**Final Project: "Design and Query a Mini Data Warehouse" (Best Buy Electronic Store)**

**Group 4**

**Ritul Soni**

**May Thu**

**Nway Ei Zaw**

California University of Management and Sciences

ADVANCED SQL

SUMMER 2025

# **GRADE: 7 out of 10**

# **FEEBACK**

**Weaknesses:**

You lost points by pasting codes for which there is no output evidence of their correctness.

Provide Evidence for Claims: The submission must include screenshots of key outputs to be credible. This includes the results of the concurrency test (e.g., two sessions demonstrating a race condition prevented by the procedure), and the before/after EXPLAIN plans for the performance section

Complete the Submission Package: The final deliverable must include all components listed in the rubric: a well-organized GitHub repository or ZIP file containing all SQL scripts, documentation, and image files in the specified folder structure (/sql, /docs, /demo). Your project lacked proper documentation.

**Strengths:** Rationale: Every query and component includes a clear explanation of its business impact, showing strong alignment with real-world use cases. Transaction Handling: The place\_order\_single stored procedure is excellent. It is complex, handles multiple scenarios (insert/update), uses proper locking (FOR UPDATE), and includes comprehensive error handling with rollbacks, perfectly demonstrating atomicity. Query Variety: The ten core queries effectively demonstrate a range of required techniques, including various JOINs, aggregations, subqueries, and CASE statements. View Implementation: Both regular views are well-designed and serve clear business purposes for simplifying reporting. Simulated Materialized View: The approach to simulating this advanced feature with a table and stored procedure is correct and appropriately implemented.

# 

# 

Table of Contents

1. Introduction
2. Project Overview
3. Business Flow
4. Design & Data
5. Core Queries
6. Views & Materialized Views
7. Transactions & Concurrency
8. Performance Optimization
9. Conclusion

## **Introduction**

This project was developed as part of the **Advanced SQL course**. The goal was to design and implement a **mini data warehouse** for a fictional retail company modeled after **Best Buy**. The project demonstrates skills in database design, complex querying, transactions, concurrency handling, and performance optimization.

The Best Buy case was chosen because it represents a **realistic, large-scale retail environment**, handling online and in-store sales, customer reviews, and supplier management.

## **Project Overview**

The project is divided into two main layers:

* **OLTP (Online Transaction Processing)** – Handles day-to-day operations such as customer orders, inventory tracking, and payments.
* **OLAP (Online Analytical Processing)** – Supports managerial decision-making by enabling trend analysis, product performance, and customer insights.

### **Key Deliverables**

1. **Schema Design** (OLTP in 3NF, OLAP in Star Schema).
2. **Core Queries** (joins, aggregations, subqueries, CTEs).
3. **Views & Materialized Views** (simplify reporting and speed up analytics).
4. **Transactions & Concurrency Control** (atomic stored procedures, row locking).
5. **Performance Tuning** (indexes, EXPLAIN plans, before/after demos).

## **Business Flow**

1. **Customer places an order** → system validates stock.
2. **Order processed by employee** → stored in Orders and Order\_Details.
3. **Inventory updated** → stock decremented, preventing overselling.
4. **Payment & shipment recorded** → ensures end-to-end transaction tracking.
5. **Data periodically moved into OLAP layer** → fact/dimension tables for reporting.

This flow supports both **operational efficiency** (fast order processing) and **strategic insights** (trend analysis, top products, sales forecasts).

## **Design & Data**

### **ER Diagram (OLTP in 3NF)**

# Entities: Customers, Products, Suppliers, Employees, Orders, OrderDetails, Payments, Inventory, Reviews, Shipments.

# **Relationships**: 1:M between Customers → Orders, Orders → OrderDetails, Products → OrderDetails, etc.

# **3NF Justification**: No redundancy, atomic attributes, no transitive dependencies.

### **OLAP Star Schema**

# **Fact Table (FactSales):** Each order line (with keys to dimensions).

# **Dimension Tables:** Date, Customer, Product, Employee, Location.

# Supports analytics with denormalized design for fast reads.



# 

# **OLTP (Online Transaction Processing) Design**

The OLTP schema is optimized for day-to-day operations like order placement, inventory tracking, and customer reviews.  
 It uses 3NF (Third Normal Form) to minimize redundancy and maintain data integrity.

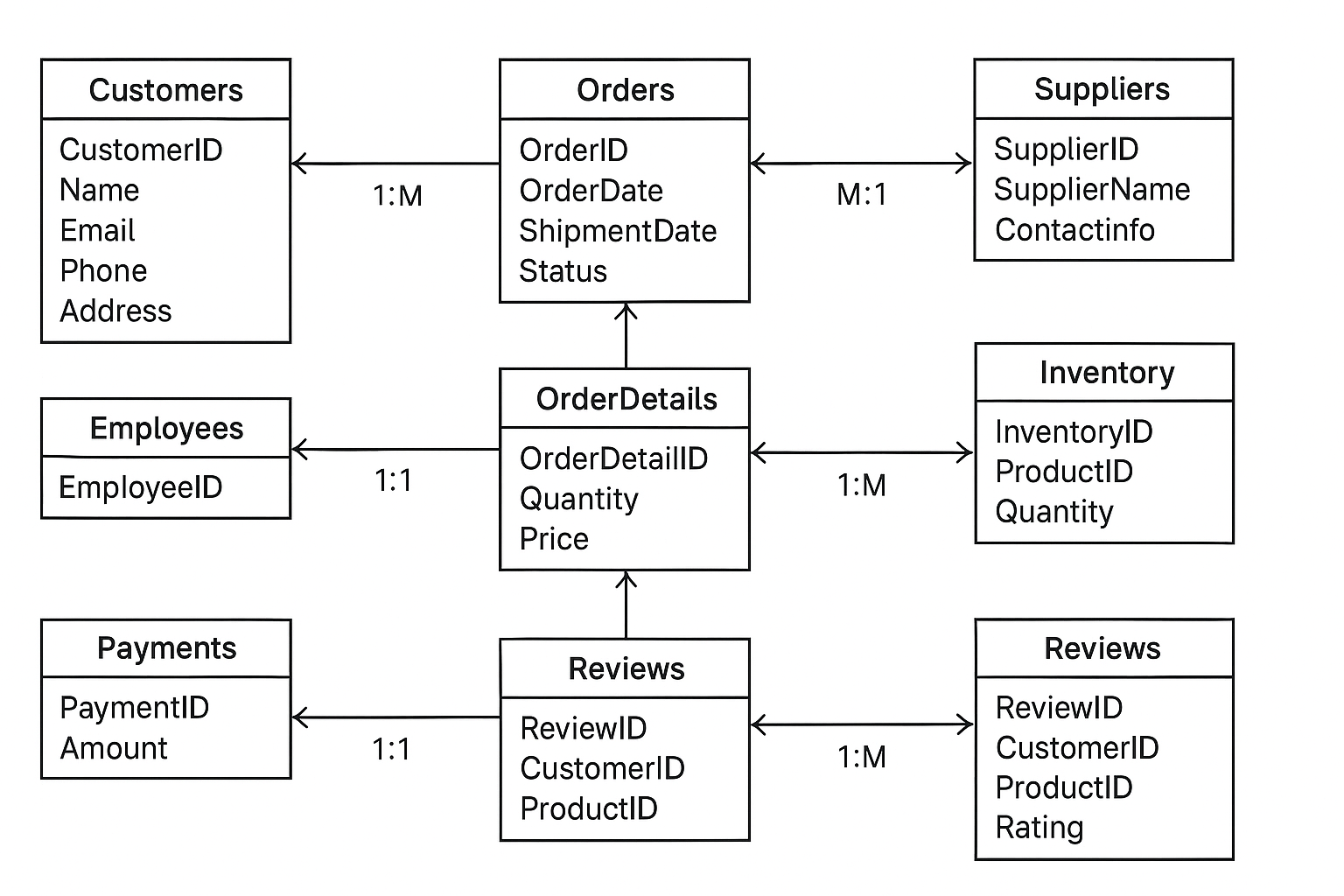
## **ER Diagram Entities**

| **Relationship** | **Type** | **Description** |
| --- | --- | --- |
| Customers → Orders | 1:M | A customer can place many orders. |
| Orders → OrderDetails | 1:M | Each order can have multiple line items. |
| Products → OrderDetails | 1:M | Each product can appear in many order details. |
| Suppliers → Products | 1:M | A supplier provides many products. |
| Employees → Orders | 1:M | An employee can process many orders. |
| Orders → Payments | 1:1 | Each order has one payment record. |
| Products → Inventory | 1:M | Each product can exist in multiple locations. |
| Products → Reviews | 1:M | A product can have multiple reviews. |
| Customers → Reviews | 1:M | A customer can leave many reviews. |
| Orders → Shipments | 1:1 or 1:M | Each order can have one or multiple shipments (depends on split shipments). |

## **3NF Tables**

| **Table** | **PK** | **FK** | **Purpose** |
| --- | --- | --- | --- |
| Customers | CustomerID | None | Stores customer personal details and address. |
| Products | ProductID | SupplierID → Suppliers | Stores product details like name, price, model year. |
| Suppliers | SupplierID | None | Stores supplier company and contact information. |
| Employees | EmployeeID | None | Tracks employees handling orders. |
| Orders | OrderID | CustomerID → Customers, EmployeeID → Employees | Stores order header data (date, status). |
| OrderDetails | OrderDetailID (or composite OrderID + ProductID) | OrderID → Orders, ProductID → Products | Stores each item sold in an order. |
| Payments | PaymentID | OrderID → Orders | Records payments for orders. |
| Inventory | InventoryID | ProductID → Products | Tracks product stock at different locations. |
| Reviews | ReviewID | CustomerID → Customers, ProductID → Products | Stores customer reviews and ratings. |
| Shipments | ShipmentID | OrderID → Orders | Tracks shipment details (carrier, tracking, status). |

### **Finalized ER Diagram**



## **Why This Design is in 3NF**

1. **1NF (Atomicity):**
   * Each column has atomic values (e.g., only one email per customer).
   * No repeating groups (e.g., separate order details instead of multiple products in one field).
2. **2NF (No Partial Dependency):**
   * Non-key attributes depend on the full primary key.
   * Example: In OrderDetails, Quantity depends on both OrderID and ProductID, not just one.
3. **3NF (No Transitive Dependency):**
   * Attributes do not depend on other non-key attributes.
   * Example: Supplier data is in Suppliers table, not duplicated in Products.

### **OLAP Star Schema Plan (Using Data)**

### **Fact Table**

**FactSales** – represents each order line (from Order Details):

| **Column** | **Source / Explanation** |
| --- | --- |
| SaleID | New unique ID for fact row |
| DateKey | Orders.OrderDate (formatted as YYYYMMDD) |
| CustomerKey | Orders.CustomerID |
| ProductKey | Order\_Details.ProductID |
| EmployeeKey | Orders.EmployeeID |
| LocationKey | Use Inventory.Location or store/warehouse info |
| Quantity | Order\_Details.Quantity |
| TotalSales | Order\_Details.Quantity \* Order\_Details.UnitPrice |

**One Row Meaning:**

* A row shows a single product purchased in an order: who bought it, when, handled by which employee, where, quantity, and total price.

### **Dimension Tables**

1. **DimDate** – OrderDate broken into Year, Month, Quarter.
2. **DimCustomer** – Customers table.
3. **DimProduct** – Products table.
4. **DimEmployee** – Employees table.
5. **DimLocation** – Inventory.Location or store/warehouse location.

## **How OLTP and OLAP Work Together**

* **OLTP:** Handles daily operations like placing orders, updating stock, and recording payments.
* **ETL (Extract, Transform, Load):** Periodically moves data from OLTP to OLAP.
* **OLAP:** Used by managers and analysts to generate reports, dashboards, and forecasts.

## 

## **Summary**

| **Aspect** | **OLTP** | **OLAP** |
| --- | --- | --- |
| **Purpose** | Transaction processing | Analytics and reporting |
| **Schema Type** | Normalized (3NF) | Star Schema (denormalized) |
| **Performance** | Optimized for write speed | Optimized for read speed |
| **Tables** | Many small, normalized tables | Few large, denormalized tables |
| **Users** | Cashiers, sales staff, operations | Business analysts, managers |

## **Why This Design is useful for Best Buy**

* **Supports daily operations:** Customers get fast service when shopping online.
* **Improves decision-making:** Management can track sales trends and stock issues.
* **Scalable:** Can handle **millions of orders**, especially during big sales like **Black Friday**.
* **Customer-focused:** Helps identify loyal customers and improve marketing campaigns.
* **Data-driven growth:** Reports help Best Buy stay competitive in the electronics retail industry.

