Inventory Management

Michael Condra with Copilot

[michael.condra@gmail.com](mailto:michael.condra@gmail.com)

February 21, 2025

GitHub repo for this project: <https://github.com/MikeCondra/InventoryManagement.git>

Download this way: git clone https://github.com/MikeCondra/InventoryManagement.git

The repo contains this doc and the MySql dump: SmartShopDB\_20240211\_1409.sql . In MySQL Wizard, the Server/Data Import command will can the database using your own local credentials.

**Grading Criteria Goals**

**Use Copilot to generate basic SQL queries for retrieving and filtering data**

Over 20 stored procedures were created by Copilot: See screenshot: [DB Notebook (My Connection) - EFCoreModelApp3 - Visual Studio Code](https://gyazo.com/31a4d45042a3c45d0298561fc0d78e4f)

**Use Copilot to generate complex SQL queries, including joins, nested queries and aggregations**

Use of these query-optimizing techniques can be found here.

Join: Pages 7-12  
Temporary tables (page 15)  
Nested Queries (page 15)  
Aggregations (pages 26-29)

**Debug errors with Copilot Assistance**

**My bug fix.** Some Copilot-provided queries had errors. I fixed one in MySQL Workbench, where Copilot recommended Dec as a variable name, but Dec is a MySQL reserved word. I changed that to Dece (meaning December). (Page 28).

**Copilot’s bug fix**. Copilot made a stored procedure to populate the Delivery table with random data for delivery stores, quantities, dates, and request dates. The procedure had a bug that left the request-date column null. Copilot fixed it. (Page 23).

**Optimize Query performance using Copilot suggestions**

Starting with a complex query that reported changes to product levels due to delivery and sale events, Copilot optimized it from 23 to 5 msec using these techniques: removing cursors, adding indices, adding joins.

Call RunningPerStoreInventoryChanges(); -- **23** msec (page 19); Unoptimized.  
Call RunningPerStoreInventoryChanges(); 23 msec (page 20) (Indices added)  
Call RunningPerStoreInventoryChanges\_NoCursors(); – 7 msec (page 23) (Cursors removed)  
Call RunningPerStoreInventoryChanges\_NoCursors\_WithJoin(); -- **5** msec (page 24) (Joins)

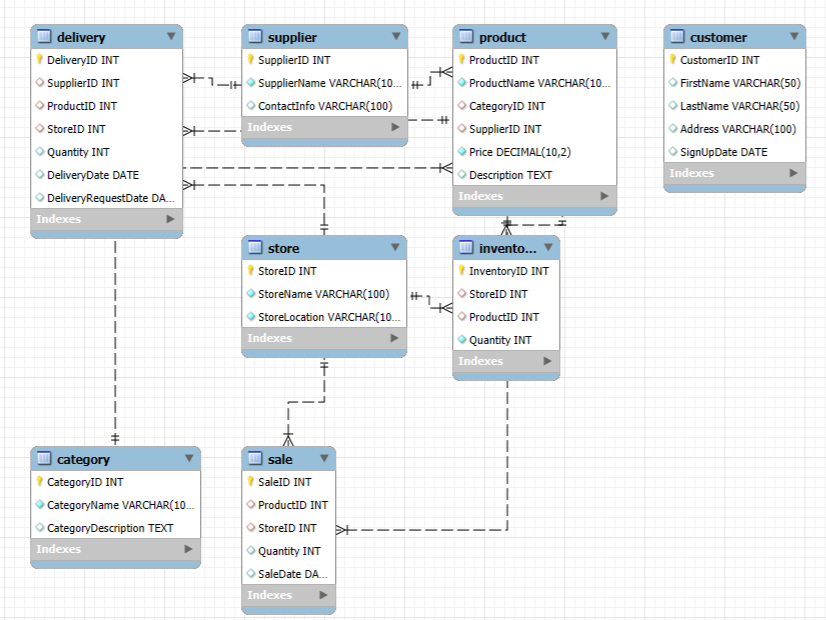
The prompt to create the complex query:

Please make a Stored Procedure “RunningPerStoreInventoryChanges” that looks at tables Delivery and Sale, and reports the inventory in each store as it changes over time. So, the first (sorted) column in the output should be date, the second column whether a sale or delivery occurred, then what store was affected, then what productid was affected, then the quantity change, then the current inventory for that productid/storied.

**Include a brief summary of how Copilot assisted**

This 29-page log contains of most of the interaction with Copilot. In brief, Copilot created a basic set of tables from a short prompt, then populated data tables with random data, then created a long list of stored procedures. Then, I asked Copilot to create more tables, Sale and Delivery, to model changes to inventory over time. Copilot built those, filled them with a couple hundred rows. Then, I asked for a (hopefully) complex query to report inventory changes over time. That query took 23 msec. Then, Copilot successfully optimized it, using the recommended strategies, and reduced the time to 5 msec. It’s in the log.

.

