completed production orders. This setup enables effective tracking of production status, inventory levels, and supplier deliveries.

Overall, the schema supports smooth operational workflows and informed decision-making within the manufacturing and supply chain processes.

3 Advanced database objects

3.1 VIEW

3.1.1 Overview of Current Production Orders

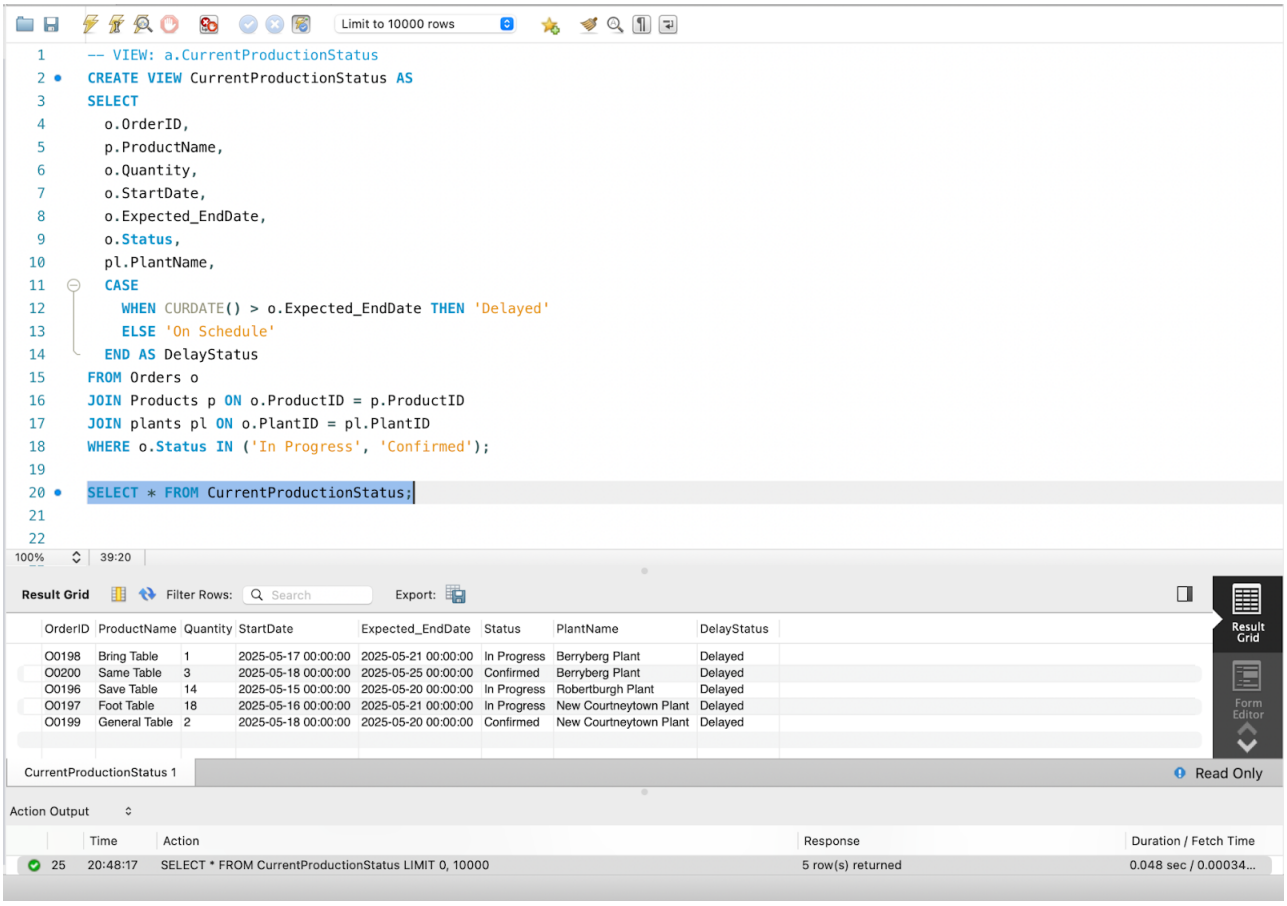


Figure 3.1: CREATE VIEW Current Production Orders

For more information, visit DBMS Final

The CurrentProductionStatus view consolidates detailed information about production orders that currently have a status of 'In Progress' or 'Confirmed'. This view joins the Orders, Products, and Plants tables to retrieve key details such as order ID, product name, quantity, start date, expected end date, current status, and the production plant's name.

A significant feature of this view is the inclusion of the DelayStatus field, which is generated using a CASE statement. This field evaluates whether each order is delayed based on the comparison between the current date and the order's expected completion date. Specifically:

- If the current date (CURDATE()) is later than the order's Expected_EndDate, the order is flagged as 'Delayed'.
- Otherwise, the order is marked as 'On Schedule'.

This conditional logic provides management with an immediate visual indicator of orders that are behind schedule, enabling them to prioritize interventions and resource allocation effectively.

Overall, the view offers a comprehensive snapshot of ongoing production activities, supporting better operational monitoring and timely decision-making to enhance production efficiency.

3.1.2 VIEW: Material Requirements for Active Production Orders

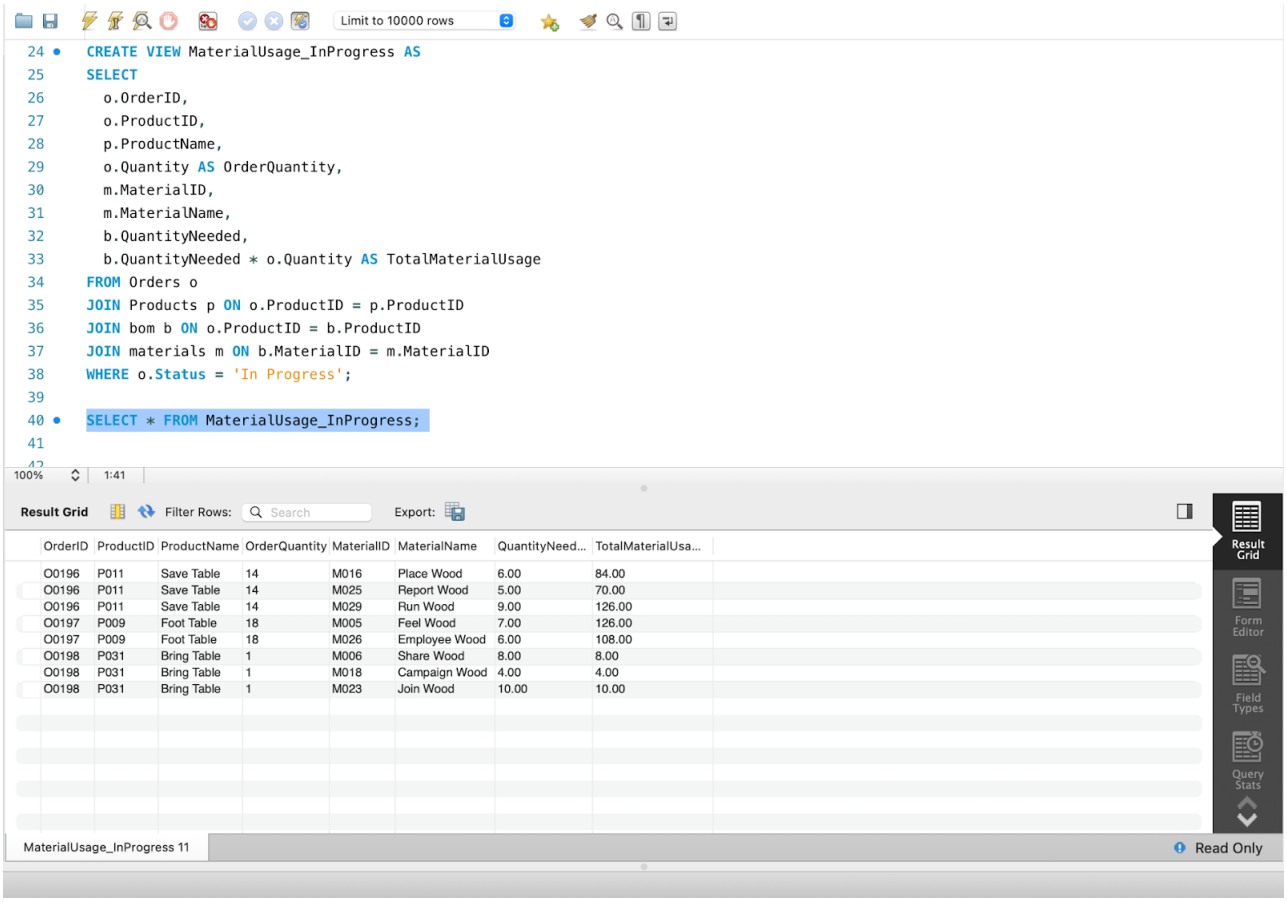


Figure 3.2: CREATE VIEW MaterialUsage_InProgressn

For more information, visit DBMS Final

This view calculates the raw material quantities required for production orders currently marked as 'In Progress'. It retrieves the order and product details along with the required materials from the Bill of Materials (bom). It also computes the total amount of each material needed by multiplying the per-unit material requirement by the order quantity. This facilitates procurement planning and inventory control.

3.1.3 VIEW: Tracking Current Supplier Deliveries

For more information, visit VIEW

```
CREATE VIEW track_supplier_deli AS
SELECT
    sd.DeliveryID,
    s.SupplierID,
    m.MaterialID,
    sd.created_at,
    sd.expected_delitime,
    sd.quantity_delivery,
    sd.tot_price,
    sd.status,
    DATEDIFF(sd.expected_delitime, CURDATE()) AS DaysLeft
FROM supplier_deli sd
JOIN suppliers s ON sd.SupplierID = s.SupplierID
JOIN materials m ON sd.MaterialID = m.MaterialID
WHERE sd.status = 'In Delivery';


SELECT *
FROM track_supplier_deli
WHERE DaysLeft <= 3; -- To filter orders with less than 3 days left
```

100% 68:79

Result Grid | Filter Rows: Search | Export:

DeliveryID	SupplierID	MaterialID	created_at	expected_delitime	quantity_delive...	tot_price	status	DaysLeft
DL007	S010	M011	2025-05-16 00:00:00	2025-05-20 00:00:00	3	36	In Delivery	-6
DL008	S002	M003	2025-05-16 00:00:00	2025-05-20 00:00:00	48	1680	In Delivery	-6
DL009	S018	M022	2025-05-16 00:00:00	2025-05-21 00:00:00	48	528	In Delivery	-5
DL010	S019	M026	2025-05-16 00:00:00	2025-05-21 00:00:00	160	3680	In Delivery	-5
DL011	S005	M006	2025-05-16 00:00:00	2025-05-21 00:00:00	8	178	In Delivery	-5
DL012	S016	M018	2025-05-17 00:00:00	2025-05-21 00:00:00	4	92	In Delivery	-5
DL013	S018	M023	2025-05-18 00:00:00	2025-05-24 00:00:00	10	230	In Delivery	-2
DL014	S008	M010	2025-05-19 00:00:00	2025-05-26 00:00:00	8	88	In Delivery	0
DL015	S010	M011	2025-05-20 00:00:00	2025-05-28 00:00:00	18	216	In Delivery	2

Result Grid | Form Editor | Field Types

track_supplier_deli 6 | Read Only

Figure 3.3: CREATE VIEW track_supplier_deli

The track_supplier_deli view is designed to provide a comprehensive overview of all ongoing supplier deliveries that are currently in transit. By integrating data from delivery records, supplier details, and material information, this view consolidates essential insights into each delivery's status and timeline.

One of the key features of this view is the calculation of the remaining days until the expected delivery date. This calculation enables users to quickly identify how much time is left before each shipment is due, facilitating proactive management and timely decision-making.

The practical benefit of this view lies in its ability to help supply chain managers and logistics teams monitor the progress of active deliveries. By filtering the view to highlight shipments due within a critical time frame, stakeholders can prioritize follow-ups, mitigate risks of delays, and ensure inventory replenishment aligns with operational demands.

Overall, this view enhances transparency across the delivery process, improves coordination between procurement and inventory functions, and supports a more efficient, data-driven approach to supply management.

3.2 TRIGGER: Automatically update product stock after production completion

For more information, visit TRIGGER

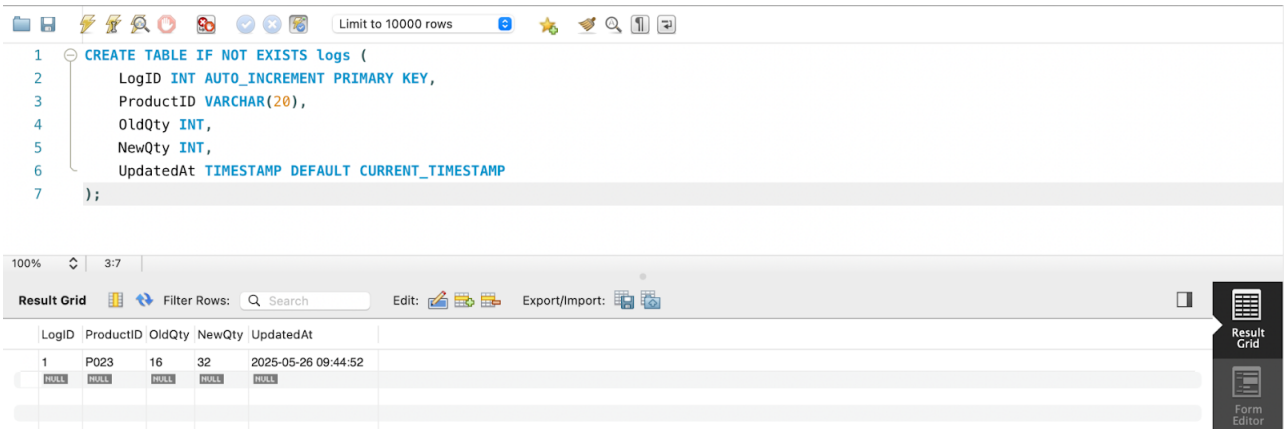


Figure 3.4: CREATE TABLE logs

The creation of a table named logs is designed to record the history of stock quantity updates for products.