## **Core Queries**

**Live Demo :**

### **Example 1: Total Sales Per Product (with JOIN and GROUP BY)**

SELECT p.ProductName,

SUM(od.Quantity \* od.UnitPrice) AS TotalSales

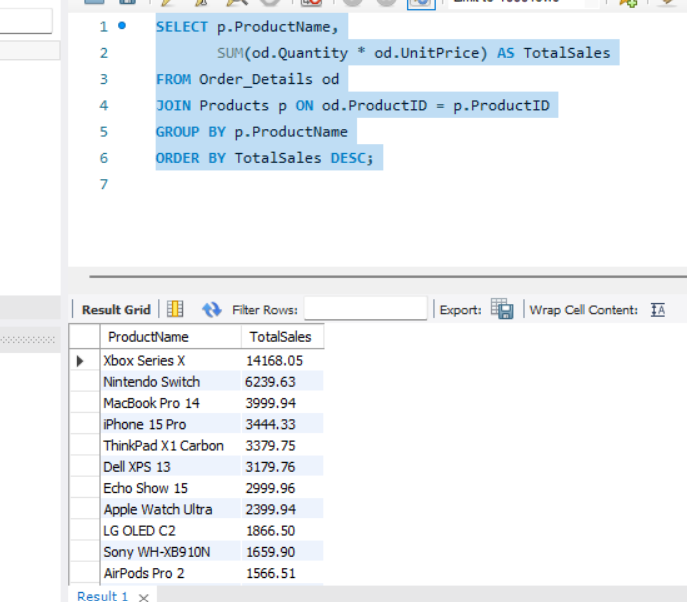
FROM Order\_Details od

JOIN Products p ON od.ProductID = p.ProductID

GROUP BY p.ProductName

ORDER BY TotalSales DESC;

**Result:**



**Business effect:**

* Shows which products bring in the most revenue.
* Helps management focus on high-performing items or promote low-performing ones.

### **Example 2: List customers who purchased a specific product(Using Join)**

### SELECT c.FirstName, c.LastName, c.Email

### FROM Customers c

### JOIN Orders o ON c.CustomerID = o.CustomerID

### JOIN Order\_Details od ON o.OrderID = od.OrderID

### WHERE od.ProductID = 3;

**Result:**

### 

**Business Impact:**

* Identifies buyers of a specific product for targeted marketing or promotions.
* Helps increase repeat sales by reaching interested customers.

### 

### **Example 3: Orders per month (using DATE functions and GROUP BY)**

SELECT MONTH(o.OrderDate) AS Month,

YEAR(o.OrderDate) AS Year,

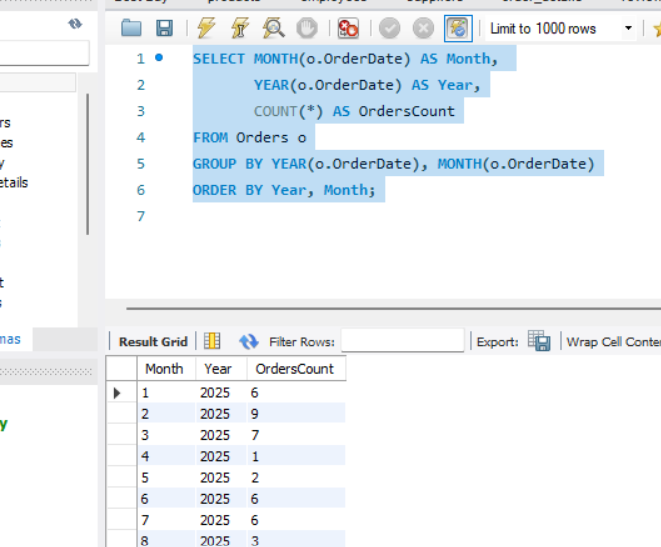
COUNT(\*) AS OrdersCount

FROM Orders o

GROUP BY YEAR(o.OrderDate), MONTH(o.OrderDate)

ORDER BY Year, Month;

**Result:**



**Business effect:**

* Shows sales trends over time.
* Helps plan promotions or staffing based on peak months.

### 

### **Example 4 : Products low in stock (with CASE/WHEN)**

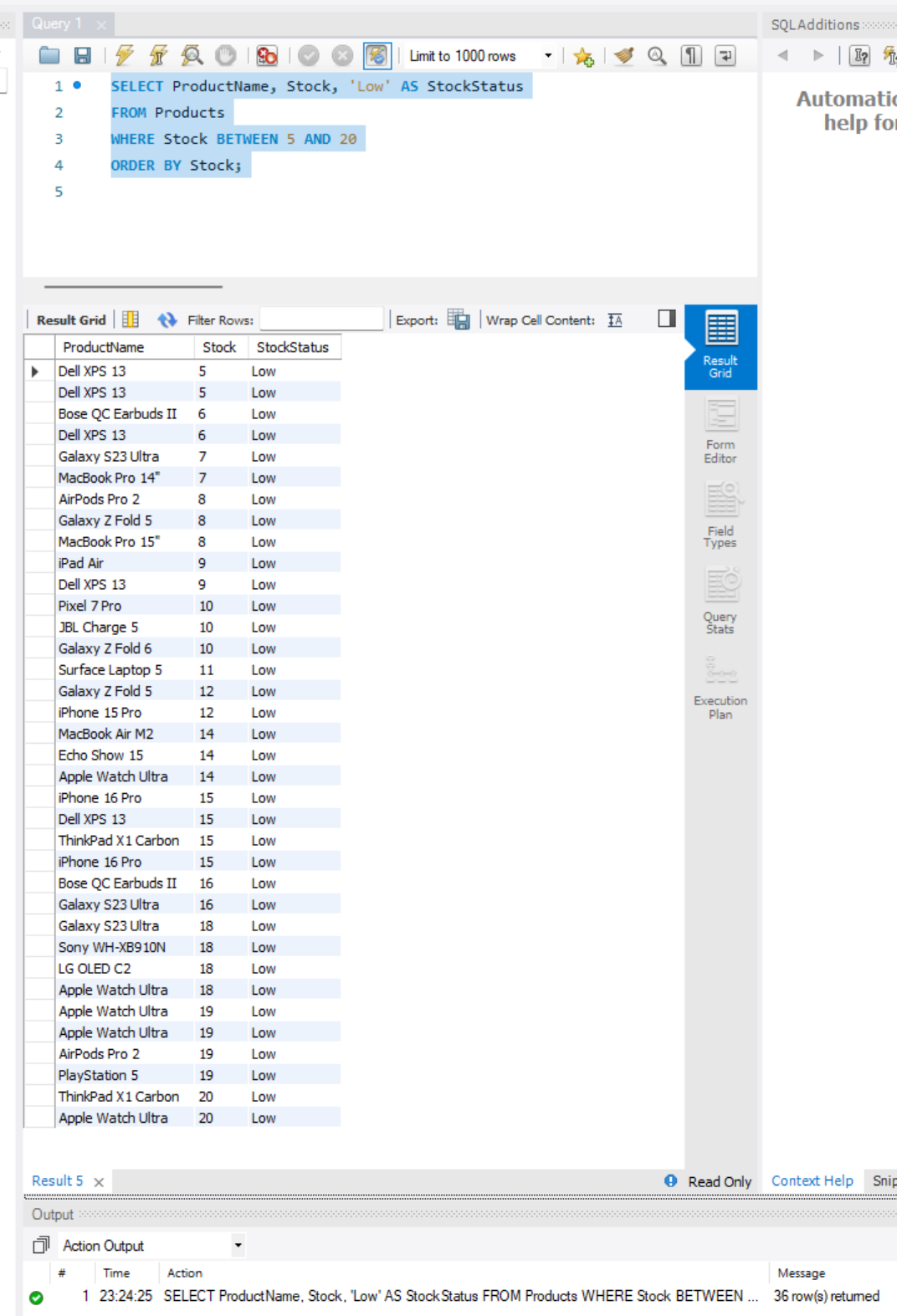
SELECT ProductName, Stock, 'Low' AS StockStatus

FROM Products

WHERE Stock BETWEEN 5 AND 20

ORDER BY Stock;

**Result:**



**Business effect:**

* Quickly identifies products needing reorder.
* Prevents stockouts, ensuring customers can always buy popular items.

### **Example 5: Top 5 employees by sales (JOIN + GROUP BY + ORDER BY)**

SELECT e.FirstName, e.LastName,

SUM(od.Quantity \* od.UnitPrice) AS SalesAmount

FROM Orders o

JOIN Employees e ON o.EmployeeID = e.EmployeeID

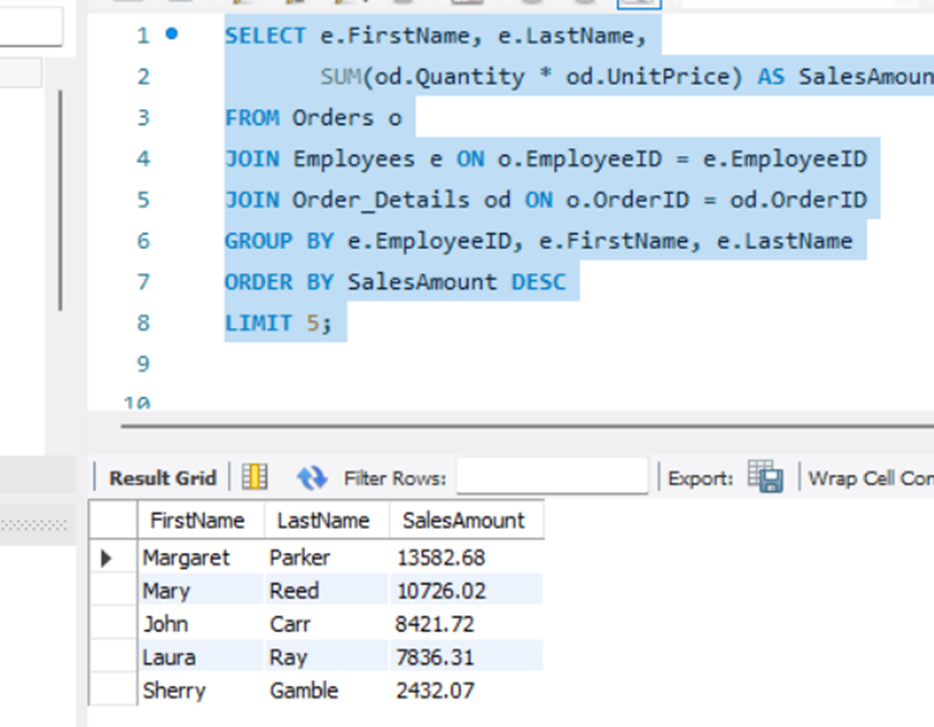
JOIN Order\_Details od ON o.OrderID = od.OrderID

GROUP BY e.EmployeeID, e.FirstName, e.LastName

ORDER BY SalesAmount DESC

LIMIT 5;

**Result:**



**Business effect:**

* Identifies top-performing sales staff.
* Can motivate employees and allocate resources efficiently.

### **Example 6: All orders with customer and employee info (INNER JOIN + subquery for total amount)**

SELECT o.OrderID,

c.FirstName AS CustomerFirstName,

c.LastName AS CustomerLastName,

e.FirstName AS EmployeeFirstName,

e.LastName AS EmployeeLastName,

(

SELECT SUM(od.Quantity \* od.UnitPrice)

FROM Order\_Details od

WHERE od.OrderID = o.OrderID

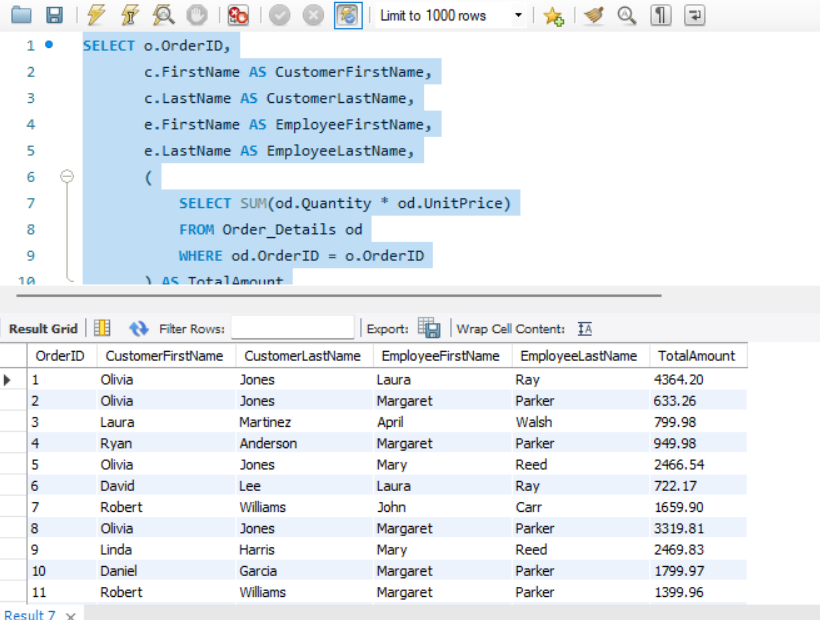
) AS TotalAmount

FROM Orders o

INNER JOIN Customers c ON o.CustomerID = c.CustomerID

INNER JOIN Employees e ON o.EmployeeID = e.EmployeeID;

**Result:**

****

**Business effect:**

* Links customer and employee data per order.
* Shows total revenue per order.
* Helps track employee performance and customer sales.