DATABASE STRUCTURE

COPILOT LOG

|  |  |
| --- | --- |
|  | **ACTIVITY 1 – create tables and populate with sample data.** |
| Hello. For a school project, this is the start of a discussion about defining the structure of a database that will model the inventory of a fictional retail business. The business has a multiple stores, each with inventory. Each product has one supplier. Products have a category, so there must be a category table. In broad terms, what would you suggest? | Hello! That sounds like an exciting project! Here's a high-level suggestion for your database structure:  **Tables and Their Relations**   1. **Store**    * StoreID (Primary Key)    * StoreName    * StoreLocation    * Other relevant store details 2. **Product**    * ProductID (Primary Key)    * ProductName    * CategoryID (Foreign Key from Category table)    * SupplierID (Foreign Key from Supplier table)    * Price    * Description    * Other relevant product details 3. **Supplier**    * SupplierID (Primary Key)    * SupplierName    * ContactInfo    * Other relevant supplier details 4. **Category**    * CategoryID (Primary Key)    * CategoryName    * CategoryDescription    * Other relevant category details 5. **Inventory**    * InventoryID (Primary Key)    * StoreID (Foreign Key from Store table)    * ProductID (Foreign Key from Product table)    * Quantity    * Other relevant inventory details   **Relationships and Example Queries**   * **Relationships:**   + A **store** has multiple **inventory** items.   + A **product** belongs to one **category** and one **supplier**.   + Each **inventory** item is associated with a specific **store** and **product**. * **Example Query:**   + To find the total quantity of a specific product across all stores:   SELECT ProductID, SUM(Quantity) AS TotalQuantity  FROM Inventory  WHERE ProductID = [specific\_product\_id]  GROUP BY ProductID;  **Diagram Representation**  If you need a visual representation, you can create an **Entity-Relationship Diagram (ERD)**. This helps to visualize tables and relationships better.  What do you think? Any additional details or specific features you want to include? |
| If the target is MySql on a PC, would you be able to generate the table definitions, plus some sample data? | Absolutely! Here are the MySQL table definitions and some sample data for your fictional retail business:  **Table Definitions**  -- Store Table  CREATE TABLE Store (  StoreID INT AUTO\_INCREMENT PRIMARY KEY,  StoreName VARCHAR(100) NOT NULL,  StoreLocation VARCHAR(100) NOT NULL  );  -- Product Table  CREATE TABLE Product (  ProductID INT AUTO\_INCREMENT PRIMARY KEY,  ProductName VARCHAR(100) NOT NULL,  CategoryID INT,  SupplierID INT,  Price DECIMAL(10, 2) NOT NULL,  Description TEXT,  FOREIGN KEY (CategoryID) REFERENCES Category(CategoryID),  FOREIGN KEY (SupplierID) REFERENCES Supplier(SupplierID)  );  -- Supplier Table  CREATE TABLE Supplier (  SupplierID INT AUTO\_INCREMENT PRIMARY KEY,  SupplierName VARCHAR(100) NOT NULL,  ContactInfo VARCHAR(100)  );  -- Category Table  CREATE TABLE Category (  CategoryID INT AUTO\_INCREMENT PRIMARY KEY,  CategoryName VARCHAR(100) NOT NULL,  CategoryDescription TEXT  );  -- Inventory Table  CREATE TABLE Inventory (  InventoryID INT AUTO\_INCREMENT PRIMARY KEY,  StoreID INT,  ProductID INT,  Quantity INT NOT NULL,  FOREIGN KEY (StoreID) REFERENCES Store(StoreID),  FOREIGN KEY (ProductID) REFERENCES Product(ProductID)  );  **Sample Data**  -- Insert into Store  INSERT INTO Store (StoreName, StoreLocation) VALUES  ('Main Street Store', '123 Main St, Tracy, CA'),  ('Second Street Store', '456 Second St, Tracy, CA');  -- Insert into Supplier  INSERT INTO Supplier (SupplierName, ContactInfo) VALUES  ('Supplier One', 'contact@supplierone.com'),  ('Supplier Two', 'contact@suppliertwo.com');  -- Insert into Category  INSERT INTO Category (CategoryName, CategoryDescription) VALUES  ('Electronics', 'Electronic items and gadgets'),  ('Clothing', 'Apparel and accessories');  -- Insert into Product  INSERT INTO Product (ProductName, CategoryID, SupplierID, Price, Description) VALUES  ('Smartphone', 1, 1, 299.99, 'Latest model smartphone with 128GB storage'),  ('T-Shirt', 2, 2, 19.99, 'Comfortable cotton t-shirt in various sizes');  -- Insert into Inventory  INSERT INTO Inventory (StoreID, ProductID, Quantity) VALUES  (1, 1, 50), -- 50 Smartphones in Main Street Store  (1, 2, 100), -- 100 T-Shirts in Main Street Store  (2, 1, 30), -- 30 Smartphones in Second Street Store  (2, 2, 80); -- 80 T-Shirts in Second Street StoreThis should give you a solid foundation to start with. You can expand and customize the data as needed for your project. Let me know if there's anything else you need or if you have specific requirements! |
| For the sample data, could you have perhaps 20 products? This would mean revising the "insert" statements into Product and Inventory tables. Some stores should have 0 inventory in a couple of items. | Sure thing! Here's an updated version with 20 products and a variety of inventory levels, including some stores having zero inventory for certain items.  **Sample Data**  -- Insert into Store  INSERT INTO Store (StoreName, StoreLocation) VALUES  ('Main Street Store', '123 Main St, Tracy, CA'),  ('Second Street Store', '456 Second St, Tracy, CA');  -- Insert into Supplier  INSERT INTO Supplier (SupplierName, ContactInfo) VALUES  ('Supplier One', 'contact@supplierone.com'),  ('Supplier Two', 'contact@suppliertwo.com'),  ('Supplier Three', 'contact@supplierthree.com');  -- Insert into Category  INSERT INTO Category (CategoryName, CategoryDescription) VALUES  ('Electronics', 'Electronic items and gadgets'),  ('Clothing', 'Apparel and accessories'),  ('Home Appliances', 'Household appliances and gadgets'),  ('Toys', 'Toys and games for children'),  ('Books', 'Various genres of books');  -- Insert into Product  INSERT INTO Product (ProductName, CategoryID, SupplierID, Price, Description) VALUES  ('Smartphone', 1, 1, 299.99, 'Latest model smartphone with 128GB storage'),  ('Laptop', 1, 1, 999.99, 'High-performance laptop with 16GB RAM'),  ('Headphones', 1, 2, 49.99, 'Noise-cancelling over-ear headphones'),  ('T-Shirt', 2, 2, 19.99, 'Comfortable cotton t-shirt in various sizes'),  ('Jeans', 2, 2, 39.99, 'Stylish denim jeans'),  ('Microwave', 3, 3, 89.99, 'Compact microwave oven with digital controls'),  ('Refrigerator', 3, 3, 499.99, 'Energy-efficient refrigerator with large capacity'),  ('Toaster', 3, 1, 29.99, 'Two-slice toaster with adjustable settings'),  ('Toy Car', 4, 2, 9.99, 'Remote control toy car'),  ('Doll', 4, 3, 14.99, 'Fashionable doll with accessories'),  ('Puzzle', 4, 1, 12.99, '1000-piece jigsaw puzzle'),  ('Children\'s Book', 5, 1, 6.99, 'Illustrated children\'s storybook'),  ('Novel', 5, 2, 15.99, 'Bestselling fiction novel'),  ('Cookbook', 5, 3, 24.99, 'Gourmet cookbook with recipes'),  ('Smartwatch', 1, 1, 199.99, 'Fitness-tracking smartwatch'),  ('Bluetooth Speaker', 1, 2, 59.99, 'Portable Bluetooth speaker with excellent sound quality'),  ('Jacket', 2, 2, 79.99, 'Waterproof and windproof jacket'),  ('Blender', 3, 3, 69.99, 'High-speed blender with multiple settings'),  ('Board Game', 4, 2, 29.99, 'Popular strategy board game'),  ('Non-Fiction Book', 5, 3, 18.99, 'Informative non-fiction book on various topics');  -- Insert into Inventory  INSERT INTO Inventory (StoreID, ProductID, Quantity) VALUES  (1, 1, 50), -- 50 Smartphones in Main Street Store  (1, 2, 30), -- 30 Laptops in Main Street Store  (1, 3, 100), -- 100 Headphones in Main Street Store  (1, 4, 150), -- 150 T-Shirts in Main Street Store  (1, 5, 80), -- 80 Jeans in Main Street Store  (1, 6, 20), -- 20 Microwaves in Main Street Store  (1, 7, 15), -- 15 Refrigerators in Main Street Store  (1, 8, 60), -- 60 Toasters in Main Street Store  (1, 9, 90), -- 90 Toy Cars in Main Street Store  (1, 10, 70), -- 70 Dolls in Main Street Store  (1, 11, 50), -- 50 Puzzles in Main Street Store  (1, 12, 100), -- 100 Children's Books in Main Street Store  (1, 