This table consists of several columns, including LogID, which is an auto-incrementing primary key to uniquely identify each log entry. The ProductID column stores the identifier of the product whose stock level is being tracked. Two integer columns, OldQty and NewQty, record the stock quantity before and after the update, respectively. Additionally, the UpdatedAt column is a timestamp that defaults to the current time when the log entry is created, providing a precise record of when the stock change occurred.

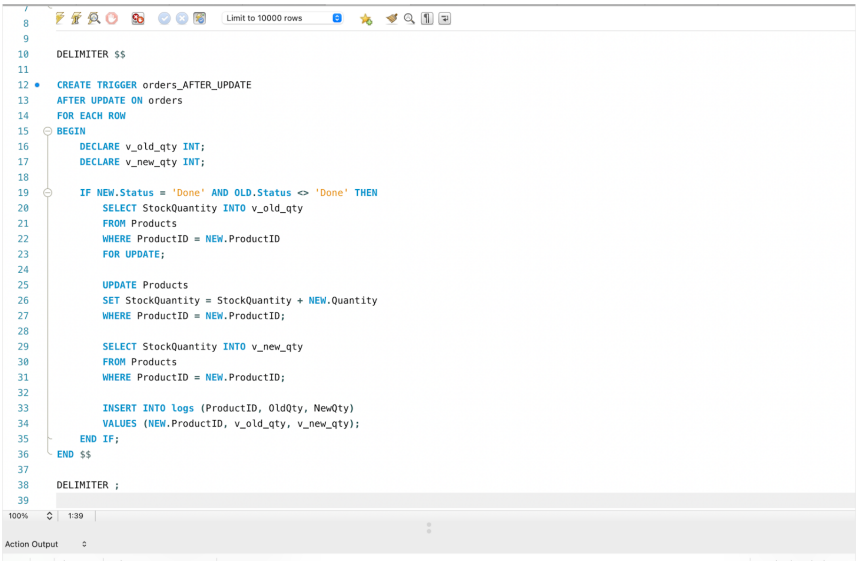
The purpose of this table is to maintain an audit trail for inventory changes, which is essential for tracking stock movement and diagnosing discrepancies.

Figure 3.5 illustrates the creation of an AFTER UPDATE trigger named orders_AFTER_UPDATE on the orders table. This trigger activates for each row whenever an update operation occurs on the orders table.

Two integer variables are declared to store the old and new stock quantities of a product. The trigger logic specifically checks if the status of an order changes to 'Done' from any other status, indicating the completion of production for that order.

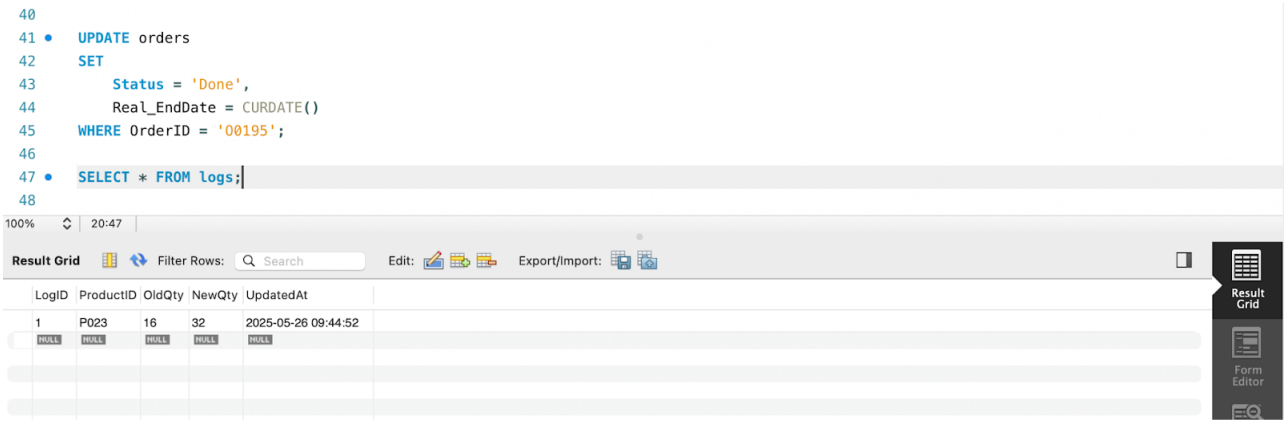
Once this condition is met, the trigger retrieves the current stock quantity of the relevant product, ensuring data integrity by locking the row for update using FOR UPDATE. Subsequently, it updates the stock quantity in the Products table by adding the quantity from the completed order to the existing stock. After updating, it fetches the new stock quantity.

Finally, the trigger inserts a record into the logs table with the product ID, old stock quantity, and new stock quantity, thereby logging this transaction. This automated mechanism ensures that the inventory is immediately updated and changes are transparently recorded without requiring manual intervention.



```
DELIMITER $$

CREATE TRIGGER orders_AFTER_UPDATE
AFTER UPDATE ON orders
FOR EACH ROW
BEGIN
    DECLARE v_old_qty INT;
    DECLARE v_new_qty INT;

    IF NEW.Status = 'Done' AND OLD.Status <> 'Done' THEN
        SELECT StockQuantity INTO v_old_qty
        FROM Products
        WHERE ProductID = NEW.ProductID
        FOR UPDATE;

        UPDATE Products
        SET StockQuantity = StockQuantity + NEW.Quantity
        WHERE ProductID = NEW.ProductID;

        SELECT StockQuantity INTO v_new_qty
        FROM Products
        WHERE ProductID = NEW.ProductID;

        INSERT INTO logs (ProductID, OldQty, NewQty)
        VALUES (NEW.ProductID, v_old_qty, v_new_qty);
    END IF;
END $$

DELIMITER ;
```

Figure 3.5: CREATE TRIGGER orders_AFTER_UPDATE



```
UPDATE orders
SET
    Status = 'Done',
    Real_EndDate = CURDATE()
WHERE OrderID = '00195';

SELECT * FROM logs;
```

Result Grid

LogID	ProductID	OldQty	NewQty	UpdatedAt
1	P023	16	32	2025-05-26 09:44:52
NULL	NULL	NULL	NULL	NULL

Figure 3.6: Practical testing of TRIGGER

The next part is the practical testing of the trigger by executing an update query on the orders table.

Specifically, it updates the status of the order with OrderID 'O0195' to 'Done' and sets the actual end date to the current date, signaling the completion of the order. Following this update, a select query retrieves all records from the logs table to verify whether the trigger successfully captured the stock change event.

The output shows a log entry corresponding to the updated order, displaying the product identifier alongside the previous and new stock quantities, as well as the timestamp of the change. This confirms that the trigger operates as expected, accurately updating inventory levels and maintaining a detailed audit trail for stock modifications triggered by production order completions.

3.3 User Defined Functions: Calculate material cost per product unit

For more information, visit User Defined Functions

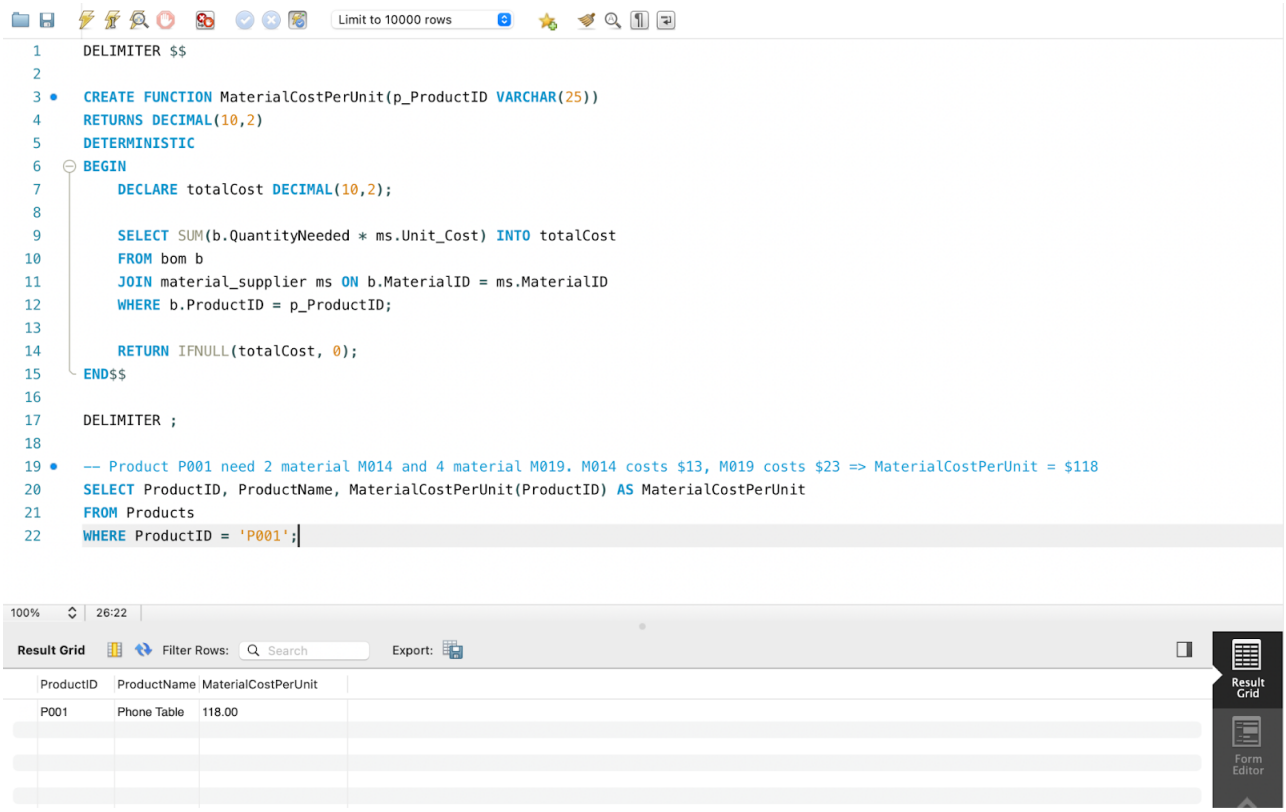


Figure 3.7: CREATE FUNCTION MaterialCostPerUnit

This user-defined function calculates the total material cost required to produce one unit of a specified product. It takes a ProductID as input and performs the following steps:

- It joins the Bill of Materials (bom) with the material_supplier table to get the unit cost of each material used in the product.
- It calculates the sum of the product of the quantity needed (QuantityNeeded) and the unit cost (Unit_Cost) for all materials associated with the given product.
- If no materials are found, it returns 0 to avoid null results.

Sample Query Using the Function:

SELECT ProductID, ProductName, MaterialCostPerUnit(ProductID) AS MaterialCostPerUnit
FROM Products
WHERE ProductID = 'P001';

This query demonstrates the usage of the MaterialCostPerUnit function by calculating the material cost per unit for the product with ID 'P001'. For example, if product 'P001' requires

2 units of material 'M014' costing $13 each and 4 units of material 'M019' costing $23 each, the function will compute the total material cost as:

(2 * 13) + (4 * 23) = $118 The query returns the product ID, the product name, and the calculated material cost per unit, allowing cost analysis for production planning and pricing decisions.

3.4 INDEX and Optimization Query

for more information, please visit Index code

This section demonstrates the use of indexes along with EXPLAIN and EXPLAIN ANALYZE to simulate optimization techniques aimed at improving query performance on large-scale operational data.

The specific objective of the following queries is to improve the efficiency of searching production orders by indexing relevant columns.

First, inspect the existing indexes on the Order table.

Key_name	Column_name	Non_unique	Cardinality	Index_type
PRIMARY	OrderID	0	200	BTREE
OrderID_UNIQUE	OrderID	0	200	BTREE
CategoryID_idx	CategoryID	1	5	BTREE
PlantID_idx	PlantID	1	5	BTREE
ProductID_idx	ProductID	1	38	BTREE

Table 3.1: Index_Order_Table

- The OrderID column is the primary key of the table, thus it has a PRIMARY index and a UNIQUE index (OrderID_UNIQUE) to enforce the uniqueness constraint.
- The columns CategoryID, PlantID, and ProductID are foreign keys referencing other tables in the database. MySQL automatically creates corresponding indexes (_idx) for these columns to enhance performance when executing JOIN operations or filtering queries by these attributes.

3.4.1 Use Case 1: Single-column Index on a Simple WHERE Condition

This query filters data from the orders table based on the Status field, aiming to retrieve orders that are currently marked as 'In Progress'.

Create an index on the Status column to support this query

Use EXPLAIN and EXPLAIN ANALYZE to compare the query execution plan before and after adding the index.

```sql
# UC1: Index đơn giản trên 1 điều kiện trong WHERE
SELECT *
FROM orders
WHERE Status = 'In Progress';
```

Figure 3.8: Single-column Index on a Simple WHERE Condition

```sql
CREATE INDEX idx_orders_status ON orders(Status);
```

Figure 3.9: CREATE INDEX

- Situation without INDEX

select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered
SIMPLE	orders	NULL	ALL	NULL	NULL	NULL	NULL	201	10.00

Figure 3.10: Explain without INDEX

```
-> Filter: (orders.`Status` = 'In Progress')  (cost=20.4 rows=20.1) (actual time=0.221..0.226 rows=4 loops=1)
    -> Table scan on orders  (cost=20.4 rows=201) (actual time=0.076..0.209 rows=201 loops=1)
```

Figure 3.11: Analyze without INDEX

- Situation with INDEX on status (idx_orders_status)

select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered
SIMPLE	orders	NULL	ref	idx_orders_status	idx_orders_status	83	const	4	100.00

Figure 3.12: Explain with INDEX

```
-> Index lookup on orders using idx_orders_status (Status='In Progress')  (cost=1.15 rows=4) (actual
time=0.124..0.129 rows=4 loops=1)
```

Figure 3.13: Analyze with INDEX

EXPLAIN shows a clear difference between having and not having an index on the Status column.