### **Example 7: Customers with pending orders and last product purchased (LEFT JOIN + subquery)**

### SELECT c.FirstName, c.LastName, o.Status,

### (

### SELECT p.ProductName

### FROM Order\_Details od

### JOIN Products p ON od.ProductID = p.ProductID

### WHERE od.OrderID = o.OrderID

### ORDER BY od.OrderID DESC

### LIMIT 1

### ) AS LastProductPurchased

### FROM Customers c

### LEFT JOIN Orders o ON c.CustomerID = o.CustomerID

### WHERE o.Status = 'Pending';

**Result:**

### 

### **Business effect:**

### **Shows pending orders with the most recent product per customer.**

### **Helps prioritize follow-up and shipping decisions.**

**Example 8:**  **Check stock status with CASE/WHEN**

SELECT ProductName, Stock,

CASE

WHEN Stock = 0 THEN 'Out of Stock'

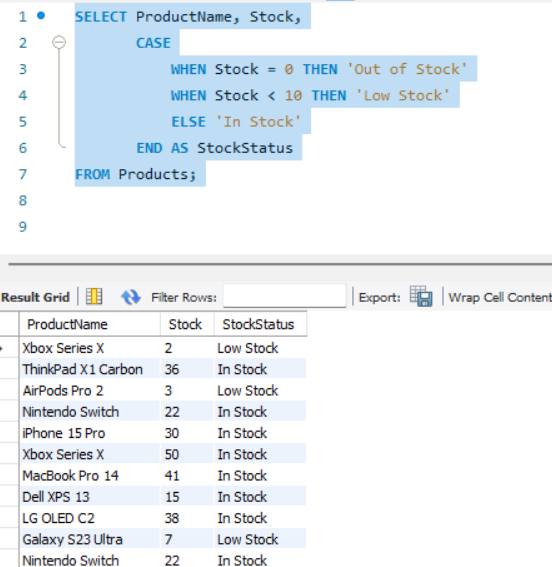
WHEN Stock < 10 THEN 'Low Stock'

ELSE 'In Stock'

END AS StockStatus

FROM Products;

**Result:**



**Business effect:**

* Quickly identifies products needing restock.
* Ensures popular products are always available for customers.

**Example 9 : All orders with products (INNER JOIN multiple tables)**

SELECT

Orders.OrderID,

Orders.OrderDate,

Customers.FirstName,

Customers.LastName,

Products.ProductName,

Order\_Details.Quantity,

Order\_Details.UnitPrice,

(Order\_Details.Quantity \* Order\_Details.UnitPrice) AS LineTotal

FROM Orders

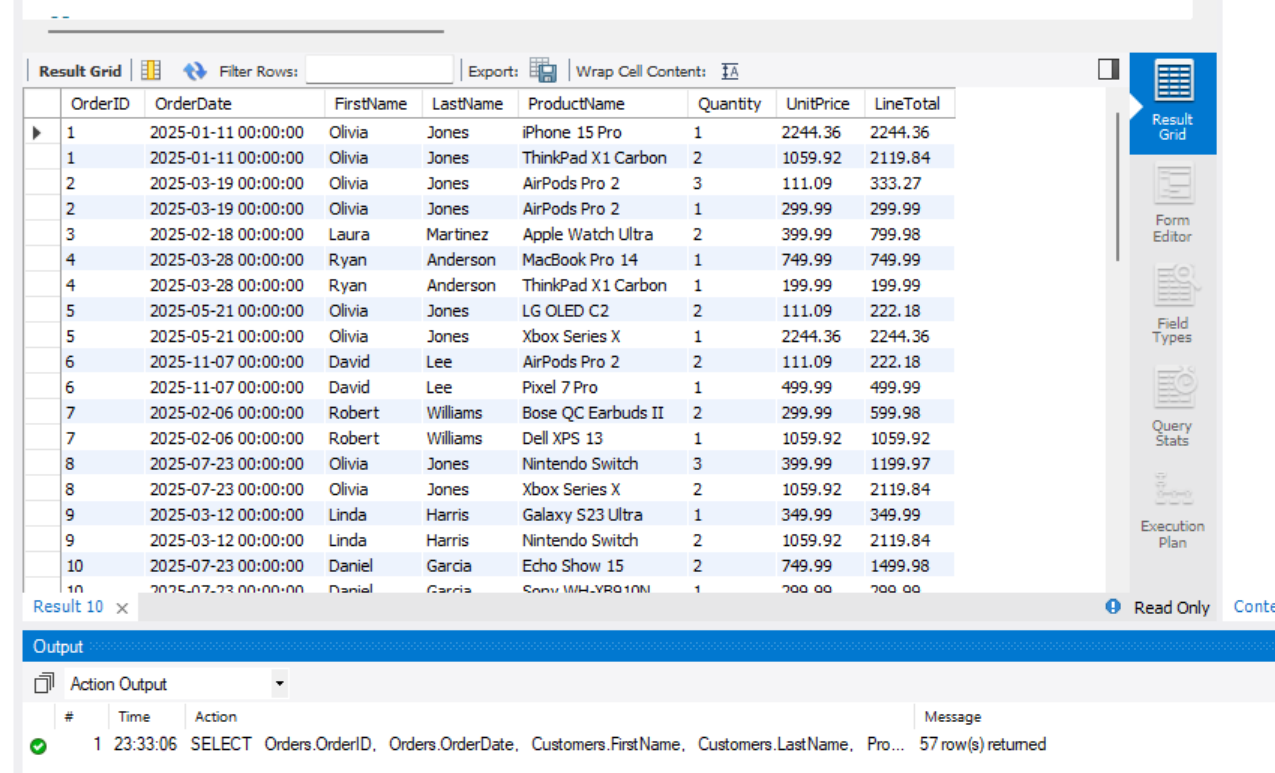
INNER JOIN Customers ON Customers.CustomerID = Orders.CustomerID

INNER JOIN Order\_Details ON Order\_Details.OrderID = Orders.OrderID

INNER JOIN Products ON Products.ProductID = Order\_Details.ProductID

ORDER BY Orders.OrderID, Products.ProductName;

**Result:**

****

**Business effect:**

* **Shows every product purchased per order.**
* **Helps with packing, shipping, and product-level analysis.**

**Example 10: Find Top 5 Most Expensive Products(Subquery,comparison operator)**

SELECT

\*

FROM Products

WHERE Price >= (

SELECT DISTINCT Price

FROM Products

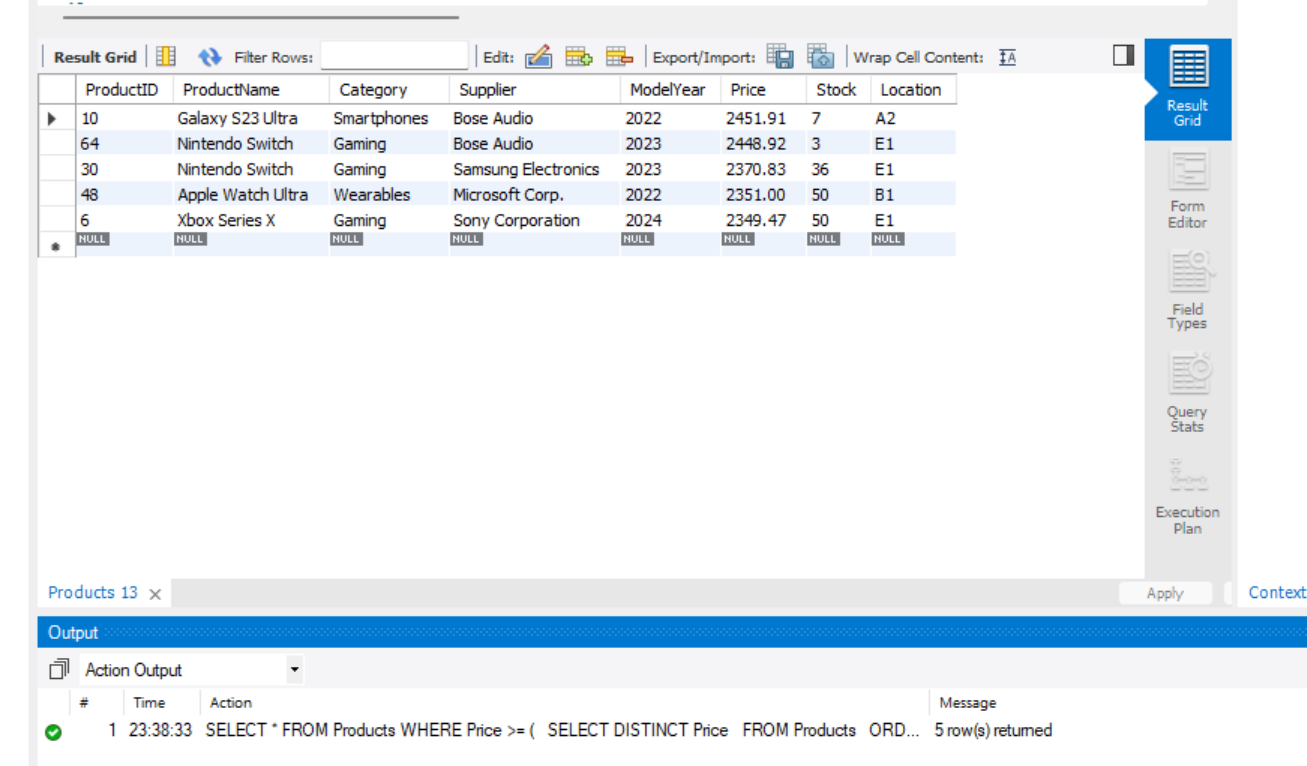
ORDER BY Price DESC

LIMIT 1 OFFSET 4 -- 5th highest distinct price

)

ORDER BY Price DESC, ProductName;

**Result:**

****

**Business Effect**

* **The subquery finds the 5th highest price in the Products table.**
* **The outer query selects all products priced equal to or above that threshold.**
* **Helps management identify premium products for marketing campaigns, discount decisions, or inventory focus.**