13, 40), -- 40 Novels in Main Street Store  (1, 14, 30), -- 30 Cookbooks in Main Street Store  (1, 15, 60), -- 60 Smartwatches in Main Street Store  (1, 16, 80), -- 80 Bluetooth Speakers in Main Street Store  (1, 17, 70), -- 70 Jackets in Main Street Store  (1, 18, 25), -- 25 Blenders in Main Street Store  (1, 19, 40), -- 40 Board Games in Main Street Store  (1, 20, 50), -- 50 Non-Fiction Books in Main Street Store  (2, 1, 0), -- 0 Smartphones in Second Street Store  (2, 2, 25), -- 25 Laptops in Second Street Store  (2, 3, 40), -- 40 Headphones in Second Street Store  (2, 4, 100), -- 100 T-Shirts in Second Street Store  (2, 5, 70), -- 70 Jeans in Second Street Store  (2, 6, 10), -- 10 Microwaves in Second Street Store  (2, 7, 0), -- 0 Refrigerators in Second Street Store  (2, 8, 50), -- 50 Toasters in Second Street Store  (2, 9, 60), -- 60 Toy Cars in Second Street Store  (2, 10, 40), -- 40 Dolls in Second Street Store  (2, 11, 30), -- 30 Puzzles in Second Street Store  (2, 12, 50), -- 50 Children's Books in Second Street Store  (2, 13, 20), -- 20 Novels in Second Street Store  (2, 14, 0), -- 0 Cookbooks in Second Street Store  (2, 15, 30), -- 30 Smartwatches in Second Street Store  (2, 16, 40), -- 40 Bluetooth Speakers in Second Street Store  (2, 17, 50), -- 50 Jackets in Second Street Store  (2, 18, 20), -- 20 Blenders in Second Street Store  (2, 19, 60), -- 60 Board Games in Second Street Store  (2, 20, 30); -- 30 Non-Fiction Books in Second Street Store |
| Write MySQL SQL code to create several stored procedures that implement SELECT statements about the current database. | DELIMITER $$  CREATE PROCEDURE GetProductDetails(IN productId INT)  BEGIN  SELECT  p.ProductID,  p.ProductName,  c.CategoryName,  s.SupplierName,  p.Price,  p.Description  FROM  Product p  JOIN  Category c ON p.CategoryID = c.CategoryID  JOIN  Supplier s ON p.SupplierID = s.SupplierID  WHERE  p.ProductID = productId;  END$$  DELIMITER ;  DELIMITER $$  CREATE PROCEDURE GetInventoryByStore(IN storeId INT)  BEGIN  SELECT  i.InventoryID,  s.StoreName,  p.ProductName,  i.Quantity  FROM  Inventory i  JOIN  Store s ON i.StoreID = s.StoreID  JOIN  Product p ON i.ProductID = p.ProductID  WHERE  i.StoreID = storeId;  END$$  DELIMITER ;  DELIMITER $$  CREATE PROCEDURE GetProductsByCategory(IN categoryId INT)  BEGIN  SELECT  p.ProductID,  p.ProductName,  p.Price,  p.Description,  s.SupplierName  FROM  Product p  JOIN  Supplier s ON p.SupplierID = s.SupplierID  WHERE  p.CategoryID = categoryId;  END$$  DELIMITER ;  DELIMITER $$  CREATE PROCEDURE GetAllSuppliers()  BEGIN  SELECT  SupplierID,  SupplierName,  ContactInfo  FROM  Supplier;  END$$  DELIMITER ;  DELIMITER $$  CREATE PROCEDURE GetInventoryForProduct(IN productId INT)  BEGIN  SELECT  i.InventoryID,  s.StoreName,  p.ProductName,  i.Quantity  FROM  Inventory i  JOIN  Store s ON i.StoreID = s.StoreID  JOIN  Product p ON i.ProductID = p.ProductID  WHERE  i.ProductID = productId;  END$$  DELIMITER ; |
| Thank you. Please write versions of the above stored procedures that implement filtering based on categories and availability, that is, inventory levels greater than zero.  May I ask that you use different names for the procedures that take input arguments, such as GetProductDetailsByProductId instead of GetProductDetails. | DELIMITER $$  CREATE PROCEDURE GetProductDetailsByProductId(IN productId INT)  BEGIN  SELECT  p.ProductID,  p.ProductName,  c.CategoryName,  s.SupplierName,  p.Price,  p.Description  FROM  Product p  JOIN  Category c ON p.CategoryID = c.CategoryID  JOIN  Supplier s ON p.SupplierID = s.SupplierID  JOIN  Inventory i ON p.ProductID = i.ProductID  WHERE  p.ProductID = productId  AND i.Quantity > 0;  END$$  DELIMITER ;  DELIMITER $$  CREATE PROCEDURE GetInventoryByStoreId(IN storeId INT)  BEGIN  SELECT  i.InventoryID,  s.StoreName,  p.ProductName,  i.Quantity  FROM  Inventory i  JOIN  Store s ON i.StoreID = s.StoreID  JOIN  Product p ON i.ProductID = p.ProductID  WHERE  i.StoreID = storeId  AND i.Quantity > 0;  END$$  DELIMITER ;  DELIMITER $$  CREATE PROCEDURE GetProductsByCategoryId(IN categoryId INT)  BEGIN  SELECT  p.ProductID,  p.ProductName,  p.Price,  p.Description,  s.SupplierName  FROM  Product p  JOIN  Supplier s ON p.SupplierID = s.SupplierID  JOIN  Inventory i ON p.ProductID = i.ProductID  WHERE  p.CategoryID = categoryId  AND i.Quantity > 0;  END$$  DELIMITER ;  DELIMITER $$  CREATE PROCEDURE GetAllSuppliers()  BEGIN  SELECT  SupplierID,  SupplierName,  ContactInfo  FROM  Supplier;  END$$  DELIMITER ;  DELIMITER $$  CREATE PROCEDURE GetInventoryForProductId(IN productId INT)  BEGIN  SELECT  i.InventoryID,  s.StoreName,  p.ProductName,  i.Quantity  FROM  Inventory i  JOIN  Store s ON i.StoreID = s.StoreID  JOIN  Product p ON i.ProductID = p.ProductID  WHERE  i.ProductID = productId  AND i.Quantity > 0;  END$$  DELIMITER ; |
| Thank you; that was excellent. May I ask you write versions of the stored procedures that implement filtering, and use "FilteredBy..." as part of the stored procedure names. | DELIMITER $$  CREATE PROCEDURE GetProductDetailsFilteredByProductId(IN productId INT)  BEGIN  SELECT  p.ProductID,  p.ProductName,  c.CategoryName,  s.SupplierName,  p.Price,  p.Description  FROM  Product p  JOIN  Category c ON p.CategoryID = c.CategoryID  JOIN  Supplier s ON p.SupplierID = s.SupplierID  JOIN  Inventory i ON p.ProductID = i.ProductID  WHERE  p.ProductID = productId  AND i.Quantity > 0;  END$$  DELIMITER ;  DELIMITER $$  CREATE PROCEDURE GetInventoryFilteredByStoreId(IN storeId INT)  BEGIN  SELECT  i.InventoryID,  s.StoreName,  p.ProductName,  i.Quantity  FROM  Inventory i  JOIN  Store s ON i.StoreID = s.StoreID  JOIN  Product p ON i.ProductID = p.ProductID  WHERE  i.StoreID = storeId  AND i.Quantity > 0;  END$$  DELIMITER ;  DELIMITER $$  CREATE PROCEDURE GetProductsFilteredByCategoryId(IN categoryId INT)  BEGIN  SELECT  p.ProductID,  p.ProductName,  p.Price,  p.Description,  s.SupplierName  FROM  Product p  JOIN  Supplier s ON p.SupplierID = s.SupplierID  JOIN  Inventory i ON p.ProductID = i.ProductID  WHERE  p.CategoryID = categoryId  AND i.Quantity > 0;  END$$  DELIMITER ;  DELIMITER $$  CREATE PROCEDURE GetAllSuppliers()  BEGIN  SELECT  SupplierID,  SupplierName,  ContactInfo  FROM  Supplier;  END$$  DELIMITER ;  DELIMITER $$  CREATE PROCEDURE GetInventoryFilteredByProductId(IN productId INT)  BEGIN  SELECT  i.InventoryID,  s.StoreName,  p.ProductName,  i.Quantity  FROM  Inventory i  JOIN  Store s ON i.StoreID = s.StoreID  JOIN  Product p ON i.ProductID = p.ProductID  WHERE  i.ProductID = productId  AND i.Quantity > 0;  END$$  DELIMITER ; |
| Thank you. May I ask that you write versions of the last set of stored procedures that implement filtering by an important field, and append "SortedBy..." to the stored procedure name.  Comment: My prompt did not ask the result to be sorted, but Copilot inferred this correctly. I only asked that “SortedBy…” be incorporated into the procedure names. | DELIMITER $$  CREATE PROCEDURE GetProductDetailsFilteredByProductIdSortedByProductName(IN productId INT)  BEGIN  SELECT  p.ProductID,  p.ProductName,  c.CategoryName,  s.SupplierName,  p.Price,  p.Description  FROM  Product p  JOIN  Category c ON p.CategoryID = c.CategoryID  JOIN  Supplier s ON p.SupplierID = s.SupplierID  JOIN  Inventory i ON p.ProductID = i.ProductID  WHERE  p.ProductID = productId  AND i.Quantity > 0  ORDER BY  p.ProductName;  END$$  DELIMITER ;  DELIMITER $$  CREATE PROCEDURE GetInventoryFilteredByStoreIdSortedByProductName(IN storeId INT)  BEGIN  SELECT  i.InventoryID,  s.StoreName,  p.ProductName,  i.Quantity  FROM  Inventory i  JOIN  Store s ON i.StoreID = s.StoreID  JOIN  Product p ON i.ProductID = p.ProductID  WHERE  i.StoreID = storeId  AND i.Quantity > 0  ORDER BY  p.ProductName;  END$$  DELIMITER ;  DELIMITER $$  CREATE PROCEDURE GetProductsFilteredByCategoryIdSortedByProductName(IN categoryId INT)  BEGIN  SELECT  p.ProductID,  p.ProductName,  p.Price,  p.Description,  s.SupplierName  FROM  Product p  JOIN  Supplier s ON p.SupplierID = s.SupplierID  JOIN  Inventory i ON p.ProductID = i.ProductID  WHERE  p.CategoryID = categoryId  AND i.Quantity > 0  ORDER BY  p.ProductName;  END$$  DELIMITER ;  DELIMITER $$  CREATE PROCEDURE GetAllSuppliersSortedBySupplierName()  BEGIN  SELECT  SupplierID,  SupplierName,  ContactInfo  FROM  Supplier  ORDER BY  SupplierName;  END$$  DELIMITER ;  DELIMITER $$  CREATE PROCEDURE GetInventoryFilteredByProductIdSortedByStoreName(IN productId INT)  BEGIN  SELECT  i.InventoryID,  s.StoreName,  p.ProductName,  i.Quantity  FROM  Inventory i  JOIN  Store s ON i.StoreID = s.StoreID  JOIN  Product p ON i.ProductID = p.ProductID  WHERE  i.ProductID = productId  AND i.Quantity > 0  ORDER BY  s.StoreName;  END$$  DELIMITER ; |