- When no index is present, MySQL performs a full table scan (type = ALL, rows = 201, filtered = 10.00), resulting in low efficiency.
- With the index idx_orders_status, MySQL switches to using ref access (since the WHERE clause uses Equality Condition '='), examining only the relevant rows (rows = 4, filtered = 100.00), making the query significantly more efficient.

EXPLAIN ANALYZE provides actual execution details.

- In the no-index case, MySQL scans the entire table and filters rows afterward, with a high cost (cost = 20.4) and longer duration.
- When the index is present, the system performs a direct index lookup, which is faster and more efficient (cost = 1.15, rows = 4).

3.4.2 Use Case 2: Composite Index with WHERE Clause Containing Two or More Conditions

The query below checks for completed orders of a specific product. In particular, it retrieves orders that are completed (Status = 'Done') for the product with ID P025 from the orders table.

```
# UC2: Composite index và phân tích Leftmost Prefix: Kiểm tra các đơn hàng đã hoàn thành của sản phẩm cụ thể
SELECT *
FROM orders
WHERE ProductID = 'P025'
  AND Status = 'Done';
```

Figure 3.14: Use Case 2

Looking at table Index_Order_Table, we can see that ProductID is a foreign key and already has an index. However, since the WHERE clause also includes an equality condition on the Status column, we will compare the performance between using only the foreign key index and a composite index on both ProductID and Status.

Create a composite index on the ProductID and Status columns:

```
CREATE INDEX idx_orders_product_status ON orders(ProductID, Status);
```

Figure 3.15: CREATE INDEX orders product status

Using Index from Foreign Key (ProductID_idx)

EXPLAIN shows the difference between using a single-column index (ProductID_idx) and a composite index (idx_orders_product_status).

- With the single-column index, MySQL uses the ref access type to search by ProductID but still needs to filter by Status (filtered = 10.00).
- In contrast, the composite index enables the query to filter by both ProductID and Status directly within the index, with filtered = 100.00 and fewer rows read (rows = 13 vs. 15), demonstrating better logic and optimization potential.

EXPLAIN ANALYZE reflects the actual execution process.

- For the single-column index, the engine must first search by ProductID and then filter by Status, consuming more resources (cost = 0.9, rows = 15, time 0.07s).

	Using Index from Foreign Key (ProductID_idx)
EXPLAIN	select_type: SIMPLE; table: orders; partitions: NULL; type: ref; possible_keys: ProductID_idx; key: ProductID_idx; key_len: 102; ref: const; rows: 15; filtered: 10.00
EXPLAIN ANALYZE	-> Filter: (orders.`Status` = 'Done') (cost=0.9 rows=1.5) (actual time=0.063..0.0762 rows=13 loops=1) -> Index lookup on orders using ProductID_idx (ProductID='P025') (cost=0.9 rows=15) (actual time=0.0589..0.0719 rows=15 loops=1)

	Composite index (idx_orders_product_status)
EXPLAIN	select_type: SIMPLE; table: orders; partitions: NULL; type: ref; possible_keys: ProductID_idx,idx_orders_product_status; key: idx_orders_product_status; key_len: 185; ref: const,const; rows: 13; filtered: 100.00
EXPLAIN ANALYZE	-> Index lookup on orders using idx_orders_product_status (ProductID='P025', Status='Done') (cost=2.05 rows=13) (actual time=0.0967..0.108 rows=13 loops=1)

Figure 3.16: index

- With the composite index, the engine can directly retrieve data that matches both conditions (cost = 2.05, rows = 13, time 0.10s).

Although the cost appears higher, this is due to the small dataset (200 rows). The composite index is expected to perform more efficiently when working with larger datasets.

Leftmost Prefix in Composite Index

With the composite index idx_orders_product_status (ProductID, Status):

- Query 1 (WHERE ProductID = 'P025'): This query can take advantage of the composite index because the filtering condition uses the first column of the index. According to the leftmost prefix rule, if a query filters using the first column (or a continuous sequence starting from the first column), then the index will be used

```
EXPLAIN
SELECT *
FROM orders
WHERE ProductID = 'P025'; -- Có thể dùng Composite Index
```

Figure 3.17: Composite index is usable

select_type	table	partitions	type	possible_keys
SIMPLE	orders	NULL	ref	ProductID_idx,idx_orders_product_status

Figure 3.18: Composite index

- In contrast, Query 2 (WHERE Status = 'In Progress') cannot use the composite index because the filtering condition does not include the first column (ProductID).

```
EXPLAIN
SELECT *
FROM orders
WHERE Status = 'In Progress';  -- Không dùng được Composite Index
```

Figure 3.19: Composite index is not usable

select_type	table	partitions	type	possible_keys
SIMPLE	orders	NULL	ALL	NULL

Figure 3.20: EXPLAIN query: Composite index is not usable

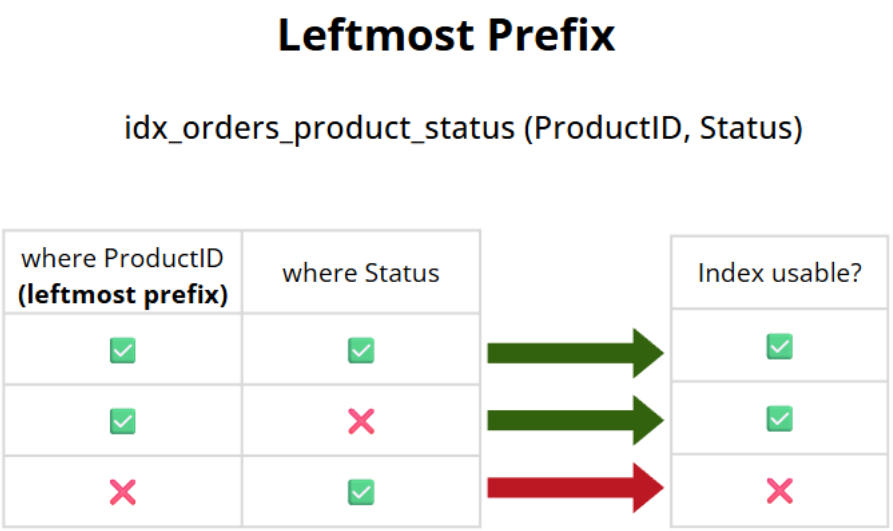


Figure 3.21: Leftmost Prefix

Even though Status is part of the composite index, since it is not used in the left-to-right order, the index is not applied and MySQL must perform a full table scan (type = ALL).

3.4.3 Use Case 3: Covering Index for SELECT + WHERE – Optimize query to avoid accessing base table

This query filters data by a time range on the StartDate column and only retrieves the OrderID and Status columns:

A composite index can be used not only for the WHERE clause but also for the SELECT clause. This section compares using a single-column index for the WHERE condition versus a composite index that covers all columns used in both WHERE and SELECT.

```sql
# UC3: Covering Index cho SELECT + WHERE - Tối ưu truy vấn để không cần đọc bảng gốc
SELECT OrderID, Status
FROM orders
WHERE StartDate >= '2025-05-01' AND StartDate < '2025-06-01';
```

Figure 3.22: Use case 3

```sql
CREATE INDEX idx_orders_startdate ON orders(StartDate);
```

Figure 3.23: Create index on StartDate column

- Create a single-column index on StartDate:

- Create a composite covering index on StartDate, OrderID, Status:

```sql
CREATE INDEX idx_orders_startdate_covering
ON orders(StartDate, OrderID, Status);
```

Figure 3.24: Create covering index

select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered
SIMPLE	orders	NULL	ref	ProductID_idx,idx_orders_product_status	idx_orders_product_status	185	const,const	13	100.00

Figure 3.25: Explain Composite index

	idx_orders_startdate_covering
EXPLAIN	select_type: SIMPLE; table: orders; partitions: NULL; type: range; possible_keys: idx_orders_startdate_covering; key: idx_orders_startdate_covering; key_len: 6; ref: NULL; rows: 20; filtered: 100.00
EXPLAIN ANALYZE	-> Filter: ((orders.StartDate >= TIMESTAMP'2025-05-01 00:00:00') and (orders.StartDate < TIMESTAMP'2025-06-01 00:00:00')) (cost=4.43 rows=20) (actual time=0.0384..0.0557 rows=20 loops=1) -> Covering index range scan on orders using idx_orders_startdate_covering over ('2025-05-01 00:00:00' <= StartDate < '2025-06-01 00:00:00') (cost=4.43 rows=20) (actual time=0.0351..0.0489 rows=20 loops=1)

EXPLAIN shows that both indexes, idx_orders_startdate and idx_orders_startdate_covering, are effectively used with the range access type to filter by the time range on the StartDate column. However, the idx_orders_startdate_covering is a covering index, which includes all columns used in the SELECT clause. This allows the database engine to retrieve data directly from the index without having to access the base table.

EXPLAIN ANALYZE confirms this advantage in practice.

- With the regular index, MySQL still needs to access the base table to fetch values of columns not present in the index, resulting in a slightly longer access time (actual time $\tilde{0}.0562 - 0.0794$).

	idx_orders_startdate
EXPLAIN	select_type: SIMPLE; table: orders; partitions: NULL; type: range; possible_keys: idx_orders_startdate; key: idx_orders_startdate; key_len: 6; ref: NULL; rows: 20; filtered: 100.00
EXPLAIN ANALYZE	-> Index range scan on orders using idx_orders_startdate over ('2025-05-01 00:00:00' <= StartDate < '2025-06-01 00:00:00'), with index condition: ((orders.StartDate >= TIMESTAMP'2025-05-01 00:00:00') and (orders.StartDate < TIMESTAMP'2025-06-01 00:00:00')) (cost=9.26 rows=20) (actual time=0.0562..0.0794 rows=20 loops=1)

- In contrast, with the covering index, all required data is already within the index, allowing the query to complete faster (actual time 0̃.0351 – 0.0489).

The difference is minor due to the dataset having only 20 rows, but with a larger dataset, a covering index would significantly improve performance.

3.4.4 Use case 4: Index in JOIN

This is a JOIN query that does not use an index.

```
# UC4: Index trong join
EXPLAIN ANALYZE
SELECT o.OrderID, o.Status, p.PlantName
FROM orders o
JOIN plants p
  ON LOWER(o.PlantID) = LOWER(p.PlantID);
```

Figure 3.26: EXPLAIN ANALYZE: Index in JOIN 1

EXPLAIN ANALYZE result:

```
-> Inner hash join (lower(o.PlantID) = lower(p.PlantID))  (cost=102 rows=1005) (actual time=0.0789..0.237
rows=201 loops=1)
    -> Table scan on o  (cost=4.07 rows=201) (actual time=0.0149..0.0856 rows=201 loops=1)
    -> Hash
        -> Table scan on p  (cost=0.75 rows=5) (actual time=0.0355..0.0415 rows=5 loops=1)
```

Figure 3.27: QUERY PLAN: Hash join

This is a JOIN that uses an index.

EXPLAIN ANALYZE result

Both queries perform a JOIN between the orders and plants tables, but their performance differs due to how the JOIN condition is written, which affects index usage.

```
EXPLAIN ANALYZE
SELECT o.OrderID, o.Status, p.PlantName
FROM orders o
JOIN plants p
  ON o.PlantID = p.PlantID;
```

Figure 3.28: EXPLAIN ANALYZE: Index in JOIN 2

```
-> Nested loop inner join  (cost=24.6 rows=201) (actual time=0.183..0.513 rows=201 loops=1)
   -> Table scan on p  (cost=0.75 rows=5) (actual time=0.0377..0.0394 rows=5 loops=1)
   -> Index lookup on o using PlantID_idx (PlantID=p.PlantID)  (cost=1.55 rows=40.2) (actual time=0.0428..0.0919
rows=40.2 loops=5)
```

Figure 3.29: QUERY PLAN: Nested loop join

Query 1 uses the condition LOWER(o.PlantID) = LOWER(p.PlantID). Because both columns are wrapped in the LOWER() function, MySQL cannot use the index on PlantID. As a result, the system performs a hash join, where:

- MySQL performs a full table scan on the plants table to build a hash table from PlantID values.
- Then, it scans the entire orders table, comparing each row against the hash table to find matches.

Query 2 uses the condition o.PlantID = p.PlantID, which allows MySQL to use the index PlantID_idx. The system performs a nested loop join, which works as follows:

- For each row in the outer table (plants), MySQL loops through the inner table (orders) by looking up matching rows using the index.
- This method is particularly efficient when the inner table has an appropriate index and the outer table is small.
- The query executes with significantly lower cost (cost = 24.6).

Although the nested loop join has a lower cost, its actual execution time is slightly higher compared to the hash join in this case. However, in larger datasets, the nested loop join combined with indexes will scale better and significantly improve performance compared to the hash join.

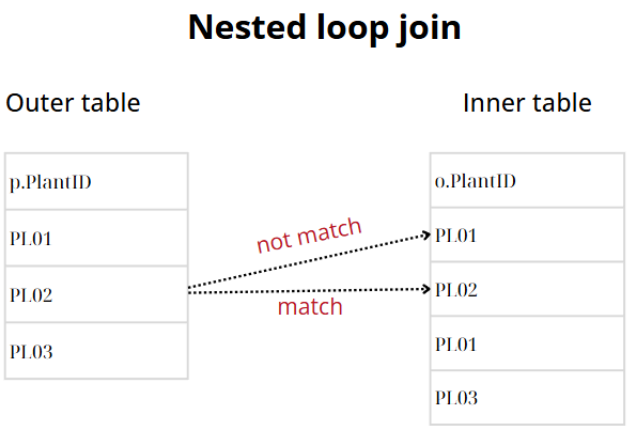


Figure 3.30: Nested loop join

3.5 Stored Procedures

3.5.1 Inventory updates:

```
DELIMITER //

CREATE PROCEDURE UpdateMaterialInventory (
    IN in_MaterialID VARCHAR(25),
    IN in_ChangeAmount INT,
    IN in_Action VARCHAR(10) CHARACTER SET utf8mb4 COLLATE utf8mb4_general_ci -- 'IN' or 'OUT'
)
BEGIN
    DECLARE currentQty INT;

    -- Get current inventory quantity
    SELECT Quantity INTO currentQty
    FROM materials #MAYBE PHAI SUA TEN BANG
    WHERE MaterialID = in_MaterialID;
```

Figure 3.31: Procedure UpdateMaterialInventory 1

for more information, please visit Stored procedures code

The procedure UpdateMaterialInventory allows updating material inventory through two actions: stock in (IN) or stock out (OUT). When called, the procedure retrieves the current inventory quantity of a material based on its MaterialID.