## **Views & Materialized Views**

**Regular View: Product Sales Summary**

# **Product Sales Summary**

## **Before (no view; repeated logic in each report)**

SELECT

p.ProductID,

p.ProductName,

p.Price,

IFNULL(SUM(od.Quantity), 0) AS TotalUnitsSold,

IFNULL(SUM(od.Quantity \* od.UnitPrice), 0) AS TotalRevenue

FROM Products AS p

LEFT JOIN Order\_Details AS od

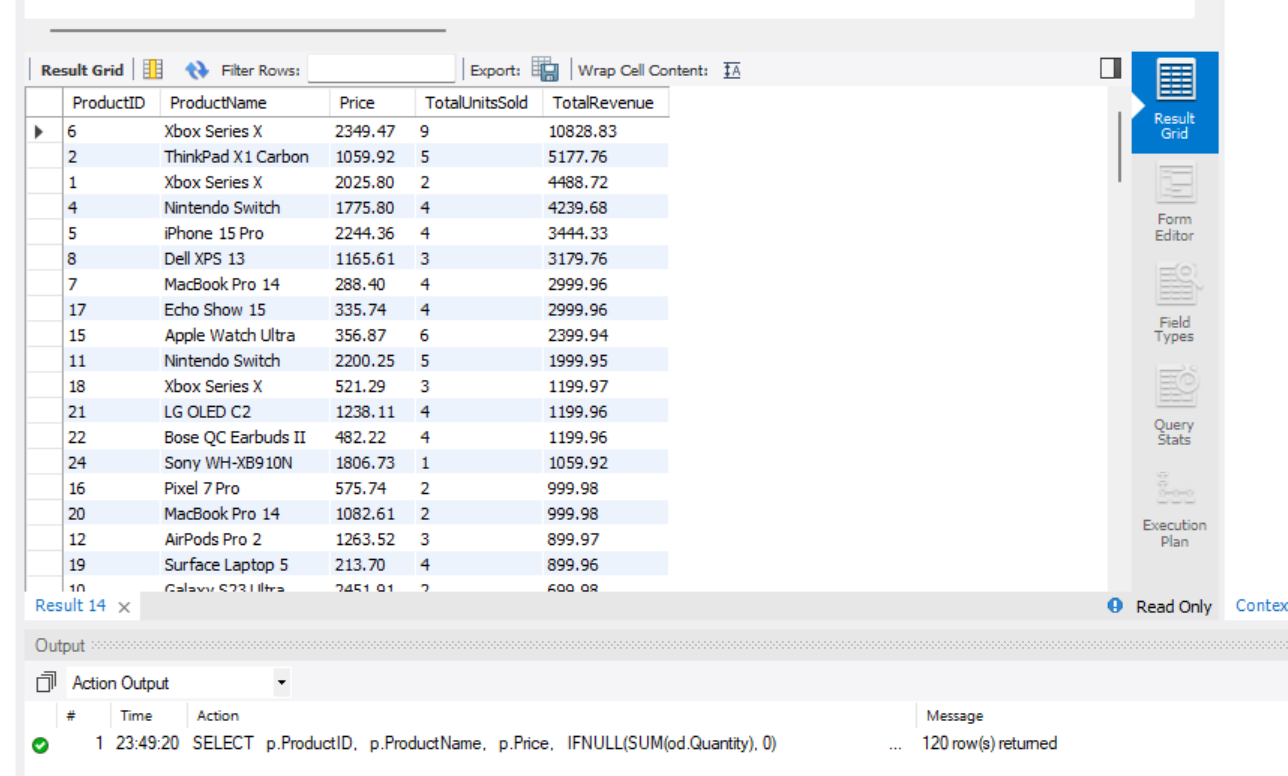
ON od.ProductID = p.ProductID

GROUP BY

p.ProductID, p.ProductName, p.Price

ORDER BY TotalRevenue DESC;

**Before Result:**



After (define a view once; reuse everywhere)

CREATE OR REPLACE VIEW ProductSalesSummary AS

SELECT

p.ProductID,

p.ProductName,

p.Price,

IFNULL(SUM(od.Quantity), 0) AS TotalUnitsSold,

IFNULL(SUM(od.Quantity \* od.UnitPrice), 0) AS TotalRevenue

FROM Products AS p

LEFT JOIN Order\_Details AS od

ON od.ProductID = p.ProductID

GROUP BY

p.ProductID, p.ProductName, p.Price;

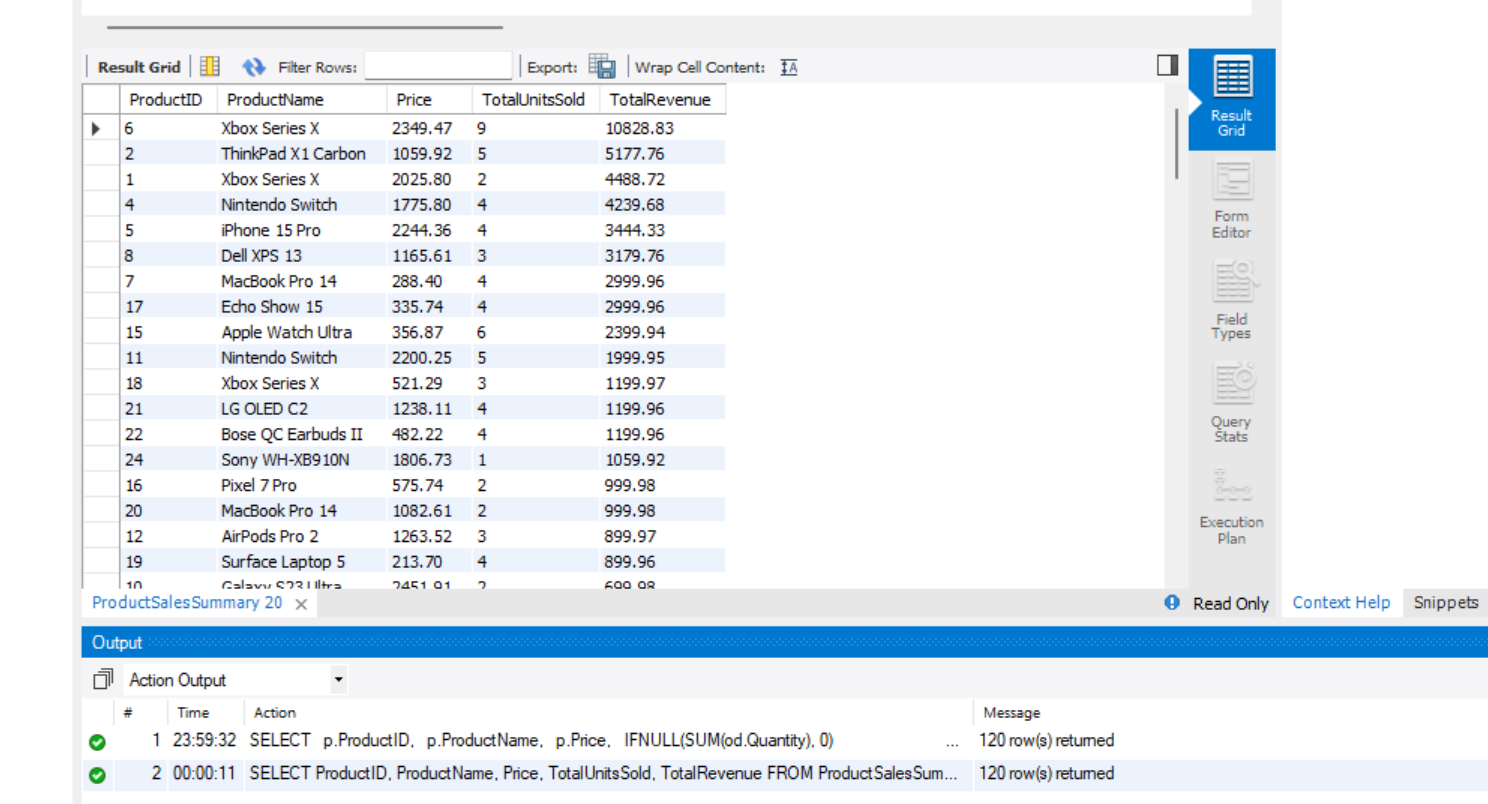
Example :

SELECT ProductID, ProductName, Price, TotalUnitsSold, TotalRevenue

FROM ProductSalesSummary

ORDER BY TotalRevenue DESC;

After Result :



**Business Impact:**

* Provides quick insights into product performance without repeatedly writing complex queries.
* Supports inventory planning, promotions, and supplier negotiations.

1. Low Stock Products

Before (no view; write full CASE each time)

SELECT

ProductID,

ProductName,

Stock,

Price,

CASE

WHEN Stock = 0 THEN 'Out of Stock'

WHEN Stock <= 5 THEN 'Low Stock'

ELSE 'In Stock'

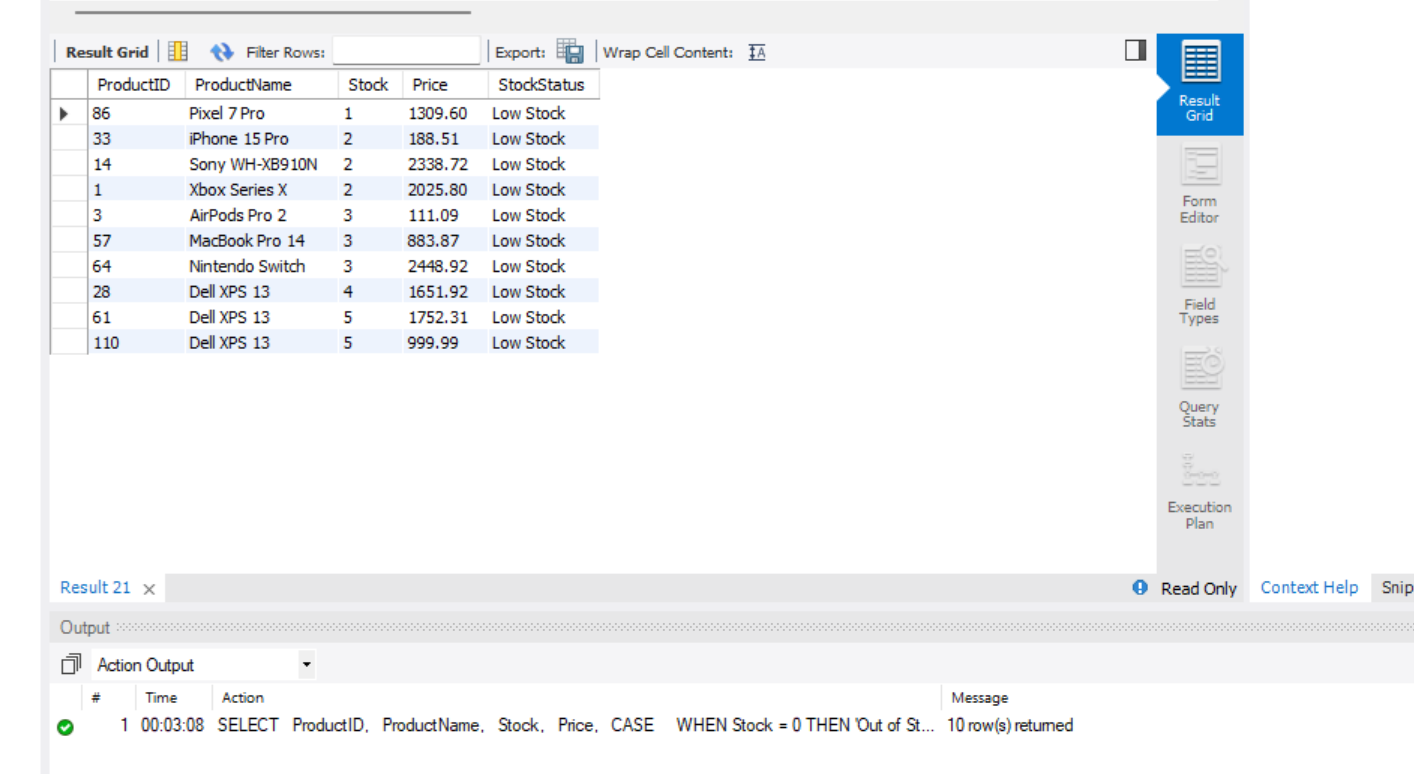
END AS StockStatus

FROM Products

WHERE Stock <= 5

ORDER BY Stock ASC, ProductName;

**Before Result:**



After (define a reusable view)

CREATE OR REPLACE VIEW LowStockProducts AS

ProductID,

SELECT

ProductName,

Stock,

Price,

CASE

WHEN Stock = 0 THEN 'Out of Stock'

WHEN Stock <= 5 THEN 'Low Stock'

ELSE 'In Stock'