| (Dumping database to external media) | * + - Mysqldump -u root -p smartshopdb > c:$$coursera$$smartshopdb\_20250221.sql |
| **ACTIVITY 2** |  |
| Thank you. Create a table “Delivery” with columns SupplierID, ProductID, StoreID, Quantity and Date, to model historical deliveries of product by suppliers (supplierIds are 1, 2, 3), of products (productIds are 1 thru 20), and stores receiving delivery (storeIds are 1, 2). Then, create SQL to import sample data of about 200 deliveries of random amounts in the range of 1-20 units per item per delivery, covering random dates in 2024. | CREATE TABLE Delivery (  DeliveryID INT AUTO\_INCREMENT PRIMARY KEY,  SupplierID INT,  ProductID INT,  StoreID INT,  Quantity INT,  DeliveryDate DATE,  FOREIGN KEY (SupplierID) REFERENCES Supplier(SupplierID),  FOREIGN KEY (ProductID) REFERENCES Product(ProductID),  FOREIGN KEY (StoreID) REFERENCES Store(StoreID)  );  -- Generate 200 sample deliveries  DELIMITER $$  CREATE PROCEDURE GenerateSampleDeliveries()  BEGIN  DECLARE i INT DEFAULT 1;  DECLARE maxDeliveries INT DEFAULT 200;  WHILE i <= maxDeliveries DO  INSERT INTO Delivery (SupplierID, ProductID, StoreID, Quantity, DeliveryDate)  VALUES (  FLOOR(1 + (RAND() \* 3)), -- Random SupplierID between 1 and 3  FLOOR(1 + (RAND() \* 20)), -- Random ProductID between 1 and 20  FLOOR(1 + (RAND() \* 2)), -- Random StoreID between 1 and 2  FLOOR(1 + (RAND() \* 20)), -- Random Quantity between 1 and 20  DATE\_ADD('2024-01-01', INTERVAL FLOOR(RAND() \* 365) DAY) -- Random date in 2024  );  SET i = i + 1;  END WHILE;  END$$  DELIMITER ;  -- Call the procedure to generate the sample deliveries  CALL GenerateSampleDeliveries(); |
| Thank you. Please make MySql SQL to create a new table, Customer, with these fields: unique CustomerID, FirstName, LastName, Address, SignUpDate. Next, make up SQL to load the table with 20 customers with random First and Last names, and sign-up dates sometime in 2023. | CREATE TABLE Customer (  CustomerID INT AUTO\_INCREMENT PRIMARY KEY,  FirstName VARCHAR(50),  LastName VARCHAR(50),  Address VARCHAR(100),  SignUpDate DATE  );  -- Insert sample data into Customer table  INSERT INTO Customer (FirstName, LastName, Address, SignUpDate) VALUES  ('John', 'Smith', '123 Elm St, Tracy, CA', DATE\_ADD('2023-01-01', INTERVAL FLOOR(RAND() \* 365) DAY)),  ('Jane', 'Doe', '456 Oak St, Tracy, CA', DATE\_ADD('2023-01-01', INTERVAL FLOOR(RAND() \* 365) DAY)),  ('Alice', 'Johnson', '789 Pine St, Tracy, CA', DATE\_ADD('2023-01-01', INTERVAL FLOOR(RAND() \* 365) DAY)),  ('Bob', 'Brown', '101 Maple St, Tracy, CA', DATE\_ADD('2023-01-01', INTERVAL FLOOR(RAND() \* 365) DAY)),  ('Charlie', 'Davis', '202 Cedar St, Tracy, CA', DATE\_ADD('2023-01-01', INTERVAL FLOOR(RAND() \* 365) DAY)),  ('Diana', 'Evans', '303 Spruce St, Tracy, CA', DATE\_ADD('2023-01-01', INTERVAL FLOOR(RAND() \* 365) DAY)),  ('Edward', 'Garcia', '404 Birch St, Tracy, CA', DATE\_ADD('2023-01-01', INTERVAL FLOOR(RAND() \* 365) DAY)),  ('Fiona', 'Harris', '505 Walnut St, Tracy, CA', DATE\_ADD('2023-01-01', INTERVAL FLOOR(RAND() \* 365) DAY)),  ('George', 'Martinez', '606 Chestnut St, Tracy, CA', DATE\_ADD('2023-01-01', INTERVAL FLOOR(RAND() \* 365) DAY)),  ('Hannah', 'Wilson', '707 Redwood St, Tracy, CA', DATE\_ADD('2023-01-01', INTERVAL FLOOR(RAND() \* 365) DAY)),  ('Ivan', 'Moore', '808 Cypress St, Tracy, CA', DATE\_ADD('2023-01-01', INTERVAL FLOOR(RAND() \* 365) DAY)),  ('Jasmine', 'Thomas', '909 Ash St, Tracy, CA', DATE\_ADD('2023-01-01', INTERVAL FLOOR(RAND() \* 365) DAY)),  ('Kevin', 'White', '1010 Magnolia St, Tracy, CA', DATE\_ADD('2023-01-01', INTERVAL FLOOR(RAND() \* 365) DAY)),  ('Laura', 'Clark', '1111 Poplar St, Tracy, CA', DATE\_ADD('2023-01-01', INTERVAL FLOOR(RAND() \* 365) DAY)),  ('Michael', 'Lewis', '1212 Fir St, Tracy, CA', DATE\_ADD('2023-01-01', INTERVAL FLOOR(RAND() \* 365) DAY)),  ('Nina', 'Walker', '1313 Alder St, Tracy, CA', DATE\_ADD('2023-01-01', INTERVAL FLOOR(RAND() \* 365) DAY)),  ('Oscar', 'Hall', '1414 Juniper St, Tracy, CA', DATE\_ADD('2023-01-01', INTERVAL FLOOR(RAND() \* 365) DAY)),  ('Paula', 'Allen', '1515 Willow St, Tracy, CA', DATE\_ADD('2023-01-01', INTERVAL FLOOR(RAND() \* 365) DAY)),  ('Quinn', 'Young', '1616 Sequoia St, Tracy, CA', DATE\_ADD('2023-01-01', INTERVAL FLOOR(RAND() \* 365) DAY)),  ('Rachel', 'King', '1717 Dogwood St, Tracy, CA', DATE\_ADD('2023-01-01', INTERVAL FLOOR(RAND() \* 365) DAY)); |
|  |  |
| Thank you. Please create MySql SQL for a new table “Sale” with these columns: unique SaleID, ProductID, StoreID, SaleDate. Then, make up some sales data for ProductId (1-20), StoreID (1-2), Quantity (in 1-10), and SaleDate(random in 2024). Have 200 sale table records. | CREATE TABLE Sale (  SaleID INT AUTO\_INCREMENT PRIMARY KEY,  ProductID INT,  StoreID INT,  Quantity INT,  SaleDate DATE,  FOREIGN KEY (ProductID) REFERENCES Product(ProductID),  FOREIGN KEY (StoreID) REFERENCES Store(StoreID)  );  -- Generate 200 sample sales  DELIMITER $$  CREATE PROCEDURE GenerateSampleSales()  BEGIN  DECLARE i INT DEFAULT 1;  DECLARE maxSales INT DEFAULT 200;  WHILE i <= maxSales DO  INSERT INTO Sale (ProductID, StoreID, Quantity, SaleDate)  VALUES (  FLOOR(1 + (RAND() \* 20)), -- Random ProductID between 1 and 20  FLOOR(1 + (RAND() \* 2)), -- Random StoreID between 1 and 2  FLOOR(1 + (RAND() \* 10)), -- Random Quantity between 1 and 10  DATE\_ADD('2024-01-01', INTERVAL FLOOR(RAND() \* 365) DAY) -- Random date in 2024  );  SET i = i + 1;  END WHILE;  END$$  DELIMITER ;  -- Call the procedure to generate the sample sales  CALL GenerateSampleSales(); |
| Thank you so much. Please make a Stored Procedure “RunningPerStoreInventoryChanges” that looks at tables Delivery and Sale, and reports the inventory in each store as it changes over time. So, the first (sorted) column in the output should be date, the second column whether a sale or delivery occurred, then what store was affected, then what productid was affected, then the quantity change, then the current inventory for that productid/storied. Please ignore the Inventory table for now. | DELIMITER $$  CREATE PROCEDURE RunningPerStoreInventoryChanges()  BEGIN  -- Create a temporary table to store the running inventory  CREATE TEMPORARY TABLE TempInventory (  StoreID INT,  ProductID INT,  Inventory INT DEFAULT 0  );  -- Populate initial inventory values from deliveries  INSERT INTO TempInventory (StoreID, ProductID, Inventory)  SELECT StoreID, ProductID, SUM(Quantity)  FROM Delivery  GROUP BY StoreID, ProductID;  -- Declare cursor for combined deliveries and sales  DECLARE done INT DEFAULT FALSE;  DECLARE cur CURSOR FOR  **SELECT** 'Delivery' AS Event, DeliveryDate AS EventDate, StoreID, ProductID, Quantity  FROM Delivery  UNION ALL  **SELECT** 'Sale' AS Event, SaleDate AS EventDate, StoreID, ProductID, -Quantity  FROM Sale  ORDER BY EventDate;  DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = TRUE;  -- Declare variables to store cursor data  DECLARE event VARCHAR(10);  DECLARE eventDate DATE;  DECLARE storeID INT;  DECLARE productID INT;  DECLARE quantityChange INT;  -- Open cursor  OPEN cur;  -- Loop through cursor  read\_loop: LOOP  FETCH cur INTO event, eventDate, storeID, productID, quantityChange;  IF done THEN  LEAVE read\_loop;  END IF;  -- Update running inventory  UPDATE TempInventory  SET Inventory = Inventory + quantityChange  WHERE StoreID = storeID AND ProductID = productID;  -- Insert into result table  SELECT eventDate AS Date, event AS Event, storeID AS StoreID, productID AS ProductID,  quantityChange AS QuantityChange,  (SELECT Inventory FROM TempInventory WHERE StoreID = storeID AND ProductID = productID) AS CurrentInventory;  END LOOP;  -- Close cursor  CLOSE cur;  -- Drop temporary table  DROP TEMPORARY TABLE TempInventory;  END$$  DELIMITER ; |