If the action is OUT, the system checks whether the requested quantity exceeds the available stock:

- If it does, an error is returned.
- If not, the quantity is deducted from inventory.

If the action is IN, the system adds the given quantity to the current inventory.

If an invalid action is provided (not IN or OUT), the procedure returns an error message.

```
-- If the action is stock out
IF UPPER(in_Action) = 'OUT' THEN
    IF in_ChangeAmount > currentQty THEN
        SELECT CONCAT('Error: Not enough stock. Available: ', currentQty, ', Requested: ', in_ChangeAmount) AS ErrorMessage;
    ELSE
        UPDATE materials #MAYBE PHAI SUA TEN BANG
        SET Quantity = Quantity - in_ChangeAmount
        WHERE MaterialID = in_MaterialID;
        SELECT CONCAT('Stock out successful: -', in_ChangeAmount, ' | Material ID: ', in_MaterialID) AS Result;
    END IF;

-- If the action is stock in
ELSEIF UPPER(in_Action) = 'IN' THEN
    UPDATE materials #MAYBE PHAI SUA TEN BANG
    SET Quantity = Quantity + in_ChangeAmount
    WHERE MaterialID = in_MaterialID;
    SELECT CONCAT('Stock in successful: +', in_ChangeAmount, ' | Material ID: ', in_MaterialID) AS Result;
```

Figure 3.32: Procedure UpdateMaterialInventory 2

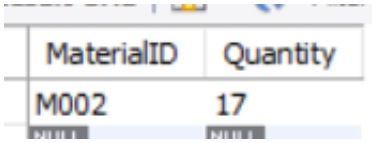
```
        -- Invalid action
        ELSE
            SELECT 'Error: Invalid action. Only IN or OUT is allowed.' AS ErrorMessage;
        END IF;
    END //

    DELIMITER ;
```

Figure 3.33: Procedure UpdateMaterialInventory 3

This procedure helps maintain real-time inventory control and ensures the validity of warehouse transactions.

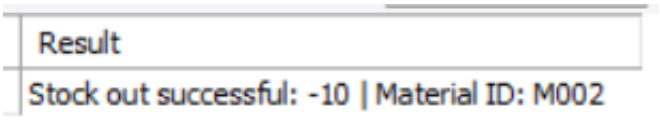
Case 1: Stock-in/Stock-out Successful Inventory before stock-out:

for more information, please visit Test procedures code



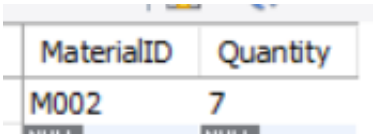
MaterialID	Quantity
M002	17

Run the procedure UpdateMaterialInventory to stock out 10 units of material M002.



Result
Stock out successful: -10 | Material ID: M002

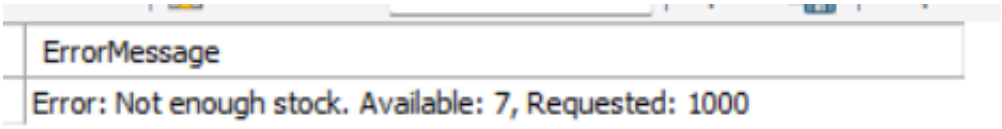
Inventory after stock-out:



MaterialID	Quantity
M002	7

Case 2: Stock-out Failure Due to Insufficient Inventory Run the procedure UpdateMaterialInventory to stock out 1000 units of material M002

for more information, please visit Test procedures code



ErrorMessage
Error: Not enough stock. Available: 7, Requested: 1000

The result shows that the operation failed due to insufficient inventory.

3.5.2 Automate production order creation include inventory update:

```
DELIMITER //

CREATE PROCEDURE CreateProductionOrder (
    IN in_OrderID VARCHAR(25),
    IN in_ProductID VARCHAR(25),
    IN in_Quantity INT,
    IN in_CategoryID VARCHAR(25),
    IN in_PlantID VARCHAR(25),
    IN in_StartDate DATETIME,
    IN in_ExpectedEndDate DATETIME
)
BEGIN
    DECLARE material_shortage TEXT;
    proc_end: BEGIN

    -- Step 1: Check if any material is insufficient
    SELECT GROUP_CONCAT(CONCAT(b.MaterialID, ' (short ', b.QuantityNeeded * in_Quantity - m.Quantity, ')'))
    INTO material_shortage
    FROM bom b
    JOIN materials m ON b.MaterialID = m.MaterialID
    WHERE b.ProductID = in_ProductID
      AND b.QuantityNeeded * in_Quantity > m.Quantity;
```

Figure 3.34: Store procedure: CreateProductionOrder 1

for more information, please visit Stored procedures code

```
    -- Step 2: If material is insufficient, stop
    IF material_shortage IS NOT NULL THEN
        SELECT CONCAT('Error: Insufficient materials: ', material_shortage) AS ErrorMessage;
        LEAVE proc_end;
    END IF;

    -- Step 3: Create production order
    INSERT INTO orders (
        OrderID, ProductID, Quantity, CategoryID, PlantID,
        StartDate, Expected_EndDate, Status
    ) VALUES (
        in_OrderID, in_ProductID, in_Quantity, in_CategoryID, in_PlantID,
        in_StartDate, in_ExpectedEndDate, 'Planned'
    );

    -- Step 4: Deduct inventory
    UPDATE materials m
    JOIN bom b ON m.MaterialID = b.MaterialID
    SET m.Quantity = m.Quantity - (b.QuantityNeeded * in_Quantity)
    WHERE b.ProductID = in_ProductID;

    SELECT CONCAT('Production Order created: ', in_OrderID) AS Result;

    END proc_end;
END //

DELIMITER ;
```

Figure 3.35: Store procedure: CreateProductionOrder 2

The procedure CreateProductionOrder is designed to automate the creation of production orders while strictly monitoring the availability of materials.It takes input such as product ID, required quantity, product type, manufacturing plant, and the estimated start and end dates.

Based on the bill of materials (bom), the system calculates the total material requirements for the order and compares them against the actual inventory in the materials table.

- If any material is insufficient, the procedure stops and returns an error listing the missing items.
- If all requirements are met, the system creates a new record in the orders table and simultaneously updates inventory by deducting the used quantities.

This mechanism ensures that only feasible production orders are created, thereby improving accuracy and efficiency in production planning and control.

Case 1: Sufficient Inventory - Create a new order to produce 2 units of product 'P003'

for more information, please visit Test procedures code 2

Check total required materials to produce 2 units of 'P003'

MaterialID	Required
M015	14.00
M021	6.00

MaterialID	Inventory
M015	57
M021	20

Check current inventory for materials used in product 'P003'

=> Inventory is sufficient.

After running the procedure CreateProductionOrder, the result returned:

Result
Production Order created: O6667

Since the procedure automatically updates the inventory after creating the order, the inventory changes:

MaterialID	Inventory_After
M015	43
M021	14

Case 2: Insufficient Inventory - Create a new order to produce 100 units of product 'P003'.

for more information, please visit Test procedures code 2

Check total required materials to produce 100 units of 'P003'

MaterialID	Required
M015	700.00
M021	300.00

Compared to the previous inventory, it is clear that the materials are insufficient.

- After running the procedure CreateProductionOrder, the system returns an error message specifying the shortage amount for each material

ErrorMessage
Error: Insufficient materials: M015 (short 657.00),M021 (short 286.00)

4 Database Security and Administration

4.1 Create roles for production managers, warehouse staff, and finance

for more information, please visit Create roles code

Create new users for production managers, warehouse staff, and finance staff by using the standard SQL CREATE USER statement.

```
CREATE USER 'Production_Manager'@'localhost' IDENTIFIED BY 'PM';
CREATE USER 'Warehouse_Staff'@'localhost' IDENTIFIED BY 'WS';
CREATE USER 'Finance_Staff'@'localhost' IDENTIFIED BY 'FS';
```

Figure 4.1: Create user

The database system implements role-based access control to ensure operational security and proper segregation of duties, by using the standard SQL GRANT statement.

Three primary user roles:

- Production manager (Production_Manager): Holds full privileges over all database objects and operations, because this user oversees the entire production process. They require unrestricted access to manage materials, update production data, monitor warehouse inventory, and handle supplier and order information. Full control ensures flexibility in decision-making and immediate responsiveness to operational needs. This level of access is essential for maintaining seamless coordination across all departments involved in production.

```
GRANT ALL PRIVILEGES ON database_final_DBMS.* TO 'Production_Manager'@'localhost';
```

Figure 4.2: Grant privileges

- Warehouse Staff (Warehouse_Staff): grant permissions to the user Warehouse_Staff with the following privileges:

  SELECT, INSERT, and UPDATE on the tables: materials, material_supplier, products, suppliers, suppliers_deli. These privileges allow staff to manage inventory records, update material receipts, and track supplier deliveries.

  Only SELECT on the tables: bom, categories, orders, plants; to retrieve reference information without altering sensitive production or planning data. This structure maintains operational efficiency while minimizing risks of unauthorized data modifications.

```
GRANT SELECT, INSERT, UPDATE ON database_final_DBMS.materials TO 'Warehouse_Staff'@'localhost';
GRANT SELECT, INSERT, UPDATE ON database_final_DBMS.material_supplier TO 'Warehouse_Staff'@'localhost';
GRANT SELECT, INSERT, UPDATE ON database_final_DBMS.products TO 'Warehouse_Staff'@'localhost';
GRANT SELECT, INSERT, UPDATE ON database_final_DBMS.suppliers TO 'Warehouse_Staff'@'localhost';
GRANT SELECT, INSERT, UPDATE ON database_final_DBMS.suppliers_deli TO 'Warehouse_Staff'@'localhost';

GRANT SELECT ON database_final_DBMS.bom TO 'Warehouse_Staff'@'localhost';
GRANT SELECT ON database_final_DBMS.categories TO 'Warehouse_Staff'@'localhost';
GRANT SELECT ON database_final_DBMS.orders TO 'Warehouse_Staff'@'localhost';
GRANT SELECT ON database_final_DBMS.plants TO 'Warehouse_Staff'@'localhost';
```

Figure 4.3: Grant permission

Finance Staff (Finance Staff): grant permissions to the user Finance_Staff with the following privileges:

for more information, please visit Finance staffs code

- SELECT, INSERT, and UPDATE on the tables: material_supplier, products, supplier_deli, to support cost tracking, financial reporting, and budget management. These tables contain key financial data related to materials, product pricing, and delivery records.
- Only SELECT on the tables: bom, categories, materials, orders, plants, suppliers, to reference production-related data when needed. This ensures they can perform financial duties effectively without interfering with operational processes.

```
GRANT SELECT, INSERT, UPDATE ON database_final_DBMS.material_supplier TO 'Finance_Staff'@'localhost';
GRANT SELECT, INSERT, UPDATE ON database_final_DBMS.products TO 'Finance_Staff'@'localhost';
GRANT SELECT, INSERT, UPDATE ON database_final_DBMS.supplier_deli TO 'Finance_Staff'@'localhost';

GRANT SELECT ON database_final_DBMS.bom TO 'Finance_Staff'@'localhost';
GRANT SELECT ON database_final_DBMS.categories TO 'Finance_Staff'@'localhost';
GRANT SELECT ON database_final_DBMS.materials TO 'Finance_Staff'@'localhost';
GRANT SELECT ON database_final_DBMS.orders TO 'Finance_Staff'@'localhost';
GRANT SELECT ON database_final_DBMS.plants TO 'Finance_Staff'@'localhost';
GRANT SELECT ON database_final_DBMS.suppliers TO 'Finance_Staff'@'localhost';
```

Privilege Activation and Use of FLUSH PRIVILEGES: Once privileges are granted using the GRANT statement, MySQL updates its in memory privilege tables. To ensure changes apply consistently—especially when modified by scripts—this command is used

```
FLUSH PRIVILEGES;
```

This command reloads privilege data from disk for immediate enforcement. While not always mandatory after GRANT, it is best practice to maintain session consistency. This approach strengthens database security and enforces role-based access to inventory and supplier data.