END AS StockStatus

FROM Products

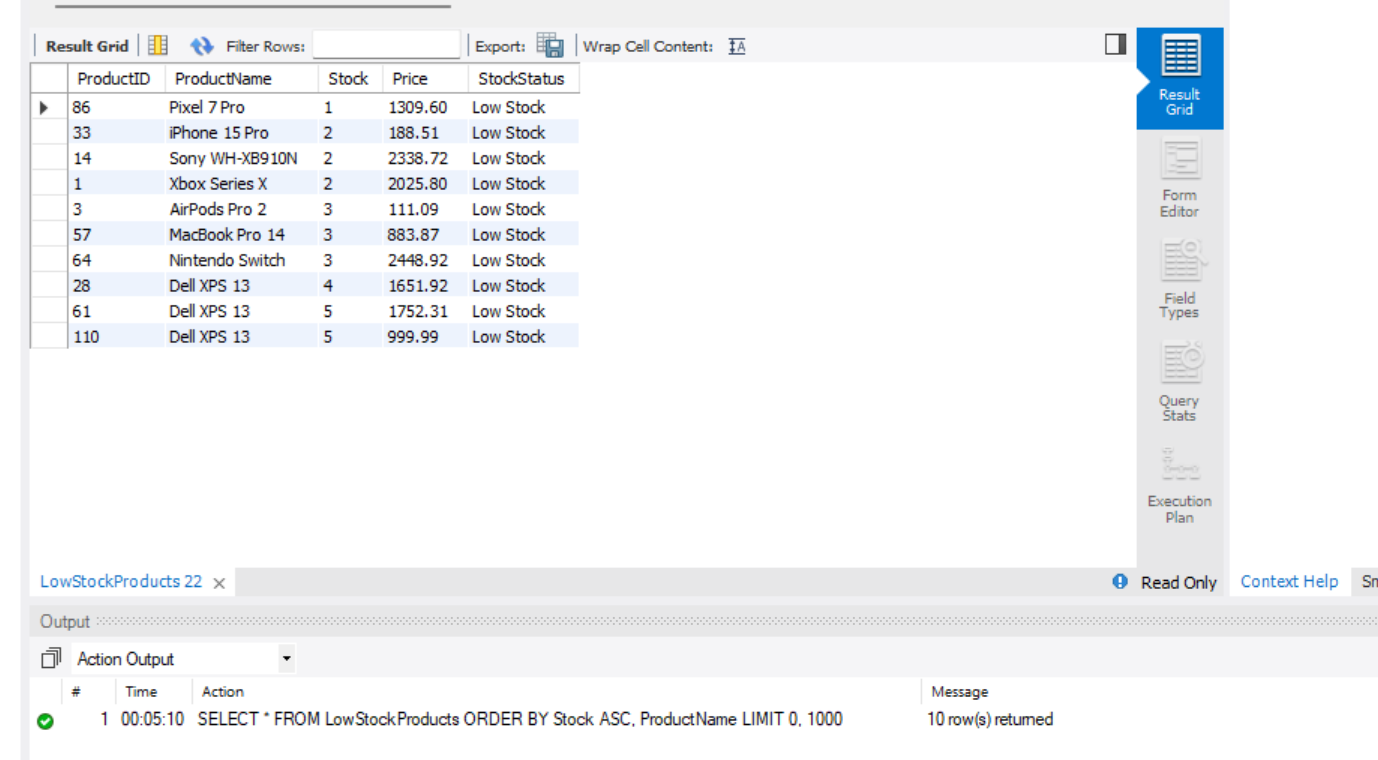
WHERE Stock <= 5;

SELECT \*

FROM LowStockProducts

ORDER BY Stock ASC, ProductName;

After Result :



**Business Impact:**

* Quickly identifies products that need restocking.
* Prevents lost sales due to out-of-stock items.
* Helps maintain customer satisfaction and operational efficiency.

## **Simulated Materialized View**

### **Step 1: Create the table**

CREATE TABLE MaterializedProductSales (

ProductID INT PRIMARY KEY,

ProductName VARCHAR(100),

TotalUnitsSold INT,

TotalRevenue DECIMAL(10,2),

LastUpdated TIMESTAMP DEFAULT CURRENT\_TIMESTAMP

);

### **Step 2: Stored procedure to refresh data**

DELIMITER //

CREATE PROCEDURE RefreshMaterializedProductSales()

BEGIN

TRUNCATE TABLE MaterializedProductSales;

INSERT INTO MaterializedProductSales (ProductID, ProductName, TotalUnitsSold, TotalRevenue)

SELECT p.ProductID,

p.ProductName,

IFNULL(SUM(od.Quantity), 0),

IFNULL(SUM(od.Quantity \* od.UnitPrice), 0)

FROM Products p

LEFT JOIN Order\_Details od ON p.ProductID = od.ProductID

GROUP BY p.ProductID, p.ProductName;

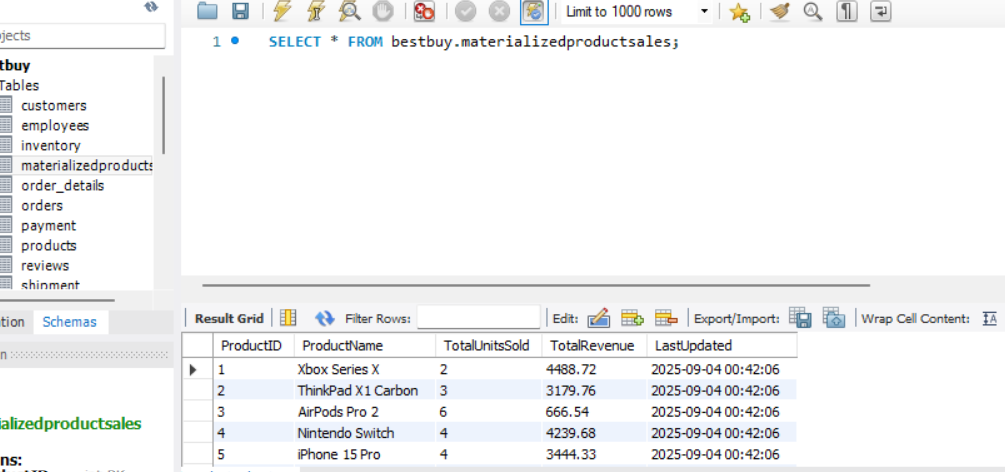
END //

DELIMITER ;

### **Step 3: Call the procedure to refresh the table**

CALL RefreshMaterializedProductSales();

**Result:**



**Business Impact:**

* Provides pre-aggregated product sales data for **faster reporting**.
* Reduces computation time for dashboards or frequent reports.
* Can be scheduled (e.g., daily via MySQL Event Scheduler) to always have up-to-date data.

## **Transactions & Concurrency**

### **1) Stored procedure: place a (single-product) order with inventory check + error handling**

This procedure demonstrates atomicity: it will either fully place the order and decrement inventory, or rollback on any error (including insufficient stock).

DELIMITER $$

CREATE PROCEDURE place\_order\_single(

IN p\_customer\_id INT,

IN p\_employee\_id INT,

IN p\_product\_id INT,

IN p\_quantity INT, -- must be > 0

OUT p\_order\_id INT,

OUT p\_note VARCHAR(255)

)

main: BEGIN

DECLARE cur\_stock INT DEFAULT NULL;

DECLARE cur\_price DECIMAL(10,2) DEFAULT NULL;

-- default outputs

SET p\_order\_id := NULL;

SET p\_note := NULL;

-- guard: positive quantity only

IF p\_quantity IS NULL OR p\_quantity <= 0 THEN

SET p\_note := 'ERROR: quantity must be a positive integer';

LEAVE main;

END IF;

START TRANSACTION;

/\* 1) Product not found (no lock needed yet) \*/

IF NOT EXISTS (SELECT 1 FROM Inventory WHERE ProductID = p\_product\_id) THEN

ROLLBACK;

SET p\_note := CONCAT('PRODUCT\_NOT\_FOUND: ProductID=', p\_product\_id);

LEAVE main;

END IF;

/\* 2) Lock the product row and grab current values \*/

SELECT StockQuantity, Price

INTO cur\_stock, cur\_price

FROM Inventory

WHERE ProductID = p\_product\_id

FOR UPDATE;

/\* 3) Insufficient stock \*/

IF cur\_stock < p\_quantity THEN

ROLLBACK;

SET p\_note := CONCAT('INSUFFICIENT\_STOCK: need ', p\_quantity, ', have ', cur\_stock);

LEAVE main;

END IF;

/\* 4) Success: create order + line, then decrement stock \*/

INSERT INTO Orders (CustomerID, EmployeeID, OrderDate, Status, PaymentStatus, ShippingMethod)

VALUES (p\_customer\_id, p\_employee\_id, NOW(), 'Placed', 'Pending', 'Standard');

SET p\_order\_id := LAST\_INSERT\_ID();

INSERT INTO Order\_Details (OrderID, ProductID, Quantity, UnitPrice, DiscountApplied)

VALUES (p\_order\_id, p\_product\_id, p\_quantity, cur\_price, 0.00);

UPDATE Inventory

SET StockQuantity = StockQuantity - p\_quantity

WHERE ProductID = p\_product\_id;

COMMIT;

SET p\_note := CONCAT(

'SUCCESS\_INSERT: Order ', p\_order\_id,

' | Product ', p\_product\_id,

' | Qty ', p\_quantity,

' | RemainingStock ', cur\_stock - p\_quantity

);

END$$

DELIMITER ;

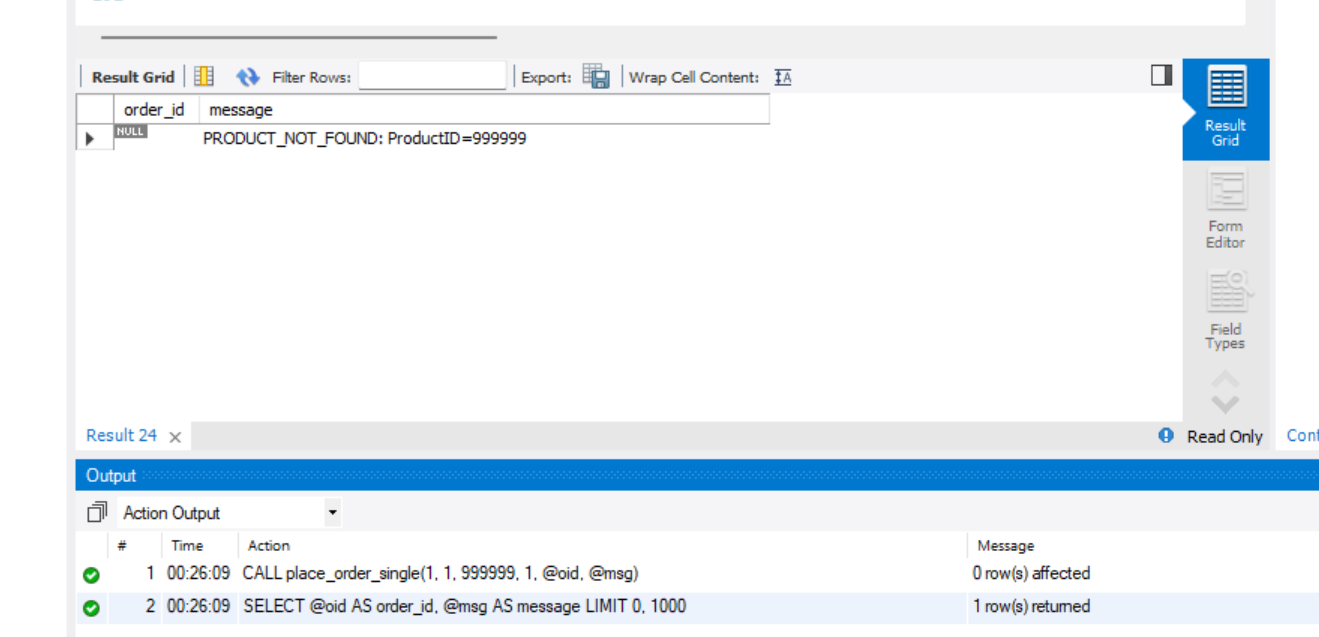
Demo Scenarios

A) Product Not Found

CALL place\_order\_single(1, 1, 999999, 1, @oid, @msg); -- non-existent ProductID

SELECT @oid AS order\_id, @msg AS message;

**Result:**

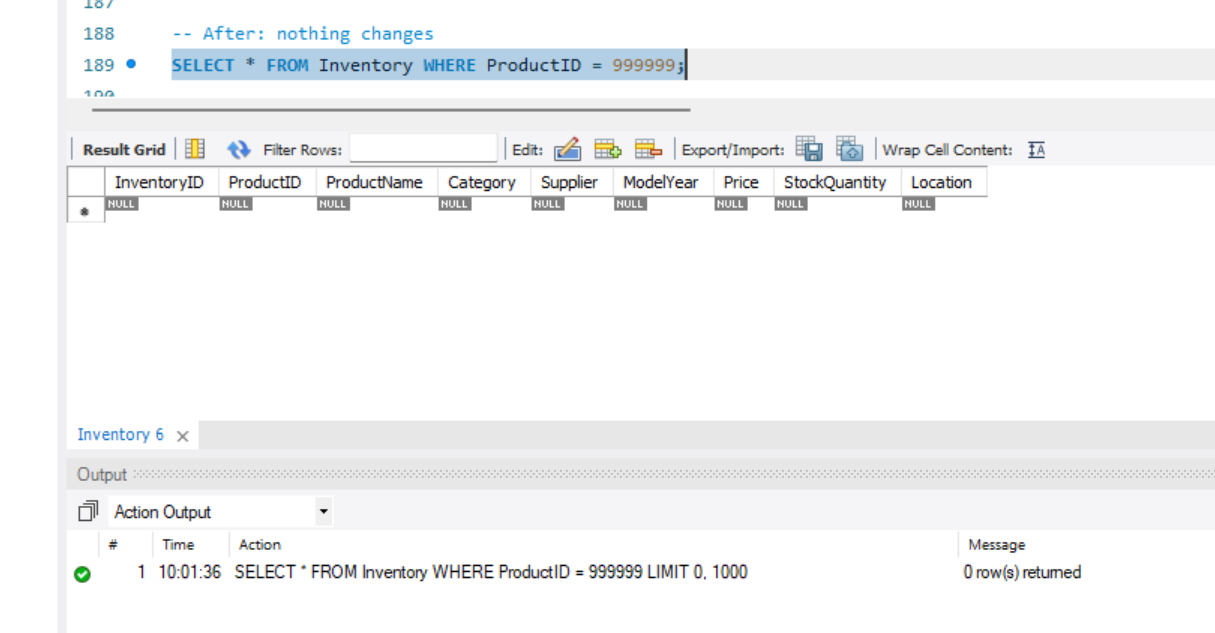


**Expected message:** PRODUCT\_NOT\_FOUND: ProductID=999999 does not exist

-- After: nothing changes

SELECT \* FROM Inventory WHERE ProductID = 999999;