| That was excellent. There appears to be a little problem around the "DECLARE done INT DEFAULT FALSE;" line; MySQL Workbench says an END was expected. What do you think? | I see what might be causing the issue. It seems MySQL Workbench is expecting an END statement for the BEGIN block before encountering the DECLARE statement. In MySQL, you cannot have DECLARE statements within a BEGIN...END block after executable statements.  Let's revise the stored procedure by moving the DECLARE statements to the top of the procedure body, right after the BEGIN, and ensure proper ordering. Here is the corrected version:  DELIMITER $$  CREATE PROCEDURE RunningPerStoreInventoryChanges()  BEGIN  -- Declare cursor-related variables  DECLARE done INT DEFAULT FALSE;  DECLARE event VARCHAR(10);  DECLARE eventDate DATE;  DECLARE storeID INT;  DECLARE productID INT;  DECLARE quantityChange INT;  -- Declare cursor for combined deliveries and sales  DECLARE cur CURSOR FOR  SELECT 'Delivery' AS Event, DeliveryDate AS EventDate, StoreID, ProductID, Quantity  FROM Delivery  UNION ALL  SELECT 'Sale' AS Event, SaleDate AS EventDate, StoreID, ProductID, -Quantity  FROM Sale  ORDER BY EventDate;  -- Declare continue handler  DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = TRUE;  -- Create a temporary table to store the running inventory  CREATE TEMPORARY TABLE TempInventory (  StoreID INT,  ProductID INT,  Inventory INT DEFAULT 0  );  -- Populate initial inventory values from deliveries  INSERT INTO TempInventory (StoreID, ProductID, Inventory)  SELECT StoreID, ProductID, SUM(Quantity)  FROM Delivery  GROUP BY StoreID, ProductID;  -- Open cursor  OPEN cur;  -- Loop through cursor  read\_loop: LOOP  FETCH cur INTO event, eventDate, storeID, productID, quantityChange;  IF done THEN  LEAVE read\_loop;  END IF;  -- Update running inventory  UPDATE TempInventory  SET Inventory = Inventory + quantityChange  WHERE StoreID = storeID AND ProductID = productID;  -- Insert into result table  SELECT eventDate AS Date, event AS Event, storeID AS StoreID, productID AS ProductID,  quantityChange AS QuantityChange,  (SELECT Inventory FROM TempInventory WHERE StoreID = storeID AND ProductID = productID) AS CurrentInventory;  END LOOP;  -- Close cursor  CLOSE cur;  -- Drop temporary table  DROP TEMPORARY TABLE TempInventory;  END$$  DELIMITER ; |
| Comment: Copilot created a great, complex and unoptimized procedure but it had a bug, then Copilot fixed it. | DELIMITER $$  CREATE PROCEDURE RunningPerStoreInventoryChanges()  BEGIN  -- Declare variables  DECLARE done INT DEFAULT FALSE;  DECLARE event VARCHAR(10);  DECLARE eventDate DATE;  DECLARE storeID INT;  DECLARE productID INT;  DECLARE quantityChange INT;  DECLARE startTime TIMESTAMP(3);  DECLARE endTime TIMESTAMP(3);  DECLARE elapsedTime DECIMAL(10, 3);  -- Declare cursor for combined deliveries and sales  DECLARE cur CURSOR FOR  SELECT 'Delivery' AS Event, DeliveryDate AS EventDate, StoreID, ProductID, Quantity  FROM Delivery  UNION ALL  SELECT 'Sale' AS Event, SaleDate AS EventDate, StoreID, ProductID, -Quantity  FROM Sale  ORDER BY EventDate;  -- Declare continue handler  DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = TRUE;  -- Create a temporary table to store the running inventory  CREATE TEMPORARY TABLE TempInventory (  StoreID INT,  ProductID INT,  Inventory INT DEFAULT 0  );  -- Populate initial inventory values from deliveries  INSERT INTO TempInventory (StoreID, ProductID, Inventory)  SELECT StoreID, ProductID, SUM(Quantity)  FROM Delivery  GROUP BY StoreID, ProductID;  -- Get the start time  SET startTime = CURRENT\_TIMESTAMP(3);  -- Open cursor  OPEN cur;  -- Loop through cursor  read\_loop: LOOP  FETCH cur INTO event, eventDate, storeID, productID, quantityChange;  IF done THEN  LEAVE read\_loop;  END IF;  -- Update running inventory  UPDATE TempInventory  SET Inventory = Inventory + quantityChange  WHERE StoreID = storeID AND ProductID = productID;  -- Insert into result table  SELECT eventDate AS Date, event AS Event, storeID AS StoreID, productID AS ProductID,  quantityChange AS QuantityChange,  (SELECT Inventory FROM TempInventory WHERE StoreID = storeID AND ProductID = productID) AS CurrentInventory;  END LOOP;  -- Close cursor  CLOSE cur;  -- Get the end time  SET endTime = CURRENT\_TIMESTAMP(3);  -- Calculate elapsed time in milliseconds  SET elapsedTime = TIMESTAMPDIFF(MICROSECOND, startTime, endTime) / 1000;  -- Output the elapsed time  SELECT CONCAT('Elapsed time: ', elapsedTime, ' milliseconds') AS ElapsedTime;  -- Drop temporary table  DROP TEMPORARY TABLE TempInventory;  END$$  DELIMITER ; |
| **QUERY OPTIMIZATION** |  |
| That fixed it. Thank you so much. What is the finest-grained time measurement we can perform when this stored procedure is run. Could you add an elapsed-milliseconds timer? | I deleted the old copy of RunningPerStoreInventoryChanges, and ran the code Copilot supplied:  DELIMITER $$  CREATE PROCEDURE RunningPerStoreInventoryChanges()  BEGIN  -- Declare variables  DECLARE done INT DEFAULT FALSE;  DECLARE event VARCHAR(10);  DECLARE eventDate DATE;  DECLARE storeID INT;  DECLARE productID INT;  DECLARE quantityChange INT;  DECLARE startTime TIMESTAMP(3);  DECLARE endTime TIMESTAMP(3);  DECLARE elapsedTime DECIMAL(10, 3);  -- Declare cursor for combined deliveries and sales  DECLARE cur CURSOR FOR  SELECT 'Delivery' AS Event, DeliveryDate AS EventDate, StoreID, ProductID, Quantity  FROM Delivery  UNION ALL  SELECT 'Sale' AS Event, SaleDate AS EventDate, StoreID, ProductID, -Quantity  FROM Sale  ORDER BY EventDate;  -- Declare continue handler  DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = TRUE;  -- Create a temporary table to store the running inventory  CREATE TEMPORARY TABLE TempInventory (  StoreID INT,  ProductID INT,  Inventory INT DEFAULT 0  );  -- Populate initial inventory values from deliveries  INSERT INTO TempInventory (StoreID, ProductID, Inventory)  SELECT StoreID, ProductID, SUM(Quantity)  FROM Delivery  GROUP BY StoreID, ProductID;  -- Get the start time  SET startTime = CURRENT\_TIMESTAMP(3);  -- Open cursor  OPEN cur;  -- Loop through cursor  read\_loop: LOOP  FETCH cur INTO event, eventDate, storeID, productID, quantityChange;  IF done THEN  LEAVE read\_loop;  END IF;  -- Update running inventory  UPDATE TempInventory  SET Inventory = Inventory + quantityChange  WHERE StoreID = storeID AND ProductID = productID;  -- Insert into result table  SELECT eventDate AS Date, event AS Event, storeID AS StoreID, productID AS ProductID,  quantityChange AS QuantityChange,  (SELECT Inventory FROM TempInventory WHERE StoreID = storeID AND ProductID = productID) AS CurrentInventory;  END LOOP;  -- Close cursor  CLOSE cur;  -- Get the end time  SET endTime = CURRENT\_TIMESTAMP(3);  -- Calculate elapsed time in milliseconds  SET elapsedTime = TIMESTAMPDIFF(MICROSECOND, startTime, endTime) / 1000;  -- Output the elapsed time  SELECT CONCAT('Elapsed time: ', elapsedTime, ' milliseconds') AS ElapsedTime;  -- Drop temporary table  DROP TEMPORARY TABLE TempInventory;  END$$  DELIMITER ;  Unoptimized, this procedure ran in 23 milliseconds:  +-----------------------------------+  | ElapsedTime |  +-----------------------------------+  | Elapsed time: 23.000 milliseconds |  +-----------------------------------+  1 row in set (0.0092 sec) |
