e. Query Documentation

This section documents the essential SQL queries implemented within the C# WinForms application (ShopX). These queries demonstrate the system's capability to handle **Data Retrieval**, **Complex Transactions**, **Data Manipulation**, and **Analytical Reporting** while adhering to security best practices (prevention of SQL Injection).

i. Data Retrieval Query (Dashboard Statistics)

Function: Calculates the total revenue generated in the last 30 days to display on the Home Dashboard immediately upon login.

Code Implementation (C# Snippet):

Query Explanation:

Purpose: To provide the administrator with an immediate snapshot of financial performance without needing to generate a full report.

Syntax Analysis:

SUM(ol.Quantity * ol.UnitPrice): Aggregates the total value of sold items.

ISNULL(..., 0): Ensures the query returns 0 instead of NULL if no sales exist, preventing application runtime errors.

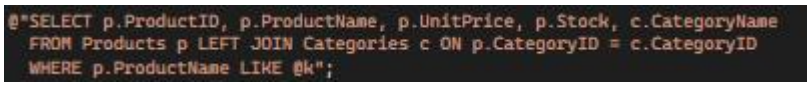
DATEADD(DAY, -30, GETDATE()): Dynamically filters records to include only transactions from the last 30 days relative to the current system time.

Efficiency: The query uses an inner JOIN to link Orders and OrderLines, processing the calculation directly on the database server to minimize data transfer.

ii. Complex Data Retrieval (Product Search with Category)

Function: Retrieves product details joined with their respective Category Names for the management grid, supporting keyword search.

Code Implementation (C# Snippet):



Query Explanation:

Purpose: Allows staff to view human-readable category names (e.g., "Beverages") instead of numeric IDs (e.g., "1") while searching for products.

Syntax Analysis:

LEFT JOIN Categories: Ensures that products are still displayed even if they are not assigned to a category (or if the category was deleted), preventing data loss in the view.

LIKE @k: Enables partial matching (e.g., searching "Pepsi" will find "Pepsi Zero").

Security: The use of SqlParameter (@k) strictly prevents **SQL Injection** attacks, ensuring that user input is treated as literal text rather than executable code.

iii. Transactional Data Manipulation (Create Order & Stock Deduct)

Function: Executes a complex sales transaction involving four atomic steps: Creating the Order, Saving Line Items, Deducting Inventory, and Logging the Movement.

Code Implementation (C# Snippet):

```
*INSERT INTO Orders (CustomerID, SalesEmpID, Status, OrderDate) VALUES (@cid, 'Paid', GETDATE()); SELECT SCOPE_IDENTITY();";
```

```
= "INSERT INTO OrderLines (OrderID, ProductID, Quantity, UnitPrice) VALUES (@oid, @pid, @qty, @price)";
```

```
"UPDATE Products SET Stock = Stock - @qty WHERE ProductID = @pid";
```

```
"INSERT INTO InventoryMovements (ProductID, QtyChange, MovementType, MovementDate) VALUES (@pid, @qty, 'OUT', GETDATE())";
```

Query Explanation:

Purpose: To guarantee **Data Integrity**. It ensures that inventory is deducted *exactly* when a sale is made.

Technical Analysis:

SCOPE_IDENTITY(): Retrieves the OrderID generated by the database in the previous statement to immediately link the items to the correct order.

SqlTransaction: Wraps all four queries. If the stock update fails (e.g., database error), the Order creation is rolled back, preventing "phantom" orders.

MovementType = 'OUT': Clearly tags this action as a sales deduction for audit trails.

iv. Inventory Management (Goods Receipt)

Function: Handles the restocking process by increasing product stock and logging the entry.

Code Implementation (C# Snippet):

```
"UPDATE Products SET Stock = Stock + @qty WHERE ProductID = @pid";
```

```
string sqlLog = @"INSERT INTO InventoryMovements
        (ProductID, QtyChange, MovementType, MovementDate, PerformedByEmpID, UnitCost,
        VALUES (@pid, @qty, 'IN', GETDATE(), @eid, @cost, @note)";
```

Query Explanation:

Purpose: To accurately record incoming stock from suppliers.

Syntax Analysis:

SET Stock = Stock + @qty: Updates the inventory incrementally (relative to current stock) rather than overwriting it, which is safer for concurrent usage.

MovementType = 'IN': Distinguishes this transaction from sales or manual adjustments in the inventory log.

v. Analytical Reporting (Top Selling Products)

Function: Aggregates sales data to identify the top 5 best-selling items within a specific date range.