4.2 Protect sensitive supplier and pricing data through permissions and encryption

for more information, please visit Protect sensitive supplier code

Data fields that need to be secured:

- SupplierName, Address, PhoneNumber belong to table 'supplier'
- unit_cost, tot_cost, status belong to table 'supplier_deli'
- Unit_Cost belongs to table 'material_supplier'

Encryption: we use the standard SQL AES_ENCRYPT statement with a key.
An example: 'material_supplier' table

```
12 •  INSERT INTO material_supplier (MaterialID, MaterialName, SupplierID, Unit_Cost)
13    VALUES
14      ('M001', 'House Wood', 'S001', AES_ENCRYPT('22.00', 'my_secret_key')),
15      ('M001', 'House Wood', 'S002', AES_ENCRYPT('23.00', 'my_secret_key')),
16      ('M002', 'Score Wood', 'S001', AES_ENCRYPT('34.00', 'my_secret_key')),
17      ('M003', 'Off Wood', 'S002', AES_ENCRYPT('35.00', 'my_secret_key')),
18      ('M004', 'Blood Wood', 'S003', AES_ENCRYPT('43.00', 'my_secret_key')),
19      ('M005', 'Feel Wood', 'S004', AES_ENCRYPT('21.00', 'my_secret_key')),
20      ('M006', 'Share Wood', 'S005', AES_ENCRYPT('22.25', 'my_secret_key')),
21      ('M007', 'Bed Wood', 'S006', AES_ENCRYPT('33.00', 'my_secret_key')),
22      ('M008', 'Charge Wood', 'S007', AES_ENCRYPT('17.00', 'my_secret_key')),
23      ('M009', 'Relate Wood', 'S008', AES_ENCRYPT('12.00', 'my_secret_key')),
24      ('M010', 'Senior Wood', 'S008', AES_ENCRYPT('11.00', 'my_secret_key')),
25      ('M010', 'Senior Wood', 'S009', AES_ENCRYPT('12.00', 'my_secret_key')),
26      ('M011', 'Red Wood', 'S010', AES_ENCRYPT('12.00', 'my_secret_key')),
27      ('M012', 'Man Wood', 'S011', AES_ENCRYPT('14.00', 'my_secret_key')),
28      ('M013', 'Bar Wood', 'S012', AES_ENCRYPT('13.00', 'my_secret_key')),
29      ('M014', 'Seek Wood', 'S013', AES_ENCRYPT('13.00', 'my_secret_key')),
30      ('M015', 'Consider Wood', 'S014', AES_ENCRYPT('23.00', 'my_secret_key')),
31      ('M016', 'Place Wood', 'S015', AES_ENCRYPT('43.00', 'my_secret_key')),
32      ('M017', 'Out Wood', 'S016', AES_ENCRYPT('22.00', 'my_secret_key')),
33      ('M018', 'Campaign Wood', 'S016', AES_ENCRYPT('23.00', 'my_secret_key')),
34      ('M019', 'Oil Wood', 'S017', AES_ENCRYPT('23.00', 'my_secret_key')),
35      ('M020', 'Choice Wood', 'S017', AES_ENCRYPT('24.00', 'my_secret_key')),
```

Figure 4.4: Insert data

To see the secured fields, we use AES_DECRYPT statement with the key we set up before
With SELECT without AES_DECRYPT statement:

for more information, please visit Test code select

SELECT with AES_DECRYPT statement:

```
36      ('M021', 'Raise Wood', 'S018', AES_ENCRYPT('35.00', 'my_secret_key')),
37      ('M022', 'Work Wood', 'S018', AES_ENCRYPT('11.00', 'my_secret_key')),
38      ('M023', 'Join Wood', 'S018', AES_ENCRYPT('23.00', 'my_secret_key')),
39      ('M024', 'Will Wood', 'S019', AES_ENCRYPT('12.00', 'my_secret_key')),
40      ('M025', 'Report Wood', 'S019', AES_ENCRYPT('11.00', 'my_secret_key')),
41      ('M026', 'Employee Wood', 'S019', AES_ENCRYPT('23.00', 'my_secret_key')),
42      ('M027', 'They Wood', 'S019', AES_ENCRYPT('14.00', 'my_secret_key')),
43      ('M027', 'They Wood', 'S020', AES_ENCRYPT('15.00', 'my_secret_key')),
44      ('M028', 'Director Wood', 'S020', AES_ENCRYPT('25.00', 'my_secret_key')),
45      ('M029', 'Run Wood', 'S020', AES_ENCRYPT('21.00', 'my_secret_key')),
46      ('M030', 'Than Wood', 'S020', AES_ENCRYPT('28.00', 'my_secret_key'))
```

Figure 4.5: Insert data

```
select * from material_supplier;
```

MaterialID	MaterialName	SupplierID	Unit_Cost
M001	House Wood	S001	BLOB
M001	House Wood	S002	BLOB
M002	Score Wood	S001	BLOB
M003	Off Wood	S002	BLOB
M004	Blood Wood	S003	BLOB
M005	Feel Wood	S004	BLOB
M006	Share Wood	S005	BLOB
M007	Bed Wood	S006	BLOB
M008	Charge Wood	S007	BLOB
M009	Relate Wood	S008	BLOB
M010	Senior Wood	S008	BLOB
M010	Senior Wood	S009	BLOB
M011	Red Wood	S010	BLOB
M012	Man Wood	S011	BLOB
M013	Bar Wood	S012	BLOB
M014	Seek Wood	S013	BLOB
M015	Consider Wood	S014	BLOB
M016	Place Wood	S015	BLOB

Figure 4.6: Output

```
SELECT
MaterialID,
MaterialName,
SupplierID,
CONVERT(AES_DECRYPT(Unit_Cost, 'my_secret_key') USING utf8) AS Unit_Cost
FROM material_supplier;
```

Figure 4.7: SELECT with AES_DECRYPT statement

4.3 Develop backup and recovery plans

Develop backup:
Regular logical backups to maintain data integrity and operational resilience are performed using mysqldump, automated via scheduled tasks (e.g., cron jobs). Backups should be stored securely and verified regularly to ensure their usability in recovery scenarios.

Create backup script file (.bat) on Notepad

MaterialID	MaterialName	SupplierID	Unit_Cost
M001	House Wood	S001	22.00
M001	House Wood	S002	23.00
M002	Score Wood	S001	34.00
M003	Off Wood	S002	35.00
M004	Blood Wood	S003	43.00
M005	Feel Wood	S004	21.00
M006	Share Wood	S005	22.25
M007	Bed Wood	S006	33.00
M008	Charge Wood	S007	17.00
M009	Relate Wood	S008	12.00
M010	Senior Wood	S008	11.00
M010	Senior Wood	S009	12.00
M011	Red Wood	S010	12.00
M012	Man Wood	S011	14.00
M013	Bar Wood	S012	13.00
M014	Seek Wood	S013	13.00
M015	Consider Wood	S014	23.00
M016	Place Wood	S015	43.00

Figure 4.8: Output

for more information, please visit Develop backup file

```
set PATH_TO_MYSQL="C:\xampp\mysql\bin"
set BACKUP_DIR="C:\backup for sql"
set FILENAME=database_final_DBMS_backup_2025-05-26.sql
%PATH_TO_MYSQL%\mysqldump.exe -u root -pPW database_final_DBMS > %BACKUP_DIR%\%FILENAME%
echo Backup completed: %FILENAME%
pause
```

Figure 4.9: Create backup script file

- C:\xampp\mysql\bin the path to the folder containing the `mysqldump.exe` file on the computer.
- C:\backup for sql the path to the folder where the user wants to save the backup file.
- database_final_DBMS_backup_2025-05-26.sql: the database name and the date when the backup was created.
- PW: the user's password.

Run the script file on CMD

The backup file "database_final_DBMS_backup_2025-05-26.sql" will appear in the folder the user requested earlier.

```
C:\backup for sql>set PATH_TO_MYSQL="C:\xampp\mysql\bin"

C:\backup for sql>set BACKUP_DIR="C:\backup for sql"

C:\backup for sql>set FILENAME=database_final_DBMS_backup_2025-05-26.sql

C:\backup for sql>"C:\xampp\mysql\bin"\mysqldump.exe -u root -pTrang*0903 database_final_DBMS  1>"C:\backup for sql"\database_final_DBMS_backup_2025
-05-26.sql

C:\backup for sql>echo Backup completed: database_final_DBMS_backup_2025-05-26.sql
Backup completed: database_final_DBMS_backup_2025-05-26.sql
```