| Thanks. That procedure ran in 23 milliseconds total. What kinds query optimization are possible here? If more than one is possible, what should we tr  y first? | You're welcome! There are several query optimization techniques that can help improve the performance of your stored procedure. Here are some options to consider, along with recommendations on what to try first:  **1. Indexing**  Creating indexes on frequently queried columns can significantly speed up query performance. For your stored procedure, consider adding indexes on the following columns:   * Delivery table: DeliveryDate, ProductID, StoreID * Sale table: SaleDate, ProductID, StoreID * TempInventory table: StoreID, ProductID   **2. Query Optimization**  Ensure that the queries within your stored procedure are optimized for performance. This includes:   * Using JOIN operations efficiently. * Minimizing the number of nested queries and subqueries. * Reducing the amount of data processed by filtering early in the query.   **3. Temporary Table Optimization**  Using temporary tables can be beneficial, but they should be used judiciously. Ensure that:   * Temporary tables have the appropriate indexes. * The scope of temporary tables is minimized to reduce overhead.   **4. Cursor Optimization**  Cursors can be slow and memory-intensive. Consider:   * Using set-based operations instead of cursors when possible. * Fetching smaller batches of data in each cursor iteration.   **Recommendations**   1. **Start with Indexing:** Adding indexes is often the simplest and most effective first step. Here are the SQL statements to create the recommended indexes:   -- Indexes for Delivery table  CREATE INDEX idx\_delivery\_date ON Delivery (DeliveryDate);  CREATE INDEX idx\_delivery\_product\_store ON Delivery (ProductID, StoreID);  -- Indexes for Sale table  CREATE INDEX idx\_sale\_date ON Sale (SaleDate);  CREATE INDEX idx\_sale\_product\_store ON Sale (ProductID, StoreID);  -- Indexes for TempInventory table  CREATE INDEX idx\_temp\_inventory\_store\_product ON TempInventory (StoreID, ProductID);  The effect of adding indexes (2 to tables, one in the stored procedure’s temporary table), was reduction from 23 ms to 20 ms.  +-----------------------------------+  | ElapsedTime |  +-----------------------------------+  | Elapsed time: 20.000 milliseconds |  +-----------------------------------+  1 row in set (0.0100 sec) |
| That was great. Time reduced from 23 to 20 msec. One of those indexes was created in the stored procedure; the others in the db permanent tables. How would you simplify the queries or use joins? |  |
| (Copilot disconnected.) Hello. You and I just had a long conversation about a MySQL school project. Do you have access to that discussion? | Hi there! I don't have direct access to our past conversations. However, I can certainly help you with MySQL again. Just let me know what you need assistance with, and we can pick up right where we left off. Whether it's queries, database design, or anything else related to your project, I'm here to help. 😊 |
| Well, you were very helpful. You made a database to prototype inventory changes (sales and product deliveries) to two stores. The last thing we did was create a stored procedure to compute running inventory totals for 20 products over a year. I will upload the stored procedure now. I was going to ask about query optimization, and the efficient use of join operations. There is a timer around the stored procedure, so we can measure improvements. Here it is:  CREATE DEFINER=`root`@`localhost` PROCEDURE `RunningPerStoreInventoryChanges`() BEGIN -- Declare variables DECLARE done INT DEFAULT FALSE; DECLARE event VARCHAR(10); DECLARE eventDate DATE; DECLARE storeID INT; DECLARE productID INT; DECLARE quantityChange INT; DECLARE startTime TIMESTAMP(3); DECLARE endTime TIMESTAMP(3); DECLARE elapsedTime DECIMAL(10, 3); -- Declare cursor for combined deliveries and sales DECLARE cur CURSOR FOR SELECT 'Delivery' AS Event, DeliveryDate AS EventDate, StoreID, ProductID, Quantity FROM Delivery UNION ALL SELECT 'Sale' AS Event, SaleDate AS EventDate, StoreID, ProductID, -Quantity FROM Sale ORDER BY EventDate; -- Declare continue handler DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = TRUE; -- Create a temporary table to store the running inventory CREATE TEMPORARY TABLE TempInventory ( StoreID INT, ProductID INT, Inventory INT DEFAULT 0 ); -- Add an index per Copilot recommendation CREATE INDEX idx\_temp\_inventory\_store\_product ON TempInventory (StoreID, ProductID); -- Populate initial inventory values from deliveries INSERT INTO TempInventory (StoreID, ProductID, Inventory) SELECT StoreID, ProductID, SUM(Quantity) FROM Delivery GROUP BY StoreID, ProductID; -- Get the start time SET startTime = CURRENT\_TIMESTAMP(3); -- Open cursor OPEN cur; -- Loop through cursor read\_loop: LOOP FETCH cur INTO event, eventDate, storeID, productID, quantityChange; IF done THEN LEAVE read\_loop; END IF; -- Update running inventory UPDATE TempInventory SET Inventory = Inventory + quantityChange WHERE StoreID = storeID AND ProductID = productID; -- Insert into result table SELECT eventDate AS Date, event AS Event, storeID AS StoreID, productID AS ProductID, quantityChange AS QuantityChange, (SELECT Inventory FROM TempInventory WHERE StoreID = storeID AND ProductID = productID) AS CurrentInventory; END LOOP; -- Close cursor CLOSE cur; -- Get the end time SET endTime = CURRENT\_TIMESTAMP(3); -- Calculate elapsed time in milliseconds SET elapsedTime = TIMESTAMPDIFF(MICROSECOND, startTime, endTime) / 1000; -- Output the elapsed time SELECT CONCAT('Elapsed time: ', elapsedTime, ' milliseconds') AS ElapsedTime; -- Drop temporary table DROP TEMPORARY TABLE TempInventory; END |  |
|  | I'm glad to hear that you found our previous session helpful! Let's take a look at optimizing your stored procedure for better performance.   1. **Indexes**:    * You've already added an index to the TempInventory table, which is excellent.    * Make sure the Delivery and Sale tables also have indexes on the columns StoreID, ProductID, and EventDate. 2. **Cursor Optimization**:    * Cursors can be slow, especially when processing large datasets. Instead of using a cursor, you can use a set-based approach to handle inventory changes. This can significantly improve performance. 3. **Reducing Temporary Table Usage**:    * While temporary tables can be useful, try to minimize their use if possible. In this case, we can perform the operations without relying on a temporary table.   Here's a revised version of the stored procedure that uses a set-based approach:  CREATE DEFINER=`root`@`localhost` PROCEDURE `RunningPerStoreInventoryChanges`()  BEGIN  -- Declare variables  DECLARE startTime TIMESTAMP(3);  DECLARE endTime TIMESTAMP(3);  DECLARE elapsedTime DECIMAL(10, 3);  -- Get the start time  SET startTime = CURRENT\_TIMESTAMP(3);  -- Create a result table to store running inventory changes  CREATE TEMPORARY TABLE ResultInventory (  EventDate DATE,  Event VARCHAR(10),  StoreID INT,  ProductID INT,  QuantityChange INT,  CurrentInventory INT  );  -- Calculate initial inventory based on deliveries  INSERT INTO ResultInventory (EventDate, Event, StoreID, ProductID, QuantityChange, CurrentInventory)  SELECT DeliveryDate AS EventDate, 'Delivery' AS Event, StoreID, ProductID, Quantity,  SUM(Quantity) OVER (PARTITION BY StoreID, ProductID ORDER BY DeliveryDate) AS CurrentInventory  FROM Delivery;  -- Update inventory based on sales  INSERT INTO ResultInventory (EventDate, Event, StoreID, ProductID, QuantityChange, CurrentInventory)  SELECT SaleDate AS EventDate, 'Sale' AS Event, StoreID, ProductID, -Quantity,  (SUM(Quantity) OVER (PARTITION BY StoreID, ProductID ORDER BY SaleDate) +  IFNULL((SELECT SUM(Quantity) FROM Delivery d WHERE d.StoreID = s.StoreID AND d.ProductID = s.ProductID AND d.DeliveryDate <= s.SaleDate), 0)) AS CurrentInventory  FROM Sale s;  -- Get the end time  SET endTime = CURRENT\_TIMESTAMP(3);  -- Calculate elapsed time in milliseconds  SET elapsedTime = TIMESTAMPDIFF(MICROSECOND, startTime, endTime) / 1000;  -- Output the result  SELECT EventDate, Event, StoreID, ProductID, QuantityChange, CurrentInventory FROM ResultInventory ORDER BY EventDate;  -- Output the elapsed time  SELECT CONCAT('Elapsed time: ', elapsedTime, ' milliseconds') AS ElapsedTime;  -- Drop temporary table  DROP TEMPORARY TABLE ResultInventory;  END; |