**Result:**

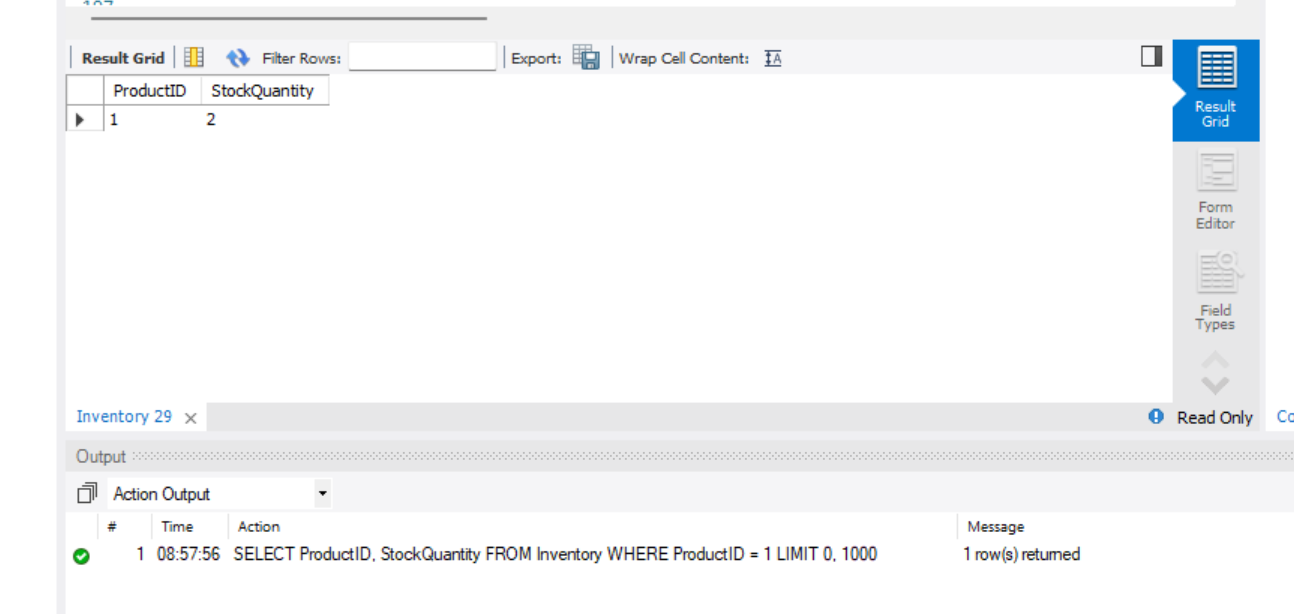


**B) Insufficient Stock**

– **Before**

SELECT ProductID, StockQuantity FROM Inventory WHERE ProductID = 1;

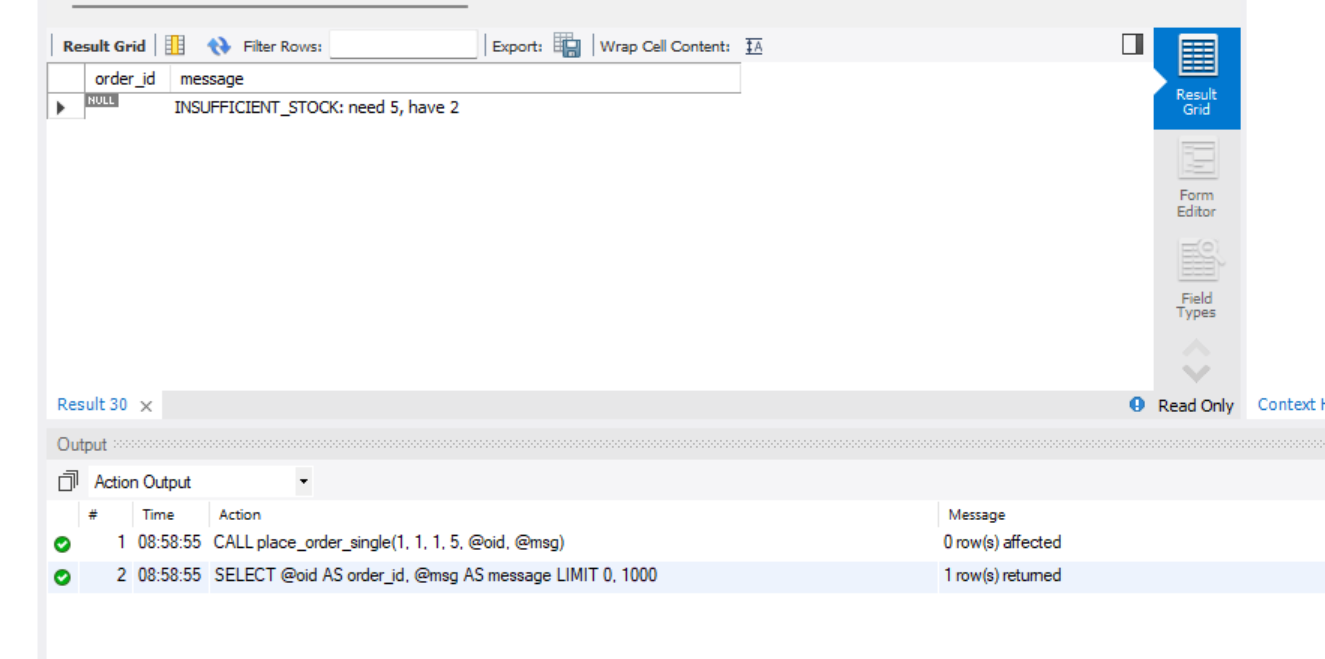
**Result:**



CALL place\_order\_single(1, 1, 1, 5, @oid, @msg);

SELECT @oid AS order\_id, @msg AS message;

**After Result:**



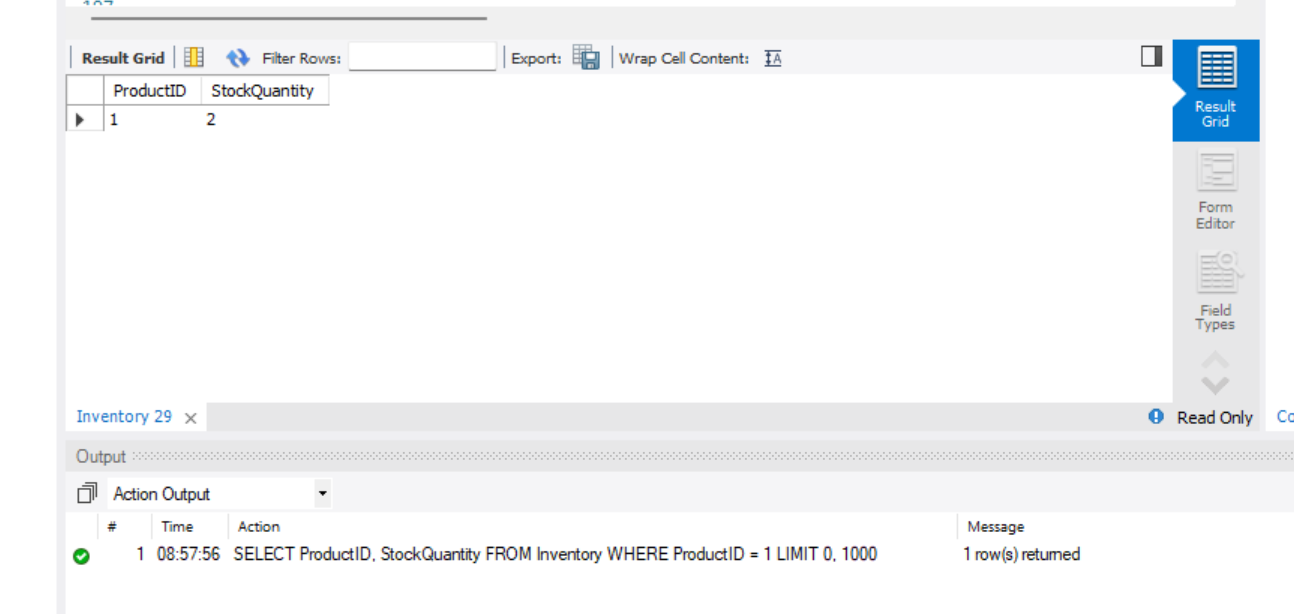
**Expected message:**

INSUFFICIENT\_STOCK: need 5 ; have 2

-- After: stock unchanged (rollback)

SELECT ProductID, StockQuantity FROM Inventory WHERE ProductID = 1;

**Result:**



**C) SUCCESS INSERT (New Order)**

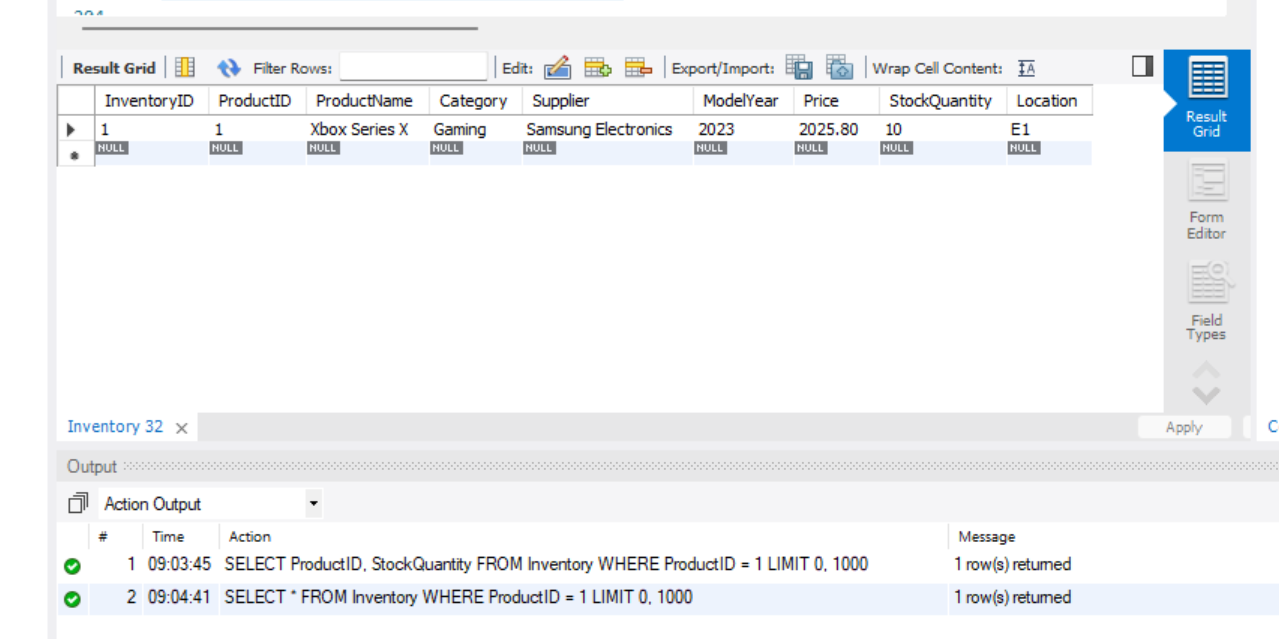
-- Set up: enough stock

UPDATE Inventory SET StockQuantity = 10 WHERE ProductID = 1;