Code Implementation (C# Snippet):

```
@"SELECT TOP 5 p.ProductName, SUM(ol.Quantity) AS TotalQty
  FROM OrderLines ol
  JOIN Orders o ON ol.OrderID=o.OrderID
  JOIN Products p ON ol.ProductID=p.ProductID
  WHERE o.Status='Paid' AND o.OrderDate BETWEEN @d1 AND @d2
  GROUP BY p.ProductName ORDER BY TotalQty DESC";
```

Query Explanation:

Purpose: Supports business decision-making by highlighting high-performance products.

Syntax Analysis:

TOP 5 ... ORDER BY ... DESC: Sorts the results by total quantity sold and limits the output to the top 5 records.

GROUP BY p.ProductName: Groups individual order lines by product to perform the SUM() calculation.

WHERE o.Status='Paid': Filters out cancelled or pending orders to ensure the report reflects actual finalized sales.

vi. System Maintenance (Backup Database)

Function: Performs a full database backup to a user-specified file path.

Code Implementation (C# Snippet):

```
$"BACKUP DATABASE [ShopX_BTEC] TO DISK='{txtBackupPath.Text}'";
```

Query Explanation:

Purpose: Meets the system requirement for **Resilience** and **Data Recovery**.

Mechanism: Invokes the native T-SQL BACKUP command directly from the application layer, allowing administrators to secure data without needing direct access to the SQL Server Management Studio (SSMS) interface.

vii. System Security (User Account Management)

Function: Retrieves a comprehensive list of system users by linking their login credentials (UserAccounts) with their personal details (Employees) and assigned permissions (Roles).

Code Implementation (C# Snippet):

```
@"SELECT u.EmployeeID, e.FullName, u.Username, r.RoleName, u.IsLocked
  FROM UserAccounts u
  JOIN Employees e ON u.EmployeeID = e.EmployeeID
  JOIN Roles r ON u.RoleID = r.RoleID
  WHERE u.Username LIKE @k OR e.FullName LIKE @k";
```

Query Explanation:

Purpose: Allows Administrators to manage system access efficiently by viewing "Who is Who" (Real Name vs. Username) and their security status.

Syntax Analysis:

JOIN Employees: Connects the UserAccounts table to the Employees table. This is crucial because UserAccounts only stores the EmployeeID, so the JOIN is needed to display the human-readable FullName.

JOIN Roles: Resolves the RoleID (e.g., 1, 2) into meaningful text (e.g., "Admin", "Sales") for the interface.

Security Feature: The query selects the IsLocked column, which allows the C# application to visually highlight locked accounts (e.g., coloring the row red) for immediate attention.

viii. Audit Trail (Inventory Movement Log)

Function: Retrieves the full history of stock changes (Stock In, Stock Out, Adjustments), serving as the primary audit trail for the Warehouse.

Code Implementation (C# Snippet):

```
@"SELECT im.MovementID, p.ProductName, im.QtyChange, im.MovementType, im.MovementDate, e.FullName AS [Performed By], im.Note
  FROM InventoryMovements im
  JOIN Products p ON im.ProductID = p.ProductID
  LEFT JOIN Employees e ON im.PerformedByEmpID = e.EmployeeID
  WHERE (im.MovementDate BETWEEN @d1 AND @d2)";
```

Query Explanation:

Purpose: Ensures accountability by tracking exactly *who* changed stock levels, *when*, and *why*.

Syntax Analysis:

LEFT JOIN Employees: Uses a LEFT JOIN for the PerformedByEmpID. This ensures that even if an employee record is deleted (or if a system process updated the stock without an employee ID), the movement log entry remains visible (with the name showing as NULL), preserving the audit history.

BETWEEN @d1 AND @d2: Filters records by a specific date range, allowing the warehouse manager to audit daily or monthly activities.

ix. Computed Data Retrieval (Order Details)

Function: Displays the granular details of a specific order, including a dynamic calculation of the total line price.

Code Implementation (C# Snippet):

```
@"SELECT ol.OrderID, p.ProductName, o.OrderDate, ol.Quantity, ol.UnitPrice, (ol.Quantity * ol.UnitPrice) AS [Total Price]
  FROM OrderLines ol
  JOIN Products p ON ol.ProductID = p.ProductID
  JOIN Orders o ON ol.OrderID = o.OrderID
  WHERE (o.OrderDate BETWEEN @d1 AND @d2)";
```

Query Explanation:

Purpose: Provides a detailed breakdown of sales for analysis or invoicing.

Syntax Analysis:

(ol.Quantity * ol.UnitPrice) AS [Total Price]: This is a **Computed Column**. Instead of storing the "Total Price" in the database (which creates redundancy), the system calculates it on-the-fly during retrieval. This ensures the data is always mathematically correct based on the quantity and unit price.

JOIN Products: Retrieves the product name to display to the user, as the OrderLines table only stores the ProductID.

x. Standard CRUD Operation (Employee Management)

Function: Handles the logic for both inserting a new employee or updating an existing one based on the context.