Figure 4.10: Run script file

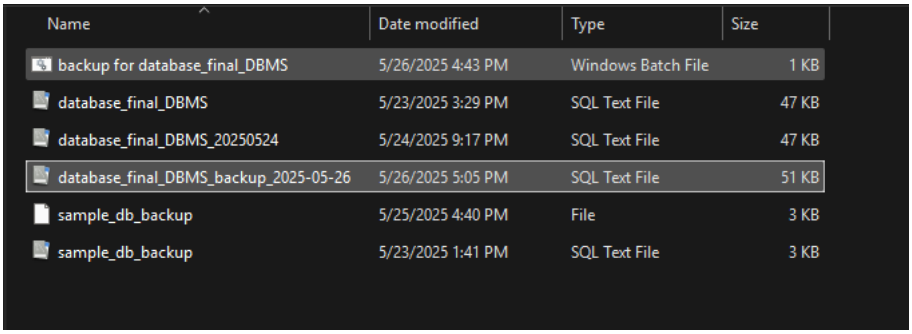


Figure 4.11: Backup files

Recovery plans: using Task Scheduler (on Window)
Open app Task Scheduler

Figure 4.12: Open app Task Scheduler

Click Create Basic Task

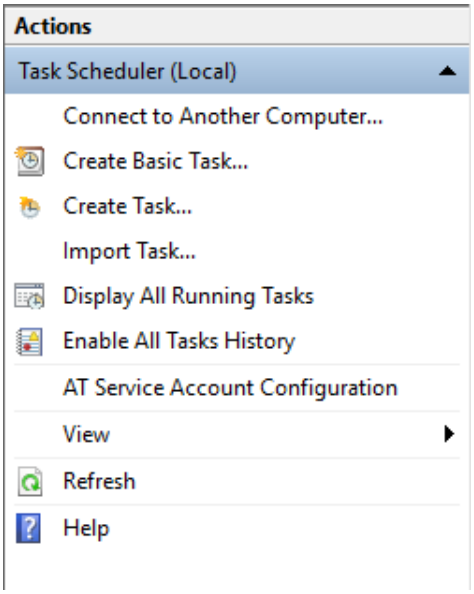


Figure 4.13: Click create basic task

Name the task ¯ click next

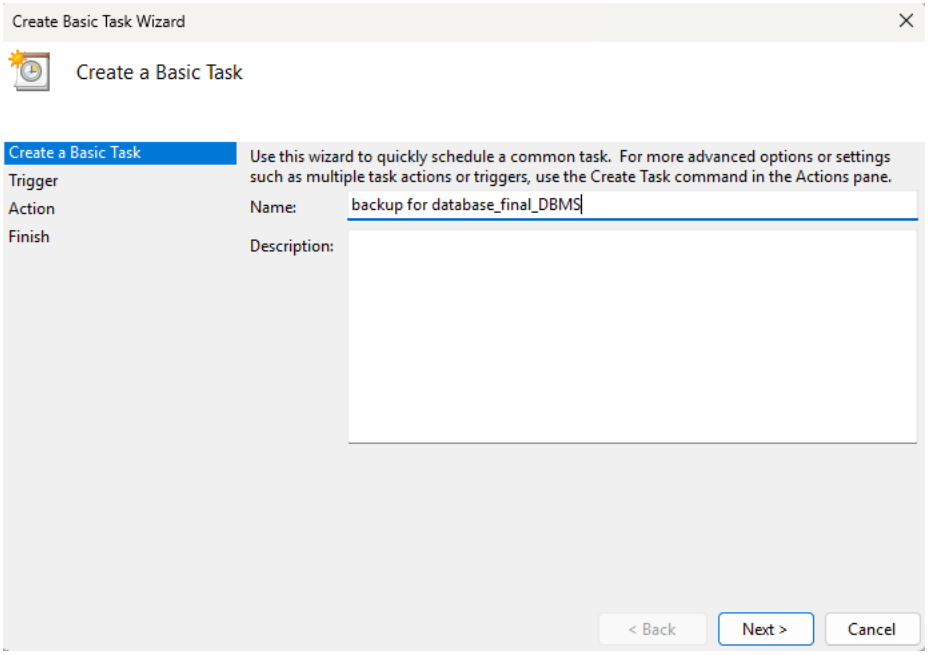


Figure 4.14: Name the task

Choose the time the user want to start (weekly) ¯ click next

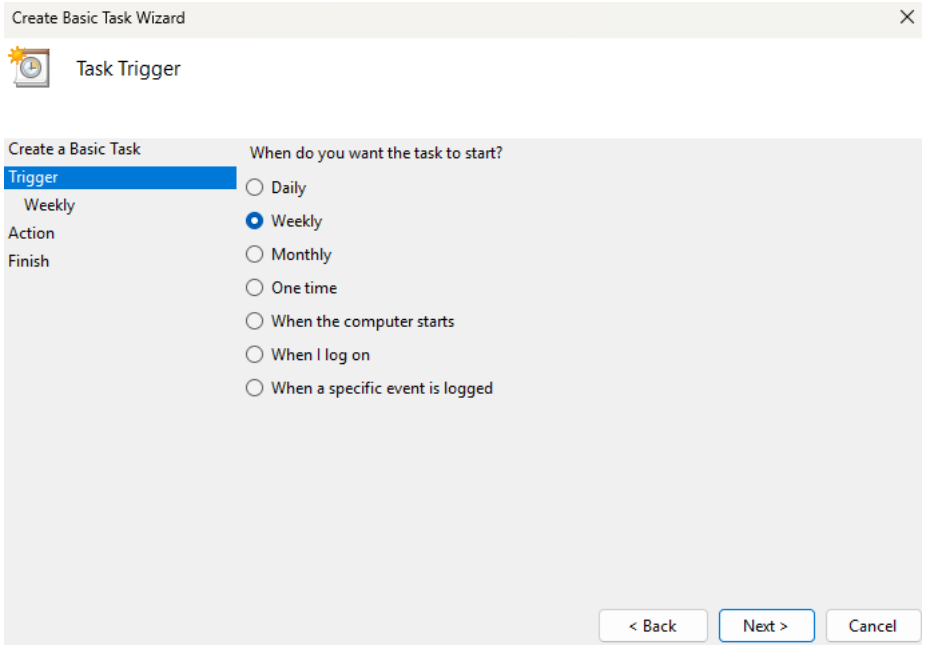


Figure 4.15: Choose time

Select the start date and time you want to run backup ¯ click next

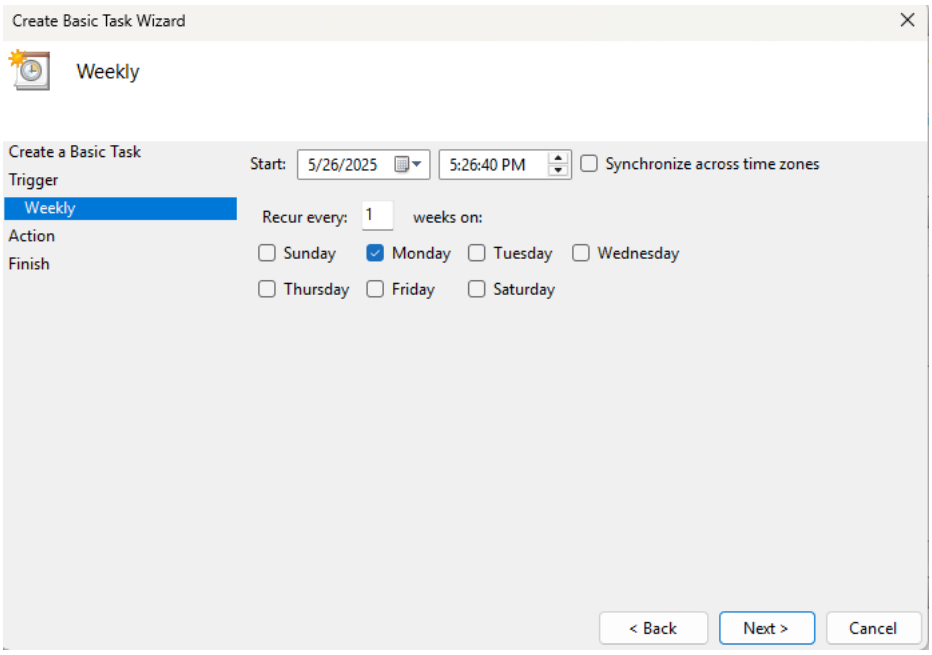


Figure 4.16: Select start date

In “Action”, click “Start a program” ¯ click next

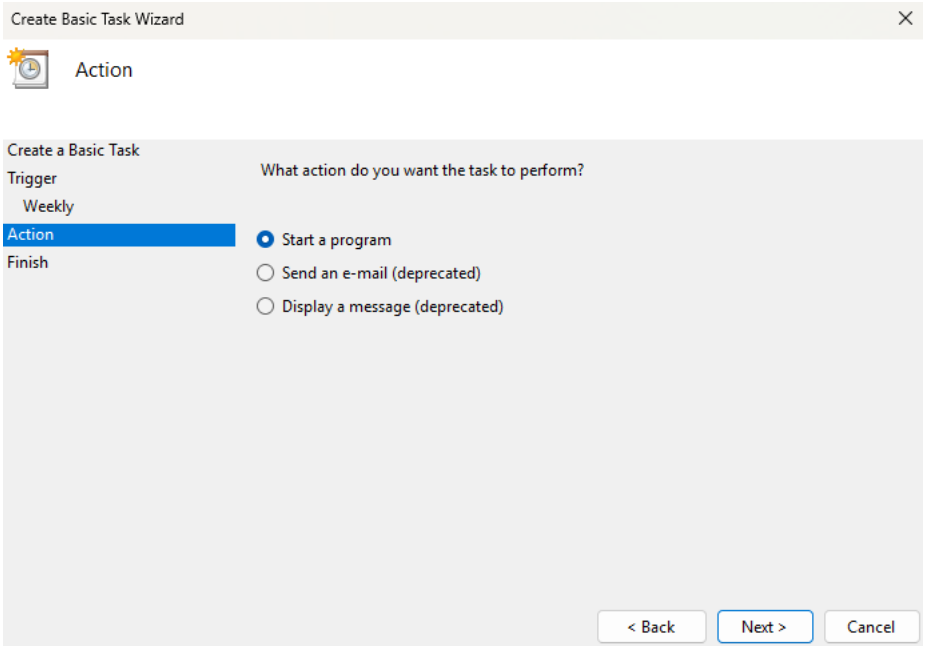


Figure 4.17: Start a program

Click Browse to choose the backup script file ¯ click next

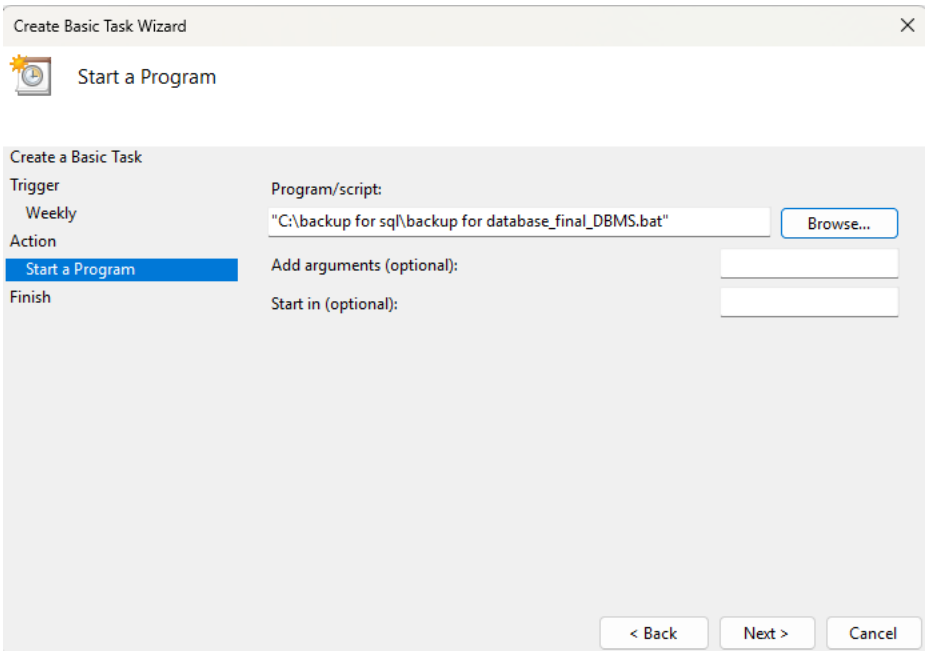


Figure 4.18: Click browse

Click Finish

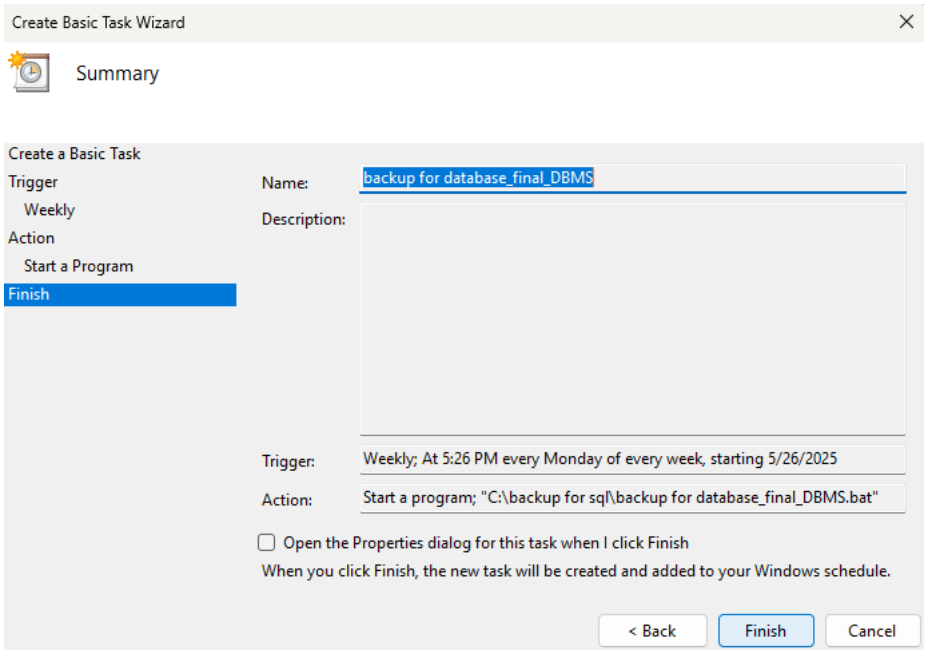


Figure 4.19: Click finish

Click “Task Scheduler Library” to see the new task

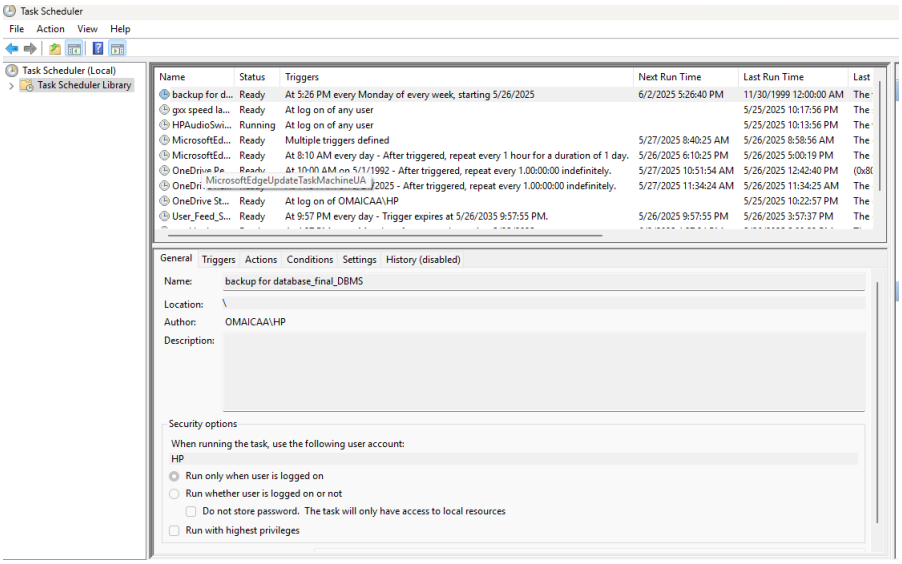


Figure 4.20: Click “Task Scheduler Library”

5 Python Application Development

5.1 Database connection

The use of mysql.connector for database connection plays a central role in integrating MySQL with Python-based applications. It enables seamless communication between Python scripts and the database, allowing developers to execute queries, retrieve data, and call stored procedures efficiently. This connection method is essential for the subsequent parts of the report, including Operational Scripts, Reporting Modules, and the User Interface

Standard structure of a MySQL database connection statement using mysql.connector, intended for calling a stored procedure.

```
import mysql.connector
```

Figure 5.1: Import mysql

```
def total_sale_employee(employee_id):
    try:
        # Connect to the MySQL database
        db_connection = mysql.connector.connect(
            host="127.0.0.1",         # MySQL server address (localhost)
            user="root",              # MySQL username
            password="*****",  # Replace with your actual password
            database="classicmodels")  # The name of the database to work with
        # Create a cursor object to execute SQL queries
        mycursor = db_connection.cursor()
        # Call the stored procedure 'total_sale_cal' with employee_id as the input parameter
        mycursor.callproc( procname: 'total_sale_cal', args: [employee_id])
        # Retrieve the result returned by the stored procedure
        output = []
        for result in mycursor.stored_results():
            output.append(result.fetchall())
        # Process the returned data
        if output[0]:
            print(f"Employee ID {employee_id}")
            print(f"Total sales {output[0][0][0]}")
        else:
            print(f"Employee ID {employee_id}")
            print(f"Total sales: 0")
```

Figure 5.2: Import mysql

The function [function_name](input_data) is defined to establish a connection to a MySQL database, call a stored procedure, and handle the returned results.

- Within the function, a try block is used to ensure safe execution. First, the connection to the database is established using mysql.connector.connect() with parameters such as host, user, password, and database.
- Next, a cursor object is created to perform the query.
- The function uses callproc() to invoke the corresponding stored procedure, passing in the required parameters.
- The result returned from the procedure is retrieved through stored_results() and stored in an output list. The returned data is then checked and processed based on the program's purpose - for example, displaying it on screen or storing it.

```
# Catch and print the error if any occurs during connection or execution
except mysql.connector.Error as err:
    print(f"Error: {err}")
# Ensure that the cursor and connection are properly closed
finally:
    if db_connection.is_connected():
        mycursor.close()
        db_connection.close()
```

Figure 5.3: Establish connection

- If an error occurs during the connection or execution process, the except block will catch it and print out the corresponding error message.
- Finally, the finally block ensures that both the connection and the cursor are closed if they are still open, helping to release resources and prevent connection leaks.

5.2 Operational Scripts

For more information, please visit Operational Scripts code

This section will use mysql.connector to call two stored procedures written in the MySQL database: UpdateMaterialInventory and CreateProductionOrder, and rerun the same test cases as illustrated earlier.

5.2.1 Inventory updates

The coding file will have the password section hidden

Case 1: Stock-in/stock-out successful

```
update_material_inventory( material_id: 'M002',  change_amount: 10,  action: 'OUT')
```

Returned result

Case 2: Stock-out failed due to insufficient inventory

Returned result

```
                                                  Message
0  Error: Not enough stock. Available: 17, Requested: 1000
```

```
update_material_inventory( material_id: 'M002',  change_amount: 1000,  action: 'OUT')
```

```
                                                  Message
0  Error: Not enough stock. Available: 17, Requested: 1000
```

5.2.2 Automate production order creation include inventory update:

The coding file will have the password section hidden

Case 1: Sufficient Inventory

```
create_production_order(
    order_id='O6600',
    product_id='P003',
    quantity=2,
    category_id='C004',
    plant_id='PL03',
    start_date=datetime.now(),
    expected_end_date=datetime.now() + timedelta(days=7))
```

Returned result

```
                          Message
0  Production Order created: O6600
```

Case 2: Insufficient Inventory

```
create_production_order(
    order_id='O6603',
    product_id='P003',
    quantity=1000,
    category_id='C004',
    plant_id='PL03',
    start_date=datetime.now(),
    expected_end_date=datetime.now() + timedelta(days=7)
)
```

Returned result

```
                                                                  Message
0  Error: Insufficient materials: M015 (short 6971.00),M021 (short 2992.00)
```

5.3 Reporting module

The reporting module is designed to generate a dashboard report that helps track monthly production, monitor stock quantities by category, analyze the distribution of material unit costs, and visualize order status distribution. All of these graphs are illustrated in the Dashboard report tab on the project's Graphical User Interface (GUI)

5.3.1 Monthly Completed Orders

Figure 5.4: Monthly completed order

Description:

This bar chart illustrates the number of production orders completed each month from November 2024 to May 2025, which was also the period that the project processed. The x-axis represents the months (formatted as YYYY-MM) while the y-axis indicates the number of completed orders.

Insights:

- The number of completed orders peaked in December 2024 and January 2025 with nearly 30 orders per month.
- There is a slight decline in subsequent months, though the production remains relatively stable from February to April 2025.
- A significant drop in May 2025 (around 15 orders) is noticeable, which might indicate either reduced demand, capacity issues. In fact, the month is not over yet so the reduction in completed orders is indicating incomplete data for that month.