-- Before

SELECT \* FROM Inventory WHERE ProductID = 2;

**Result:**

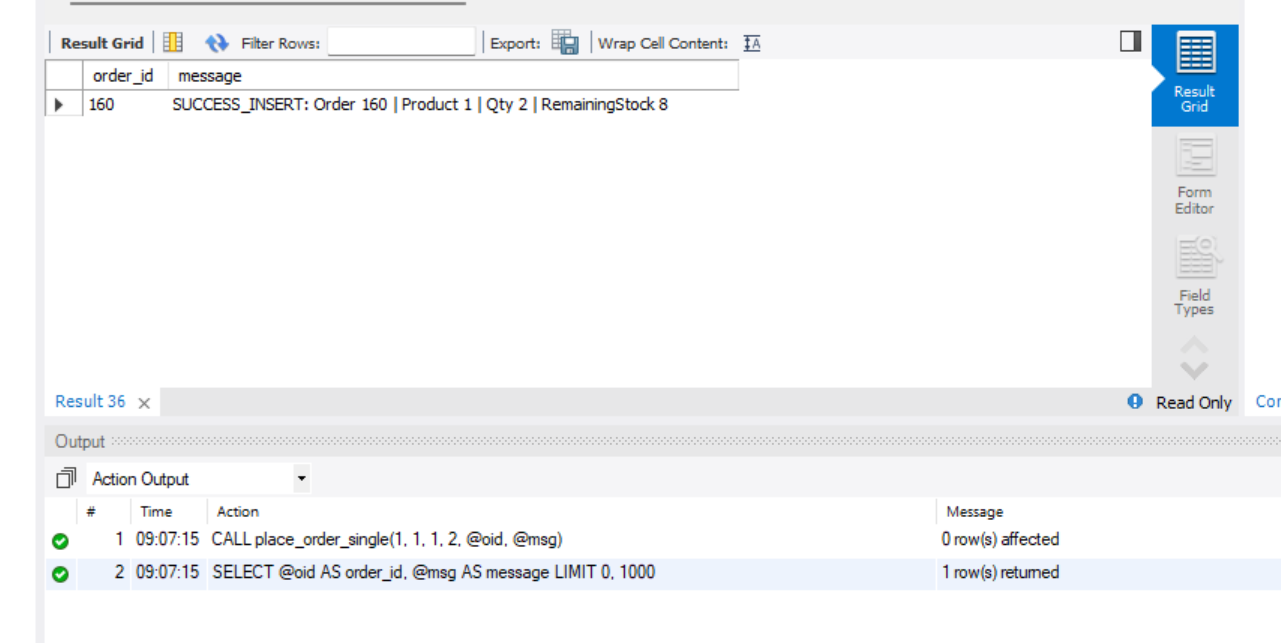


-- Place new order for 2 units

CALL place\_order\_single(1, 1, 1, 2, @oid, @msg);

SELECT @oid AS order\_id, @msg AS message;

**Result:**



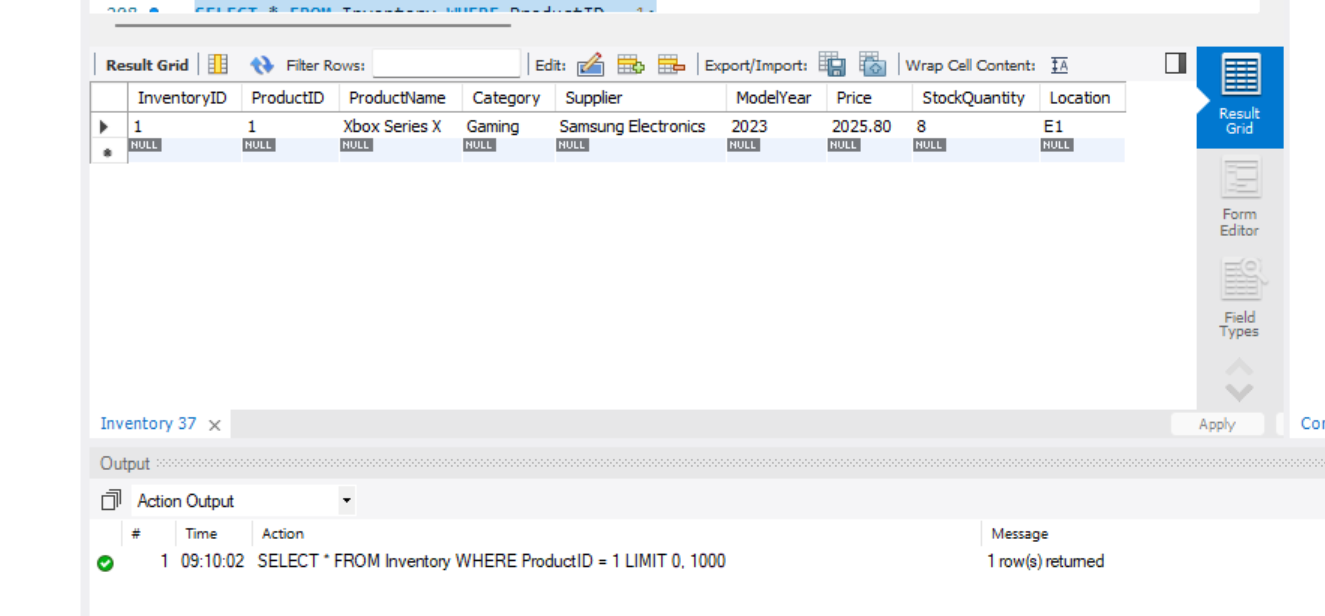
**Expected message (example):**

SUCCESS\_INSERT: Order 160 | product 1 | Qty 2 | RemainingStock 8

-- Verify stock decreased by 2 for ProductID = 1

SELECT \* FROM Inventory WHERE ProductID = 1;

After Result :



### **Business effect / impact**

* Atomicity: The purchase, the inventory decrement, and order detail insert happen as a single atomic unit. If any part fails (e.g., low stock), the procedure rolls back and no partial state remains.
* Data integrity: Prevents overselling because we lock the inventory row (FOR UPDATE) and check stock before committing.

## **Performance Optimization**

### **Case A — “Recent Orders for a Customer” (avoid filesort + table scan)**

**Business question:** “Show a customer’s last 20 orders by date.”

### **1) BEFORE — No composite index for filter+order**

-- This executes and returns real timing

EXPLAIN ANALYZE

SELECT OrderID, OrderDate, Status

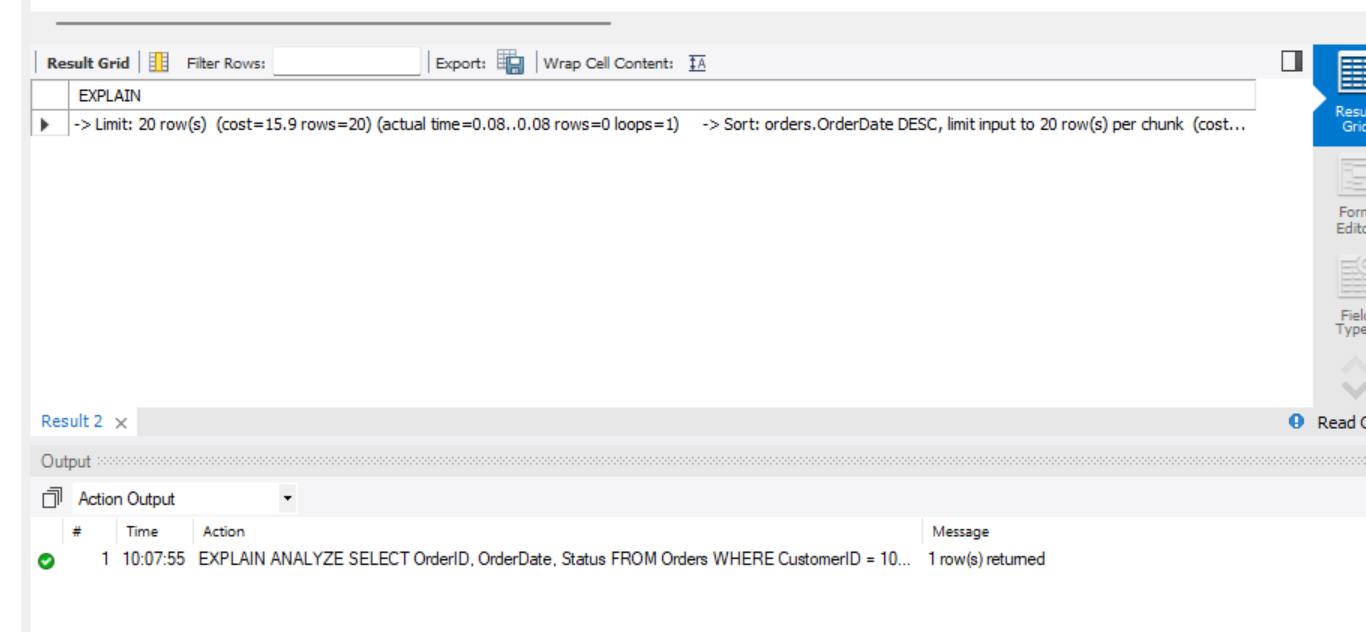
FROM Orders

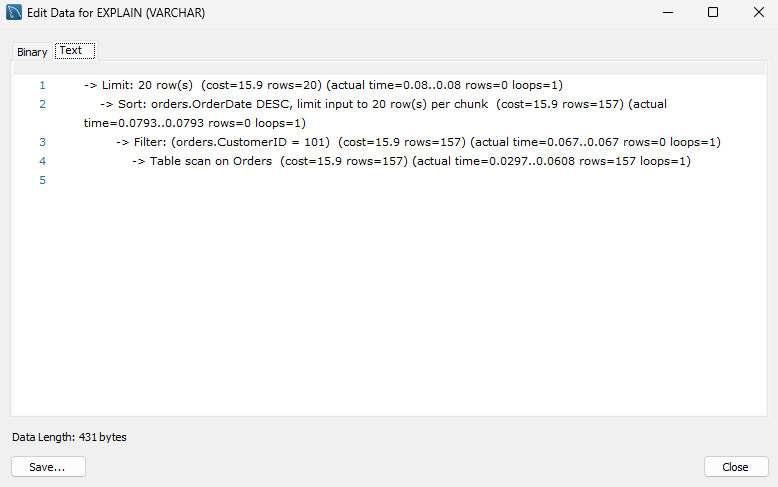
WHERE CustomerID = 101

ORDER BY OrderDate DESC

LIMIT 20;

**Result:**





**Typical BEFORE pattern**

* type: ref (on CustomerID if it has an index) **or** ALL (full scan) if not
* Extra: Using filesort (because it must sort by OrderDate)
* Higher rows estimate

### **Create index (AFTER)**

CREATE INDEX idx\_orders\_customer\_orderdate\_status

ON Orders (CustomerID, OrderDate DESC, OrderID, Status);