| Bingo; thank you. Elapsed time is down to 7ms, from 20 before this change. You mentioned use of joins earlier; do you see an opportunity to try that also? | Fantastic! I'm glad to hear that the optimizations reduced the elapsed time significantly.  Yes, using joins can help streamline your queries further. In this case, we can join the Delivery and Sale tables to calculate the running inventory changes. This approach maintains a set-based method and eliminates the need for a temporary table. Here’s how you can do it:  CREATE DEFINER=`root`@`localhost` PROCEDURE `RunningPerStoreInventoryChanges\_NoCursors`()  BEGIN  -- Declare variables  DECLARE startTime TIMESTAMP(3);  DECLARE endTime TIMESTAMP(3);  DECLARE elapsedTime DECIMAL(10, 3);  -- Get the start time  SET startTime = CURRENT\_TIMESTAMP(3);  -- Create a combined result set with running inventory  SELECT EventDate, Event, StoreID, ProductID, SUM(QuantityChange) AS QuantityChange,  SUM(SUM(QuantityChange)) OVER (PARTITION BY StoreID, ProductID ORDER BY EventDate) AS CurrentInventory  FROM (  SELECT DeliveryDate AS EventDate, 'Delivery' AS Event, StoreID, ProductID, Quantity AS QuantityChange  FROM Delivery  UNION ALL  SELECT SaleDate AS EventDate, 'Sale' AS Event, StoreID, ProductID, -Quantity AS QuantityChange  FROM Sale  ) AS InventoryChanges  GROUP BY EventDate, Event, StoreID, ProductID  ORDER BY EventDate;  -- Get the end time  SET endTime = CURRENT\_TIMESTAMP(3);  -- Calculate elapsed time in milliseconds  SET elapsedTime = TIMESTAMPDIFF(MICROSECOND, startTime, endTime) / 1000;  -- Output the elapsed time  SELECT CONCAT('Elapsed time: ', elapsedTime, ' milliseconds') AS ElapsedTime;  END;  +----------------------------------+  | ElapsedTime |  +----------------------------------+  | Elapsed time: 7.000 milliseconds |  +----------------------------------+ |
| Note: Here is a version that implements Joins, that had a reduced execution time. | CREATE DEFINER=`root`@`localhost` PROCEDURE `RunningPerStoreInventoryChanges\_NoCursors\_WithJoin`()  BEGIN  -- Declare variables  DECLARE startTime TIMESTAMP(3);  DECLARE endTime TIMESTAMP(3);  DECLARE elapsedTime DECIMAL(10, 3);  -- Get the start time  SET startTime = CURRENT\_TIMESTAMP(3);  -- Create a combined result set with running inventory  SELECT EventDate, Event, StoreID, ProductID, SUM(QuantityChange) AS QuantityChange,  SUM(SUM(QuantityChange)) OVER (PARTITION BY StoreID, ProductID ORDER BY EventDate) AS CurrentInventory  FROM (  SELECT DeliveryDate AS EventDate, 'Delivery' AS Event, StoreID, ProductID, Quantity AS QuantityChange  FROM Delivery  UNION ALL  SELECT SaleDate AS EventDate, 'Sale' AS Event, StoreID, ProductID, -Quantity AS QuantityChange  FROM Sale  ) AS InventoryChanges  GROUP BY EventDate, Event, StoreID, ProductID  ORDER BY EventDate;  -- Get the end time  SET endTime = CURRENT\_TIMESTAMP(3);  -- Calculate elapsed time in milliseconds  SET elapsedTime = TIMESTAMPDIFF(MICROSECOND, startTime, endTime) / 1000;  -- Output the elapsed time  SELECT CONCAT('Elapsed time: ', elapsedTime, ' milliseconds') AS ElapsedTime;  END  +----------------------------------+  | ElapsedTime |  +----------------------------------+  | Elapsed time**: 5.000 milliseconds** |  +----------------------------------+ |
|  |  |
| Thank you. I would like to change the subject to a different part of the project. There is a Delivery table and a stored procedure to populate it, GenerateSampleDeliveries. I would like a DeliveryRequestDate (date) column, and populate it with a random number 1-5 days before the actual delivery date, to simulate lead time to obtain a delivery. The SQL that created the table: CREATE TABLE Delivery ( DeliveryID INT AUTO\_INCREMENT PRIMARY KEY, SupplierID INT, ProductID INT, StoreID INT, Quantity INT, DeliveryDate DATE, FOREIGN KEY (SupplierID) REFERENCES Supplier(SupplierID), FOREIGN KEY (ProductID) REFERENCES Product(ProductID), FOREIGN KEY (StoreID) REFERENCES Store(StoreID) ); and the SQL that populated it: -- Generate 200 sample deliveries DELIMITER $$ CREATE PROCEDURE GenerateSampleDeliveries() BEGIN DECLARE i INT DEFAULT 1; DECLARE maxDeliveries INT DEFAULT 200; WHILE i <= maxDeliveries DO INSERT INTO Delivery (SupplierID, ProductID, StoreID, Quantity, DeliveryDate) VALUES ( FLOOR(1 + (RAND() \* 3)), -- Random SupplierID between 1 and 3 FLOOR(1 + (RAND() \* 20)), -- Random ProductID between 1 and 20 FLOOR(1 + (RAND() \* 2)), -- Random StoreID between 1 and 2 FLOOR(1 + (RAND() \* 20)), -- Random Quantity between 1 and 20 DATE\_ADD('2024-01-01', INTERVAL FLOOR(RAND() \* 365) DAY) -- Random date in 2024 ); SET i = i + 1; END WHILE; END$$ DELIMITER ; -- Call the procedure to generate the sample deliveries CALL GenerateSampleDeliveries(); So, could you add the new column and revise the populate-the-table procedure? | Sure, I'd be happy to help with that! First, you'll need to modify the Delivery table to add the new DeliveryRequestDate column. Then, you'll need to revise the GenerateSampleDeliveries stored procedure to populate this new column with a random date 1-5 days before the actual delivery date.  Here are the steps:   1. **Alter the Delivery Table**:    * Add the DeliveryRequestDate column to the Delivery table. 2. **Revise the Stored Procedure**:    * Update the GenerateSampleDeliveries procedure to populate the DeliveryRequestDate column.   ALTER TABLE Delivery  ADD COLUMN DeliveryRequestDate DATE; |
| **Thank you. There appears to be an issue with the DeliveryRequestDate column; the values are all null.** | I'm sorry to hear that there's an issue with the DeliveryRequestDate column. Let's take a closer look and debug it together.  It seems like the issue could be related to the way the dates are being calculated and inserted. Let's try updating the stored procedure to ensure that the DeliveryRequestDate is correctly calculated and inserted:  DELIMITER $$  CREATE PROCEDURE GenerateSampleDeliveries()  BEGIN  DECLARE i INT DEFAULT 1;  DECLARE maxDeliveries INT DEFAULT 200;  DECLARE deliveryDate DATE;  DECLARE requestDate DATE;  WHILE i <= maxDeliveries DO  -- Generate random delivery date  SET deliveryDate = DATE\_ADD('2024-01-01', INTERVAL FLOOR(RAND() \* 365) DAY);    -- Generate random request date 1-5 days before the delivery date  SET requestDate = DATE\_SUB(deliveryDate, INTERVAL FLOOR(1 + (RAND() \* 5)) DAY);  INSERT INTO Delivery (SupplierID, ProductID, StoreID, Quantity, DeliveryDate, DeliveryRequestDate)  VALUES (  FLOOR(1 + (RAND() \* 3)), -- Random SupplierID between 1 and 3  FLOOR(1 + (RAND() \* 20)), -- Random ProductID between 1 and 20  FLOOR(1 + (RAND() \* 2)), -- Random StoreID between 1 and 2  FLOOR(1 + (RAND() \* 20)), -- Random Quantity between 1 and 20  deliveryDate,  requestDate  );  SET i = i + 1;  END WHILE;  END$$  DELIMITER ; |
|  |  |