Implication: This trend analysis helps production managers assess performance over time and identify months with potential inefficiencies or seasonal variations in output.

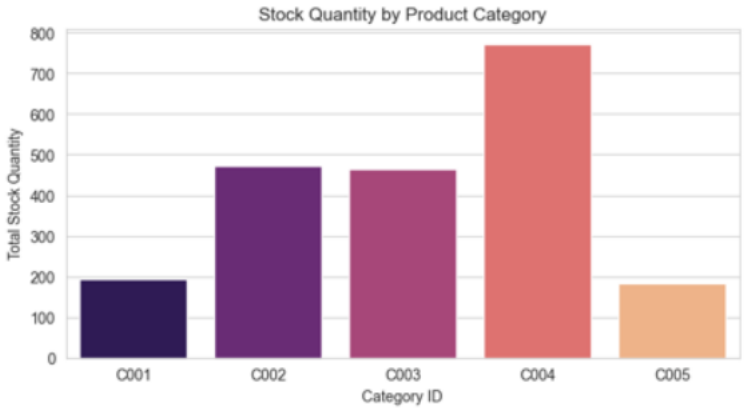


Figure 5.5: Stock Quantity by Product Category

5.3.2 Stock Quantity by Product Category

Description:

This bar chart displays the total stock quantity of products categorized by Category ID (C001 to C005). The x-axis lists the product categories, and the y-axis represents the corresponding total stock quantity.

Insights:

- Category C004 has the highest stock quantity (over 750 units), significantly more than other categories.
- Categories C002 and C003 have moderate and nearly equal stock levels (around 500 units).
- C001 and C005 have the lowest stock, with C005 being particularly low.

Implication: This chart aids inventory managers in identifying which categories may be overstocked (potential storage cost or low turnover) or understocked (risk of stockouts), allowing them to adjust procurement and production plans accordingly.

5.3.3 Material Unit Cost Distribution

Description: This boxplot presents the distribution of unit cost for raw materials. The y-axis shows the cost in dollars ($), while the boxplot itself summarizes the central tendency and variability.

Insights:

- The interquartile range (IQR) spans from approximately $14 to 26$, with a median around $22.
- There is one noticeable outlier above $40, which may represent a high-cost material that requires further analysis.

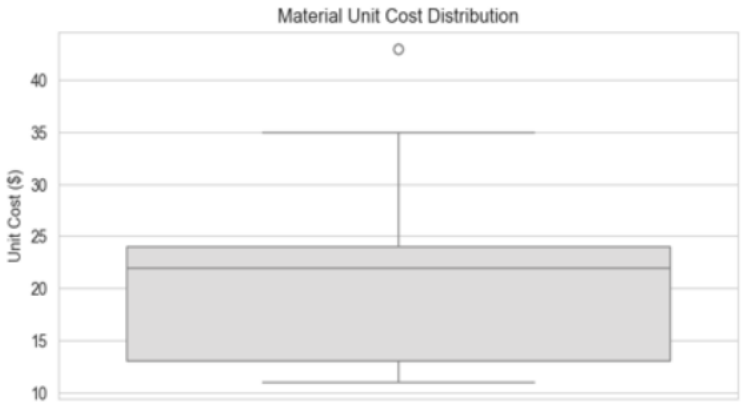


Figure 5.6: Material Unit Cost Distribution

- The relatively compact IQR indicates moderate variability in material costs, with most materials priced within a controlled range.

Implication: This visualization helps the procurement team monitor material cost fluctuations and detect any anomalies or exceptionally high-cost items that may need to be renegotiated or replaced.

5.3.4 Order Status Distribution

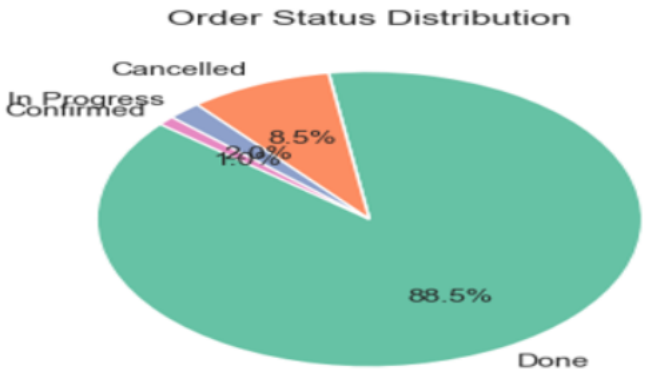


Figure 5.7: Order Status Distribution

Description: This pie chart shows the percentage distribution of orders across various statuses: Done, Cancelled, In Progress, and Confirmed.

Insights:

- A dominant 88.5% of all orders are marked as "Done", reflecting a high completion rate.
- 8.5% of orders have been Cancelled, and a small fraction are either In Progress (1.0%) or Confirmed (2.0%).

- The minimal portion of pending orders may suggest efficient order processing, or it might indicate data logged after completion.

Implication: This visualization offers a quick snapshot of operational status, helping stakeholders understand production efficiency and backlog risks. A high completion rate is a positive indicator, but consistent tracking of pending orders is also necessary to avoid production delays.

5.4 User Interface

To build the user interface in Python, the primary package that is optimally utilized is tkinter. Within this package, a variety of different widgets are effectively employed to enhance the functionality and usability of the interface. The optimized widgets include:

- Frame: This widget displays a simple rectangle, frames help to organize your user interface, often both visually and at the coding level.
- Label: A Tkinter widget used to display simple lines of text on a GUI. Also very versatile, and can even be used to display images.
- Button: A button is used as a way for the user to interact with the User interface. Once the button is clicked, an action is triggered by the program.
- Entry: A standard Tkinter widget used to take input from the user through the user interface. A simple box is provided where the user can input text
- Scrollbar: This widget allows scrolling in a Tkinter window or enable scroll for certain widgets
- Treeview: Treeview widget displays a hierarchy of items and allows users to browse through it.

5.4.1 Manufacturing Orders tab

In this tab, the interface is divided into three different parts, including the entries for the user to type values in, the buttons section and another treeview designed to upload the database connected from MySQL.

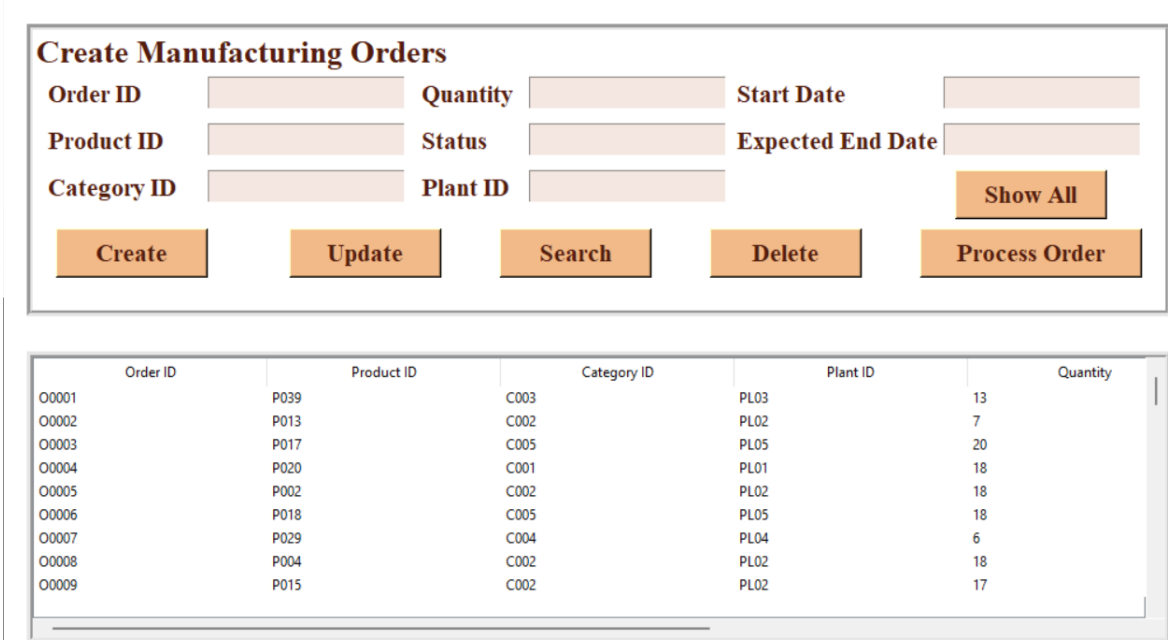


Figure 5.8: Manufacturing Orders tab

In terms of functionality, the user can click on any value within the Treeview and then use the Update or Delete buttons to modify or remove an order. Additionally, users can enter any desired criteria into the entry fields and click the Search button to conveniently locate values that match the specified criteria. The two most critical features are Create Order and Process Order, which allow users to generate a new manufacturing order and execute the order respectively.

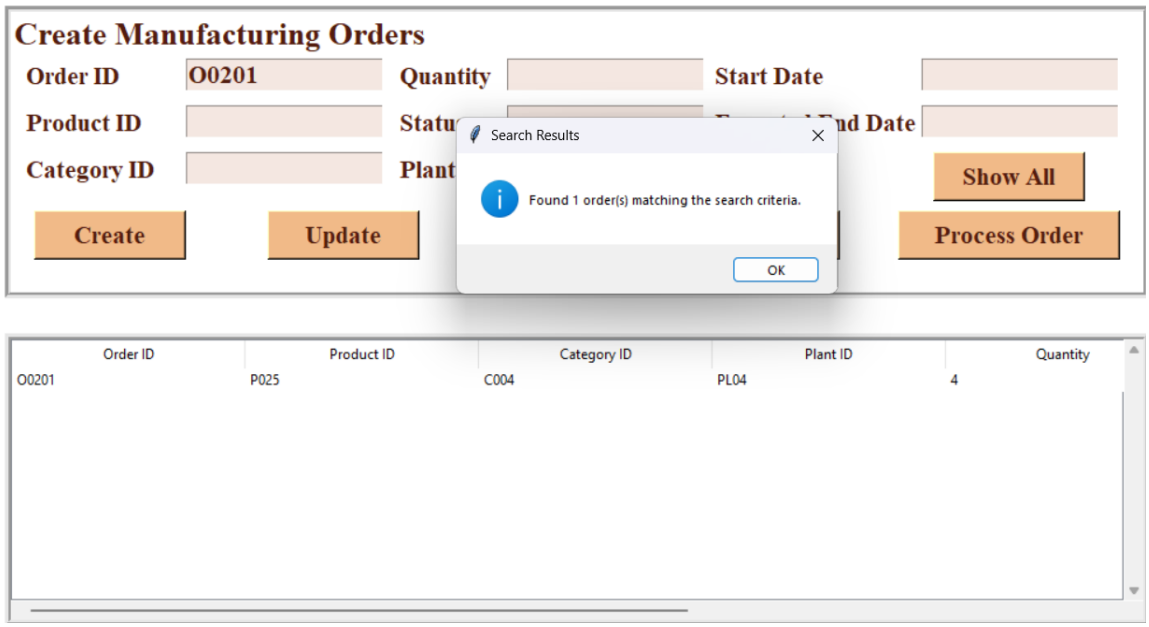


Figure 5.9: Search functionality

For an example of creating the manufacturing order, the user types in all of the entries available for creating an order. If any of the fields are missing, the system will display a message box stating "All of the fields are required" to ensure that every manufacturing order contains complete information and to prevent null values.

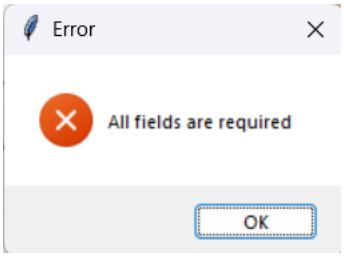


Figure 5.10: All fields are required

Whenever manufacturing order entries are filled in and the user click into the "Create" button, a new order appears in the treeview and the system will announce about the new manufacturing order created successfully.

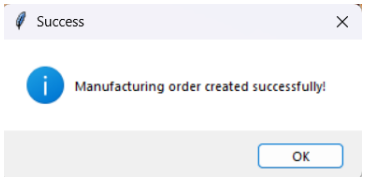


Figure 5.11: Success: Manufacturing order created successfully

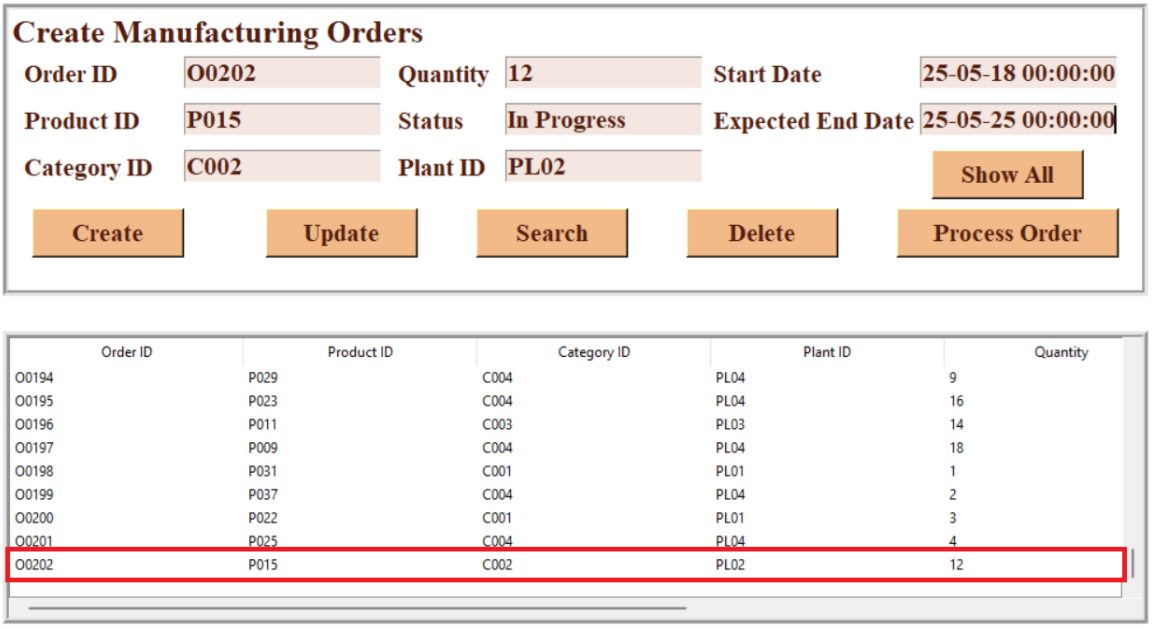


Figure 5.12: New manufacturing order appears on the treeview (status = in progress)