Code Implementation (C# Snippet):

```
"INSERT INTO Employees (FullName, Email, PhoneNumber, Position) VALUES (@n,@e,@p,@pos)" :
"UPDATE Employees SET FullName=@n, Email=@e, PhoneNumber=@p, Position=@pos WHERE EmployeeID=@id";
```

Query Explanation:

Purpose: Manages the lifecycle of employee records (Create and Update).

Syntax Analysis:

Conditional Logic: The application uses C# logic (id == null) to decide whether to execute an INSERT or UPDATE statement, streamlining the code.

Parameters: Uses @n, @e, etc., to map user input to database columns. This is essential for handling names with special characters (e.g., O'Connor) correctly without breaking the SQL syntax.

Based on your request to reach a total of 15 queries, here are the additional 5 items (from **xi** to **xv**) extracted from your Form2.cs code. These cover **Data Validation**, **Business Workflow**, **Manual Corrections**, **Disaster Recovery**, and **Alerts**.

Add these to your report to complete the documentation.

xi. Data Validation Query (Duplicate Prevention)

Function: Checks if a product name already exists in the database before allowing a new product to be created.

Code Implementation (C# Snippet):

```
"SELECT COUNT(*) FROM Products WHERE ProductName = @name";
```

Query Explanation:

Purpose: Enforces **Business Rules** regarding data uniqueness. It prevents duplicate entries which could confuse sales staff (e.g., having two different products named "Water").

Syntax Analysis:

COUNT(*): Returns the number of matches. If the result is greater than 0, the system blocks the INSERT action immediately.

@name: Uses a parameter to ensure the check is accurate even if the name contains special characters.

UX Benefit: Provides immediate feedback to the user ("Product already exists") instead of letting the application crash due to a database UNIQUE constraint violation.

xii. Business Workflow Management (Order Status Update)

Function: Updates the status of an existing order (e.g., cancelling an order or marking it as refunded).

Code Implementation (C# Snippet):

Query Explanation:

Purpose: Manages the **Lifecycle** of an order. It allows the administrator to handle exceptions, such as a customer returning goods.

Syntax Analysis:

UPDATE Orders SET Status=@s: Modifies only the specific status column for a specific record.

Note: In a more advanced implementation (as seen in your code comments), this might also trigger a stock reversal (adding stock back), which shows how status updates affect other tables.

xiii. Inventory Correction (Manual Adjustment)

Function: Allows the warehouse manager to manually adjust stock levels (e.g., for breakage, theft, or counting errors) and logs the reason.

Code Implementation (C# Snippet):

Query Explanation:

Purpose: Solves the problem of physical inventory mismatch. It differs from "Sales" or "Receipts" because it uses the MovementType = 'ADJUST'.

Syntax Analysis:

Stock=Stock+{nu.Value}: The value {nu.Value} can be negative (for loss/breakage) or positive (for found items). SQL handles the math correctly (adding a negative number subtracts it).

N'{tb.Text}': The N prefix denotes a Unicode string, ensuring that notes written in Vietnamese (or other languages) are stored correctly in the database.

xiv. Disaster Recovery (Database Restore)

Function: Restores the database from a backup file, forcing a disconnect of all current users to ensure the file is not locked.

Code Implementation (C# Snippet):

```
("ALTER DATABASE [ShopX_BTEC] SET SINGLE_USER WITH ROLLBACK IMMEDIATE", conn).
```

```
($"RESTORE DATABASE [ShopX_BTEC] FROM DISK='{t
```

```
("ALTER DATABASE [ShopX_BTEC] SET MULTI_USER",
```

Query Explanation:

Purpose: A critical **System Administration** function. It recovers the system to a previous safe state in case of data corruption.

Syntax Analysis:

SET SINGLE_USER WITH ROLLBACK IMMEDIATE: This is a powerful command that immediately terminates any other active connections to the database. This is necessary because SQL Server cannot restore a database while it is being used by others.

WITH REPLACE: Forces the restore operation to overwrite the existing database files.

xv. Critical Alerting (Low Stock Report)

Function: Identifies products that have fallen below the safety stock threshold (e.g., less than 10 items) to prompt reordering.

Code Implementation (C# Snippet):

```
("SELECT COUNT(*) FROM Products WHERE Stock<10", c))
```

Query Explanation:

Purpose: Provides **Proactive Business Intelligence**. Instead of waiting for a stockout to happen, it warns the manager in advance.

Syntax Analysis:

WHERE Stock < 10: Applies a specific business rule filter.

ORDER BY Stock ASC: Sorts the most critical items (lowest stock, e.g., 0 or 1) to the top of the list, ensuring they get immediate attention.