| Thank you; that worked very well. I would like to ask for a new stored procedure to exercise nested queries and aggregation, using the Sale, Delivery and Product tables as input. So, by reading the Sale and Delivery tables you can see how many of each product id is sold, and report each product's total sale at the end. Here are definitions for Sale, Product and Delivery tables: | Great! Let's create a stored procedure that calculates the total sales for each product by reading the Sale and Delivery tables. We will use nested queries and aggregation to achieve this.  Here's the stored procedure:  DELIMITER $$  CREATE PROCEDURE CalculateTotalProductSales()  BEGIN  -- Declare variable for product name  DECLARE productName VARCHAR(255);  -- Create a temporary table to store the results  CREATE TEMPORARY TABLE ProductSales (  ProductID INT,  ProductName VARCHAR(255),  TotalSales INT  );  -- Insert aggregated sales data into the temporary table  INSERT INTO ProductSales (ProductID, ProductName, TotalSales)  SELECT p.ProductID, p.ProductName, IFNULL(SUM(s.Quantity), 0) AS TotalSales  FROM Product p  LEFT JOIN Sale s ON p.ProductID = s.ProductID  GROUP BY p.ProductID, p.ProductName;  -- Select the result from the temporary table  SELECT \* FROM ProductSales;  -- Drop the temporary table  DROP TEMPORARY TABLE ProductSales;  END$$  DELIMITER ;  +-----------+-------------------+------------+  | ProductID | ProductName | TotalSales |  +-----------+-------------------+------------+  | 1 | Smartphone | 42 |  | 2 | Laptop | 55 |  | 3 | Headphones | 43 |  | 4 | T-Shirt | 84 |  | 5 | Jeans | 92 |  | 6 | Microwave | 77 |  | 7 | Refrigerator | 54 |  | 8 | Toaster | 46 |  | 9 | Toy Car | 23 |  | 10 | Doll | 64 |  | 11 | Puzzle | 97 |  | 12 | Children's Book | 52 |  | 13 | Novel | 47 |  | 14 | Cookbook | 31 |  | 15 | Smartwatch | 23 |  | 16 | Bluetooth Speaker | 91 |  | 17 | Jacket | 39 |  | 18 | Blender | 38 |  | 19 | Board Game | 33 |  | 20 | Non-Fiction Book | 15 |  +-----------+-------------------+------------+ |
| That's great. Now, could you write a similar procedure that reports the max, min and average lead time for deliveries of different products, again sorted by productID? | Of course! Let's create a stored procedure that calculates the maximum, minimum, and average lead time for deliveries of different products. We'll use the Delivery table and perform the necessary calculations using aggregation functions.  Here's the stored procedure:  DELIMITER $$  CREATE PROCEDURE CalculateProductLeadTimes()  BEGIN  -- Create a temporary table to store the results  CREATE TEMPORARY TABLE ProductLeadTimes (  ProductID INT,  MaxLeadTime INT,  MinLeadTime INT,  AvgLeadTime DECIMAL(10, 2)  );  -- Insert aggregated lead time data into the temporary table  INSERT INTO ProductLeadTimes (ProductID, MaxLeadTime, MinLeadTime, AvgLeadTime)  SELECT  ProductID,  MAX(DATEDIFF(DeliveryDate, DeliveryRequestDate)) AS MaxLeadTime,  MIN(DATEDIFF(DeliveryDate, DeliveryRequestDate)) AS MinLeadTime,  AVG(DATEDIFF(DeliveryDate, DeliveryRequestDate)) AS AvgLeadTime  FROM Delivery  GROUP BY ProductID  ORDER BY ProductID;  -- Select the result from the temporary table  SELECT \* FROM ProductLeadTimes;  -- Drop the temporary table  DROP TEMPORARY TABLE ProductLeadTimes;  END$$  DELIMITER ;  The result:  +-----------+-------------+-------------+-------------+  | ProductID | MaxLeadTime | MinLeadTime | AvgLeadTime |  +-----------+-------------+-------------+-------------+  | 1 | 5 | 1 | 3.38 |  | 2 | 5 | 1 | 4.17 |  | 3 | 5 | 1 | 3.33 |  | 4 | 5 | 1 | 2.42 |  | 5 | 5 | 1 | 3.71 |  | 6 | 5 | 1 | 3.38 |  | 7 | 5 | 1 | 3.20 |  | 8 | 5 | 1 | 3.40 |  | 9 | 5 | 1 | 2.91 |  | 10 | 5 | 1 | 2.92 |  | 11 | 5 | 1 | 2.88 |  | 12 | 5 | 1 | 3.00 |  | 13 | 5 | 1 | 2.86 |  | 14 | 5 | 1 | 2.43 |  | 15 | 5 | 2 | 3.40 |  | 16 | 4 | 2 | 3.00 |  | 17 | 5 | 1 | 3.33 |  | 18 | 5 | 1 | 3.19 |  | 19 | 5 | 1 | 2.30 |  | 20 | 5 | 1 | 3.80 |  +-----------+-------------+-------------+-------------+ |
| That's great. Now, could you write a procedure that reports the total sales for each product, by examining the Sale table, which contains one row per sale event? | Absolutely! Let's create a stored procedure that calculates the total sales for each product by examining the Sale table. This stored procedure will aggregate the sales quantities for each product and report the total sales.  Here's the stored procedure:  DELIMITER $$  CREATE PROCEDURE CalculateTotalSalesPerProduct()  BEGIN  -- Create a temporary table to store the results  CREATE TEMPORARY TABLE TotalSales (  ProductID INT,  TotalSales INT  );  -- Insert aggregated sales data into the temporary table  INSERT INTO TotalSales (ProductID, TotalSales)  SELECT  ProductID,  SUM(Quantity) AS TotalSales  FROM Sale  GROUP BY ProductID  ORDER BY ProductID;  -- Select the result from the temporary table  SELECT \* FROM TotalSales;  -- Drop the temporary table  DROP TEMPORARY TABLE TotalSales;  END$$  DELIMITER ;  +-----------+------------+  | ProductID | TotalSales |  +-----------+------------+  | 1 | 42 |  | 2 | 55 |  | 3 | 43 |  | 4 | 84 |  | 5 | 92 |  | 6 | 77 |  | 7 | 54 |  | 8 | 46 |  | 9 | 23 |  | 10 | 64 |  | 11 | 97 |  | 12 | 52 |  | 13 | 47 |  | 14 | 31 |  | 15 | 23 |  | 16 | 91 |  | 17 | 39 |  | 18 | 38 |  | 19 | 33 |  | 20 | 15 |  +-----------+------------+ |
| That's great. Now, could you write a stored procedure that would report for each product the total sales in each month of 2024? I believe ProductID could be the first column, then 12 columns for each month Jan-Dec 2024. | Absolutely! Let's create a stored procedure that calculates the total sales for each product in each month of 2024. We'll use conditional aggregation to sum the quantities for each month and present the results in a pivot table format.  Here's the stored procedure:  DELIMITER $$  CREATE PROCEDURE CalculateMonthlySales2024()  BEGIN  -- Create a temporary table to store the results  CREATE TEMPORARY TABLE MonthlySales (  ProductID INT,  Jan INT DEFAULT 0,  Feb INT DEFAULT 0,  Mar INT DEFAULT 0,  Apr INT DEFAULT 0,  May INT DEFAULT 0,  Jun INT DEFAULT 0,  Jul INT DEFAULT 0,  Aug INT DEFAULT 0,  Sep INT DEFAULT 0,  Oct INT DEFAULT 0,  Nov INT DEFAULT 0,  **Dece** INT DEFAULT 0  );  -- Insert aggregated sales data into the temporary table  INSERT INTO MonthlySales (ProductID, Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dece)  SELECT  ProductID,  SUM(CASE WHEN MONTH(SaleDate) = 1 THEN Quantity ELSE 0 END) AS Jan,  SUM(CASE WHEN MONTH(SaleDate) = 2 THEN Quantity ELSE 0 END) AS Feb,  SUM(CASE WHEN MONTH(SaleDate) = 3 THEN Quantity ELSE 0 END) AS Mar,  SUM(CASE WHEN MONTH(SaleDate) = 4 THEN Quantity ELSE 0 END) AS Apr,  SUM(CASE WHEN MONTH(SaleDate) = 5 THEN Quantity ELSE 0 END) AS May,  SUM(CASE WHEN MONTH(SaleDate) = 6 THEN Quantity ELSE 0 END) AS Jun,  SUM(CASE WHEN MONTH(SaleDate) = 7 THEN Quantity ELSE 0 END) AS Jul,  SUM(CASE WHEN MONTH(SaleDate) = 8 THEN Quantity ELSE 0 END) AS Aug,  SUM(CASE WHEN MONTH(SaleDate) = 9 THEN Quantity ELSE 0 END) AS Sep,  SUM(CASE WHEN MONTH(SaleDate) = 10 THEN Quantity ELSE 0 END) AS Oct,  SUM(CASE WHEN MONTH(SaleDate) = 11 THEN Quantity ELSE 0 END) AS Nov,  SUM(CASE WHEN MONTH(SaleDate) = 12 THEN Quantity ELSE 0 END) AS **Dece**  FROM Sale  WHERE YEAR(SaleDate) = 2024  GROUP BY ProductID  ORDER BY ProductID;  -- Select the result from the temporary table  SELECT \* FROM MonthlySales;  -- Drop the temporary table  DROP TEMPORARY TABLE MonthlySales;  END$$  DELIMITER ;  +-----------+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+------+  | ProductID | Jan | Feb | Mar | Apr | May | Jun | Jul | Aug | Sep | Oct | Nov | **Dece** |  +-----------+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+------+  | 1 | 7 | 0 | 0 | 0 | 0 | 10 | 0 | 4 | 0 | 8 | 4 | 9 |  | 2 | 0 | 0 | 10 | 0 | 5 | 2 | 5 | 0 | 4 | 0 | 22 | 7 |  | 3 | 0 | 0 | 0 | 9 | 0 | 9 | 0 | 16 | 0 | 8 | 0 | 1 |  | 4 | 11 | 1 | 18 | 11 | 0 | 25 | 0 | 8 | 0 | 7 | 0 | 3 |  | 5 | 10 | 2 | 23 | 0 | 5 | 10 | 7 | 7 | 6 | 8 | 14 | 0 |  | 6 | 0 | 9 | 0 | 14 | 0 | 7 | 13 | 0 | 7 | 1 | 16 | 10 |  | 7 | 0 | 1 | 0 | 0 | 9 | 0 | 5 | 6 | 6 | 0 | 20 | 7 |  | 8 | 2 | 0 | 4 | 2 | 7 | 10 | 2 | 0 | 4 | 3 | 0 | 12 |  | 9 | 0 | 2 | 0 | 2 | 6 | 0 | 0 | 4 | 0 | 0 | 0 | 9 |  | 10 | 18 | 0 | 7 | 0 | 1 | 4 | 0 | 0 | 3 | 29 | 2 | 0 |  | 11 | 10 | 0 | 19 | 16 | 5 | 6 | 16 | 3 | 10 | 4 | 5 | 3 |  | 12 | 0 | 2 | 0 | 0 | 8 | 0 | 9 | 9 | 4 | 1 | 17 | 2 |  | 13 | 14 | 0 | 0 | 3 | 9 | 4 | 4 | 8 | 0 | 0 | 0 | 5 |  | 14 | 0 | 0 | 1 | 7 | 0 | 3 | 9 | 0 | 2 | 0 | 0 | 9 |  | 15 | 4 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 19 | 0 |  | 16 | 0 | 12 | 11 | 7 | 27 | 11 | 8 | 0 | 2 | 8 | 5 | 0 |  | 17 | 3 | 0 | 7 | 0 | 5 | 0 | 0 | 10 | 14 | 0 | 0 | 0 |  | 18 | 10 | 9 | 0 | 0 | 10 | 2 | 0 | 6 | 0 | 0 | 0 | 1 |  | 19 | 3 | 6 | 0 | 2 | 2 | 10 | 0 | 3 | 5 | 0 | 0 | 2 |  | 20 | 1 | 0 | 0 | 2 | 7 | 0 | 0 | 0 | 0 | 0 | 0 | 5 |  +-----------+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+------+ |
|  |  |