-- Rerun with actual timing

EXPLAIN ANALYZE

SELECT OrderID, OrderDate, Status

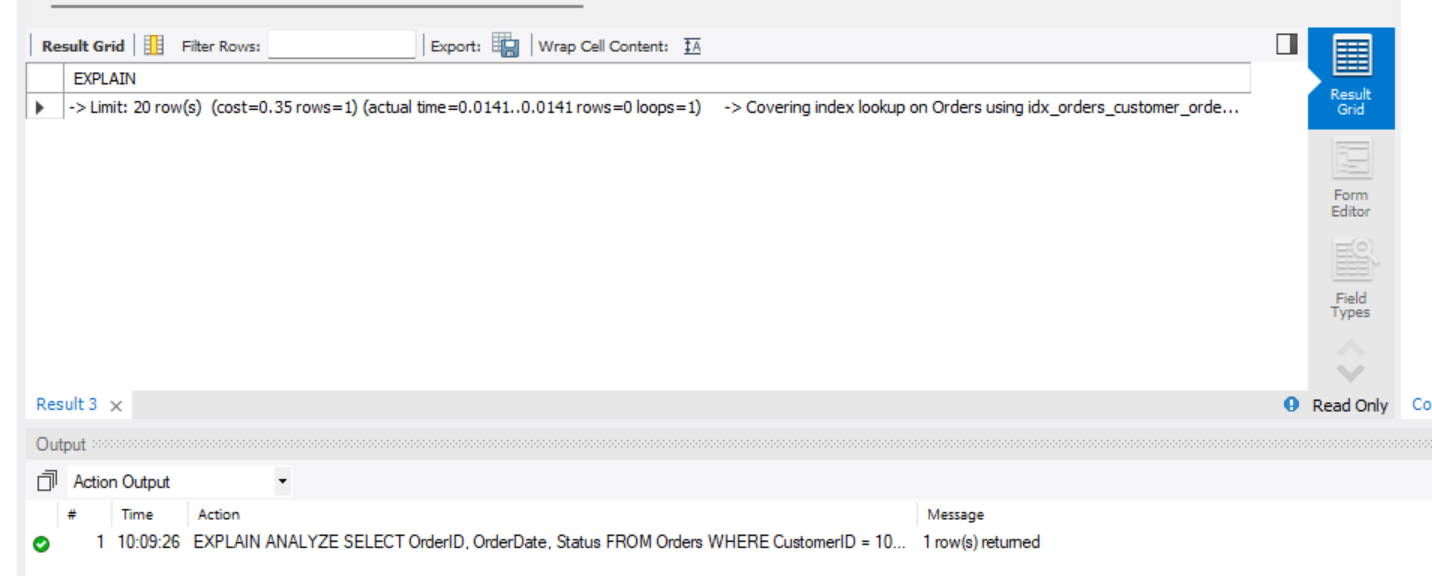
FROM Orders

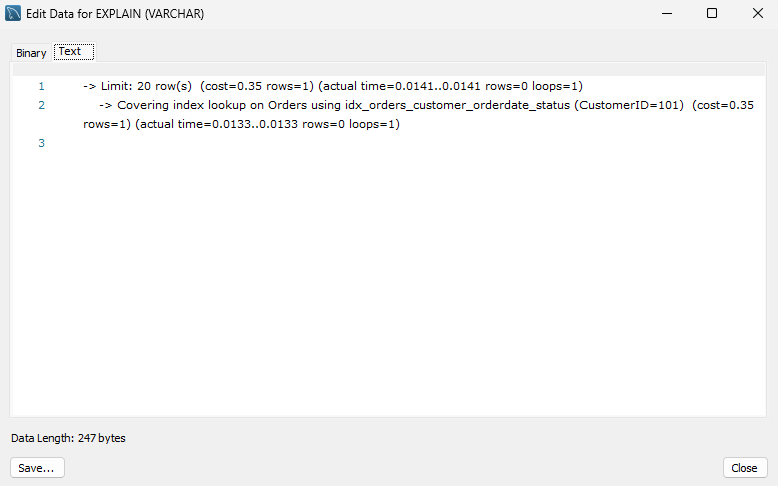
WHERE CustomerID = 101

ORDER BY OrderDate DESC

LIMIT 20;

**Result:**

****

****

**Expected AFTER pattern**

* type: ref or range
* **No Using filesort** (reads in index order)
* Fewer rows examined

**Business effect**

* **Faster “My Orders” page** and customer service lookups
* **Lower CPU/IO** due to index order scan (no extra sort)
* Helps meet **SLA** during peak traffic

**Case B — “Compute Order Total” (covering index → index-only aggregation)**

**Business question:** “Compute the total amount for a single order.”

### **BEFORE (No covering index)**

EXPLAIN ANALYZE

SELECT od.OrderID,

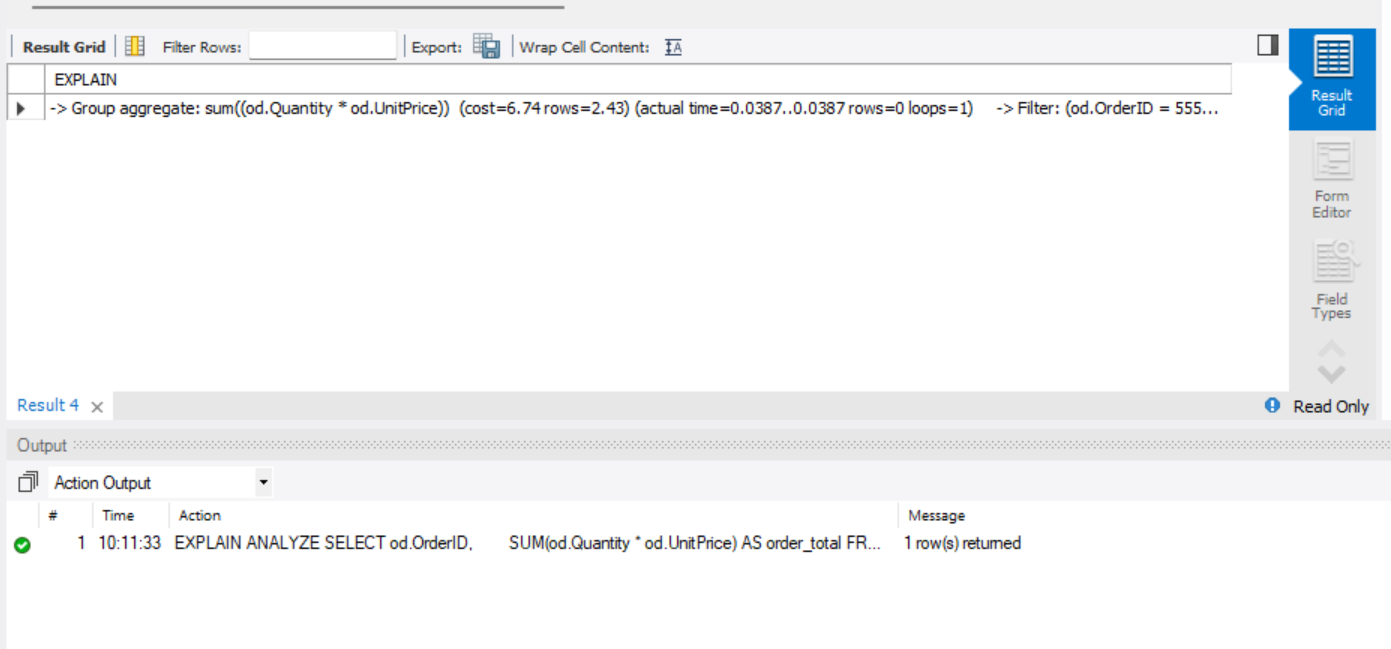
SUM(od.Quantity \* od.UnitPrice) AS order\_total

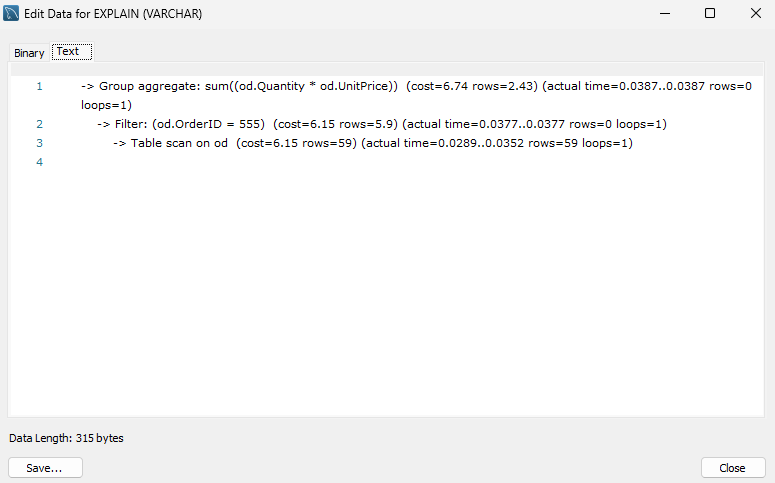
FROM Order\_Details AS od

WHERE od.OrderID = 555

GROUP BY od.OrderID;

**Before Result:**





**Typical BEFORE pattern**

* type: ALL (full scan) or ref (on OrderID)
* For each matching row, the engine still fetches table rows to get Quantity & UnitPrice
* Extra may **not** show “Using index”

### **Create covering index (AFTER)**

CREATE INDEX idx\_orderdetails\_orderid\_qty\_price

ON Order\_Details (OrderID, Quantity, UnitPrice);

EXPLAIN

SELECT od.OrderID,

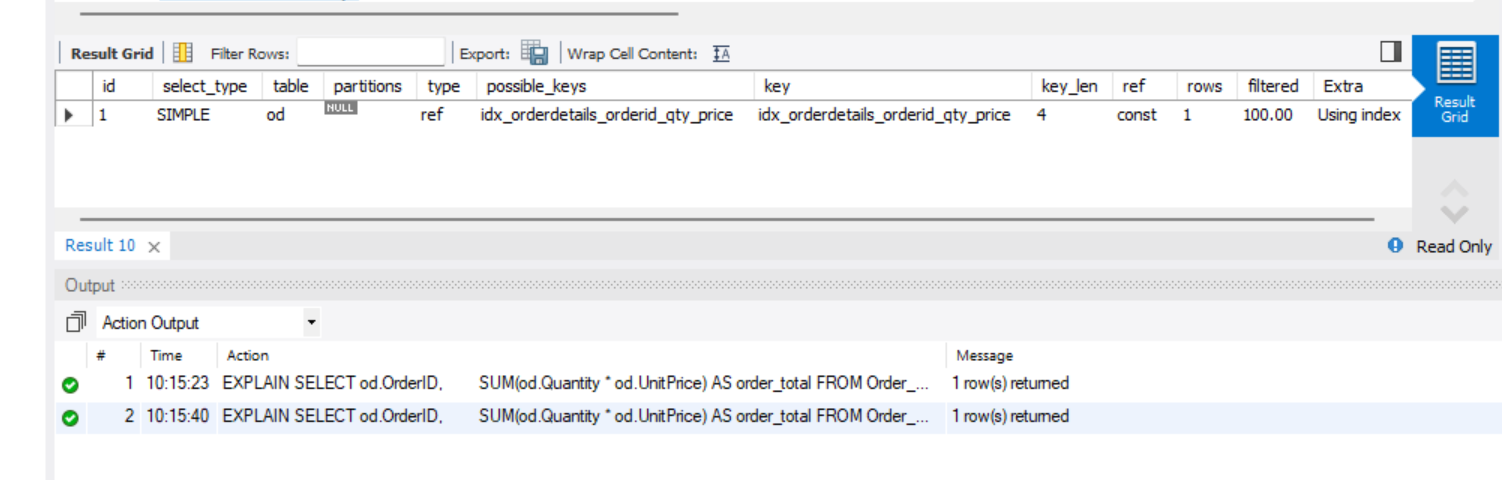
SUM(od.Quantity \* od.UnitPrice) AS order\_total

FROM Order\_Details AS od

WHERE od.OrderID = 555

GROUP BY od.OrderID;

**After Result:**



### 

**Expected AFTER pattern**

* type: range/ref on OrderID
* Extra: Using index (index-only scan)
* Much lower random I/O

**Business effect**

* **Faster order totals** for invoices and dashboards
* Smooths **checkout/CS workflows** (totals appear instantly)
* **Resource savings** → supports more concurrent users

## **Conclusion**

This project demonstrates the **complete lifecycle of a data warehouse system**: schema design, operational queries, analytics, transaction safety, and performance tuning.

For **Best Buy**, this system enables:

* **Operational efficiency:** Fast and safe order processing.
* **Analytical insights:** Product performance, sales trends, customer loyalty.
* **Scalability:** Can handle millions of rows, ensuring reliability during peak sales.

### **Future Improvements**

* Automate ETL pipelines.
* Integrate real-time dashboards.
* Expand into predictive analytics with ML.

### **What we learned from final project of Advanced SQL**

Through this project, we gained hands-on experience applying advanced SQL concepts to design and implement a complete data management solution for a fictional Best Buy Electronic Store. We learned how to build an OLTP system using 3NF to support day-to-day operations such as orders, payments, inventory management, and customer reviews. In parallel, we designed an OLAP star schema to enable efficient reporting and analytics, including trend analysis and sales forecasting.

We developed complex SQL queries involving multi-table joins, aggregations, subqueries, CTEs, and functions to answer real business questions like identifying top-selling products and tracking employee performance. By creating views and simulating a materialized view, we simplified reporting and improved query speed.

We also explored transactions and concurrency control by building a stored procedure that uses START TRANSACTION, COMMIT, and ROLLBACK to ensure atomicity and prevent overselling. This included using row locking (SELECT ... FOR UPDATE) to demonstrate how to avoid race conditions in a multi-user environment.

Finally, we applied performance optimization techniques, such as creating composite and covering indexes, and used EXPLAIN to compare query performance before and after indexing.

Overall, this project strengthened our understanding of database design, query optimization, and enterprise-level SQL practices, showing how OLTP and OLAP systems work together to support both operational efficiency and strategic decision-making.