In case of insufficient stock in the products catalog, the system requires the user to manufacture more products.

```python
def create_order(OrderID, ProductID, CategoryID, PlantID, Quantity, StartDate, Expected_EndDate, Status, OrdersTable):
    if OrderID == "" or ProductID == "" or CategoryID == "" or PlantID == "" or Quantity == "" or StartDate == "" or Expected_EndDate == "" or Status == "":
        messagebox.showerror("Error", "All fields are required")
        return

    try:
        qty = int(Quantity)
    except ValueError:
        messagebox.showerror("Error", "Quantity must be a number")
        return

    # Force status to "In Progress" when creating an order
    Status = "In Progress"

    try:
        conn = connect_database()
        cursor = conn.cursor()
        cursor.execute("""
            INSERT INTO orders (OrderID, ProductID, CategoryID, PlantID, Quantity, StartDate, Expected_EndDate, Status)
            VALUES (%s, %s, %s, %s, %s, %s, %s, %s)
        """, (OrderID, ProductID, CategoryID, PlantID, qty, StartDate, Expected_EndDate, Status))
        conn.commit()
        conn.close()
        messagebox.showinfo("Success", "Manufacturing order created successfully!")
        load_products_to_treeview(OrdersTable)  # Refresh Treeview to show the new order
    except Exception as e:
        messagebox.showerror("Database Error", f"Failed to create manufacturing order: {str(e)}")
```

Figure 5.13: Codes for processing the manufacturing order

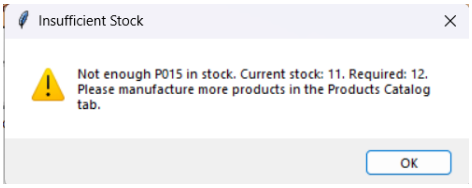


Figure 5.14: Insufficient stock (Order O0202, Product P015)

The user has to manufacture product P015 in the Products and Catalog tab to meet the required order.

5.4.2 Products Catalog tab

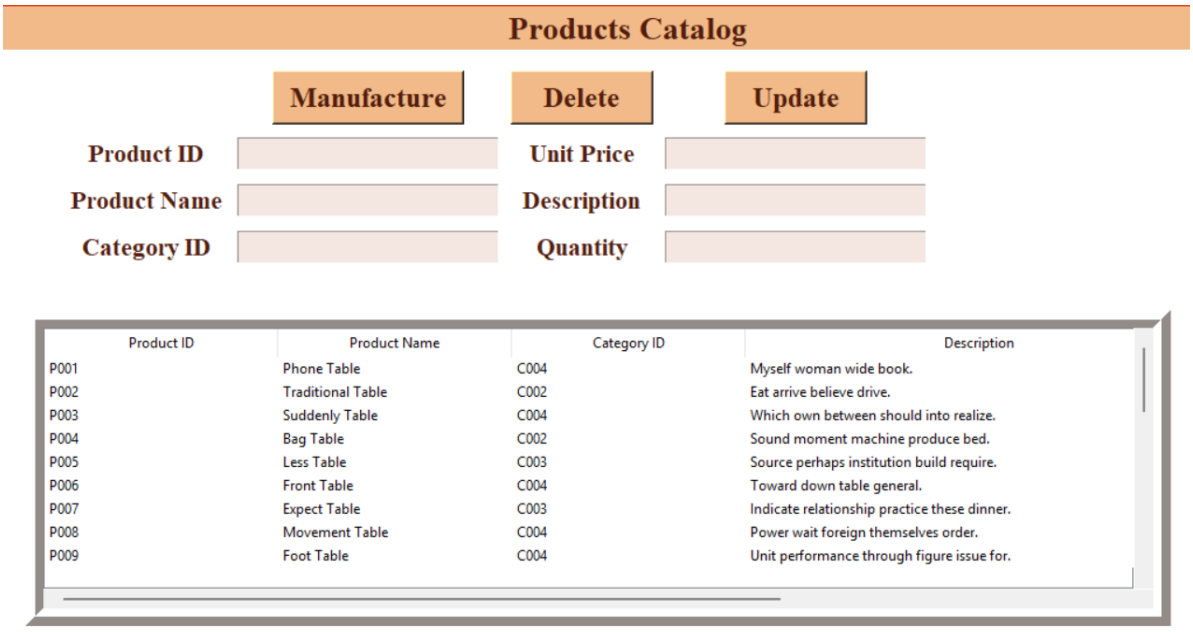
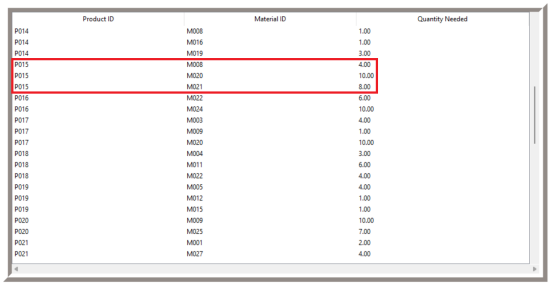


Figure 5.15: Products catalog tab

The user types into all the entries necessary to manufacture the product. The system would automatically check the bill of materials needed for the product required to manufacture.



Product ID	Material ID	Quantity Needed
P014	M008	1.00
P014	M016	1.00
P014	M019	3.00
P015	M008	4.00
P015	M020	10.00
P015	M021	8.00
P016	M022	6.00
P016	M024	10.00
P017	M003	4.00
P017	M009	1.00
P017	M020	10.00
P018	M004	3.00
P018	M011	6.00
P018	M022	4.00
P019	M005	4.00
P019	M012	1.00
P019	M015	1.00
P020	M009	10.00
P020	M025	7.00
P021	M001	2.00
P021	M027	4.00

Figure 5.16: Bill of materials interface

In case there are sufficient raw materials to produce, the system will announce success. However, if there are insufficient raw material quantities left in the inventory, the system would require the user to purchase supplemented raw materials from suppliers in the Raw materials tab.

5.4.3 Raw materials tab

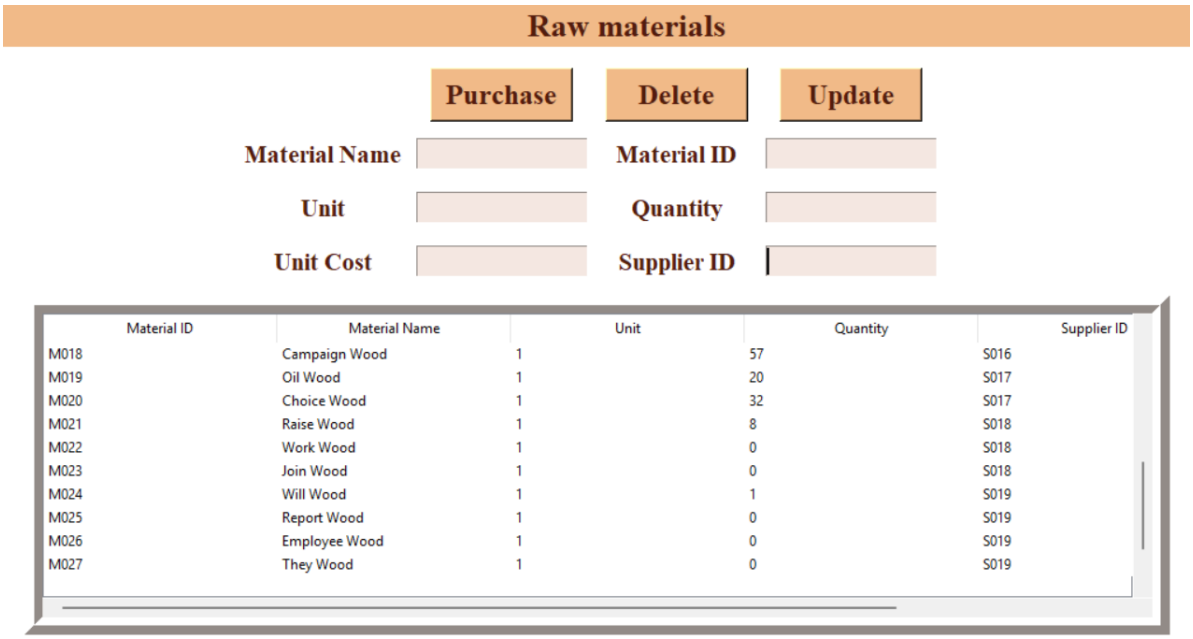


Figure 5.17: Raw materials tab

After filling all the entries required for purchasing raw materials, the user clicks into "Purchase" button to purchase from the supplier.

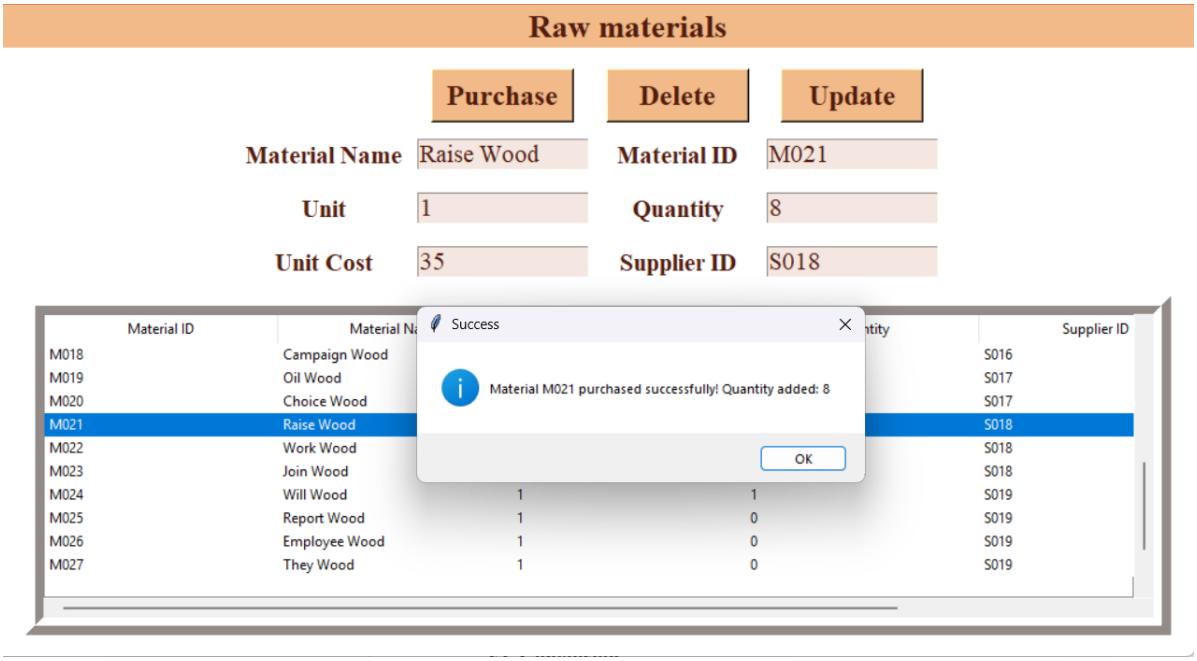


Figure 5.18: Successfully purchased material (M021)

The user then has to manufacture the product required. When the quantity of products meets the manufacturing order requirement, the user processes the manufacturing order and the execution completes.

5.5 Suppliers tab

Figure 5.19: Suppliers tab

The Suppliers tab is designed to assist users in managing the list of raw material suppliers, while also storing essential contact information. This functionality ensures the effective operation of the supply chain.

6 Conclusion

6.1 Summary of Achievements:

The project successfully designed and implemented a comprehensive production management system that includes key functionalities such as product catalog management, raw material management, manufacturing order tracking, and supplier management. The system utilizes MySQL as the database and Python for operational scripts and a simple user interface. Advanced database objects like indexes, views, stored procedures, user-defined functions, and triggers were developed to optimize performance and automate production workflows. Additionally, the system incorporates security roles and permissions, as well as backup, recovery, and regular data export policies to ensure stable long-term operations.

6.2 Recommendations:

The current dataset is still limited (approximately 200 records), which results in suboptimal index efficiency and query plan effectiveness. To enhance system performance and scalability, it is recommended to increase the volume of real data, enabling indexes to function more effectively and making query plans clearer on larger datasets.

For future improvements, consider integrating advanced features such as:

- Predictive Material Planning: Implement forecasting algorithms to optimize inventory levels, minimize waste, and meet production demands more accurately.
- ERP System Integration: Connect the production management system with popular ERP solutions to synchronize data and improve overall management efficiency.
- Improved User Interface: Develop more intuitive user interfaces such as web or mobile applications to facilitate easier operation.

References

[1] N. P. Sharma, "Introduction to nested loop joins in sql server." https://www.sqlshack.com/introduction-to-nested-loop-joins-in-sql-server/, 2025.

[2] High-Performance Programming, "Mysql composite index best practices." https://www.youtube.com/watch?v=NKakuDRExJE&list=LL, 2021.

[3] High-Performance Programming, "Use mysql explain for query optimization." https://www.youtube.com/watch?v=TukZd6LjeBc, 2021.

[4] High-Performance Programming, "Mysql covering index." https://www.youtube.com/watch?v=_fFZ2vkc0tY, 2021.

[5] High-Performance Programming, "Mysql index performance." https://www.youtube.com/watch?v=96wuo3eou0M&list=PLBrWqg4Ny6vXQZqsJ8qRLGRH9osEa45sw&index=5, 2021.

[6] High-Performance Programming, "Mysql spotting performance problems." https://www.youtube.com/watch?v=hUHQ3q9TW3o&list=PLBrWqg4Ny6vXQZqsJ8qRLGRH9osEa45sw&index=8, 2021.