NATIONAL UNIVERSITY
COLLEGE OF COMPUTING AND
INFORMATION TECHNOLOGIES

VER 1.0

Course MANUAL

DATABASE SYSTEMS

This course provides students with the theoretical knowledge and practical skills in the utilization of databases and database management systems in the ICT applications. The logical design, physical design and implementation of relational databases are all covered in this course.

INFORMATION TECHNOLOGY

JULIE ANNE ANGELES-CRYSTAL

ADVANCE DATABASE MANAGEMENT SYSTEMS

Table of Contents

MODULE

#2

NATIONAL UNIVERSITY

MODULE GOALS

- Advance Database Features
- Stored Procedure and Function
- Triggers
- Query Optimization

FLEX Course Material

Advance Database Features

Education that works.

LESSON #

#2

SQL Language Classification.

This concept helps organize SQL commands based on their functionality, making it easier for developers and database administrators.

DDL (Data Definition Language)
Deals with the **structure** of the database — creating, altering, and deleting tables and schemas.
Characteristics:

- Affects the **schema** (layout/structure) of the database.
- Auto-commits changes — they cannot be rolled back.
- Used by **database administrators (DBAs)** to design database layouts.

Commands:
•CREATE — Define new tables, databases, indexes, etc.
•ALTER — Modify existing structures (add, drop, or modify columns).
•DROP — Permanently remove tables or databases.
•TRUNCATE — Quickly deletes **all rows** in a table without logging individual row deletion

```sql
-- Create a new table
CREATE TABLE Products (
    ProductID INT PRIMARY KEY,
    ProductName VARCHAR(100),
    Price DECIMAL(10, 2)
);

-- Add a new column
ALTER TABLE Products
ADD StockQuantity INT;

-- Remove the table
DROP TABLE Products;

-- Remove all records but keep table structure
TRUNCATE TABLE Products;
```

LESSON #

#2

SQL Language Classification.

This concept helps organize SQL commands based on their functionality, making it easier for developers and database administrators.

DML (Data Manipulation Language)

•Deals with **data within tables** — retrieving, adding, updating, and deleting records.

Characteristics:

- Does **not** auto-commit — changes can be rolled back using ROLLBACK.
- Used in daily tasks by **developers and data analysts** to manage data.

Commands:

•SELECT — Retrieve records.

•INSERT — Add new rows.

•UPDATE — Modify existing rows.

•DELETE — Remove rows

```sql
-- Add a new product
INSERT INTO Products (ProductID, ProductName, Price)
VALUES (1, 'Laptop', 999.99);

-- Update the price of a product
UPDATE Products
SET Price = 899.99
WHERE ProductID = 1;

-- Retrieve all products
SELECT * FROM Products;

-- Remove a product
DELETE FROM Products
WHERE ProductID = 1;
```

LESSON #

#2

SQL Language Classification.

This concept helps organize SQL commands based on their functionality, making it easier for developers and database administrators.

DCL (Data Control Language)

Controls **access and permissions** to the database.

Characteristics:

- Used by **DBAs** to secure databases.
- Typically works with **user roles** — granting or revoking access.

Commands:

•GRANT — Assign permissions to users.

•REVOKE — Remove permissions from users.

```
-- Allow user1 to read and add records
GRANT SELECT, INSERT ON Products TO user1;

-- Remove insert permission
REVOKE INSERT ON Products FROM user1;
```

TCL (Transaction Control Language)

Manages **transactions** — sequences of database operations that must be executed as a single unit.

Characteristics:

- Ensures **data integrity** — either **all operations succeed or none do**.
- Often paired with DML commands like INSERT, UPDATE, and DELETE.

Commands:

•COMMIT — Saves all changes made in the transaction.

•ROLLBACK — Reverts changes since the last commit.

•SAVEPOINT — Sets a point to roll back to within a transaction.

•SET TRANSACTION — Configures transaction properties like isolation level.

LESSON #

#2

SQL Language Classification.

Example:

```
-- Start a transaction
BEGIN TRANSACTION;

-- Insert a record
INSERT INTO Products (ProductID, ProductName, Price)
VALUES (2, 'Phone', 499.99);

-- Save a rollback point
SAVEPOINT BeforeUpdate;

-- Update the price
UPDATE Products
SET Price = 459.99
WHERE ProductID = 2;

-- Rollback to the savepoint
ROLLBACK TO BeforeUpdate;

-- Finalize changes
COMMIT;
```

Why is this classification important?

•**Clarity:** Helps categorize SQL commands so users know what they do.

•**Error Prevention:** Understanding TCL commands prevents accidental data loss.

•**Security:** DCL ensures sensitive data is only accessible to authorized users.

•**Optimization:** DDL and DML work together to balance database structure and data handling.

LESSON #

#2

SQL Server Command Classification: Hands-On Activity

Objective:

- Understand and classify SQL commands into DDL, DML, DCL, and TCL categories.
- Apply these commands in SQL Server.

Part 1: Categorization Task

Match each SQL command to its correct category by writing **DDL, DML, DCL, or TCL** next to each statement:

1	CREATE TABLE Students (ID INT, Name VARCHAR(50));	
2	INSERT INTO Students VALUES (1, 'John Doe');	
3	GRANT SELECT ON Students TO user1;	
4	UPDATE Students SET Name = 'Jane Doe' WHERE ID = 1;	
5	COMMIT;	
6	ALTER TABLE Students ADD Age INT;	
7	DELETE FROM Students WHERE ID = 1;	
8	ROLLBACK;	
9	DROP TABLE Students;	
10	REVOKE INSERT ON Students FROM user1;	

LESSON #

#2

SQL Server Command Classification: Hands-On Activity

Part 2: Hands-On SQL Practice Exercise:

Open SQL Server Management Studio (SSMS) and run the following exercises.

DDL Exercise:

1.Create a table named Products:

```
CREATE TABLE Products (
ProductID INT PRIMARY KEY,
ProductName VARCHAR(100),
Price DECIMAL(10, 2) )
```

2. Add a new column StockQuantity:

```
ALTER TABLE Products
ADD StockQuantity INT;
```

3. Remove the Products table:

```
DROP TABLE Products;
```

DML Exercise:

1.Insert two records into Products:

```
NSERT INTO Products (ProductID, ProductName, Price)
VALUES (1, 'Laptop', 999.99), (2, 'Phone', 499.99);
```

2. Update the price of the Phone:

```
UPDATE Products
SET Price = 450.00
WHERE ProductID = 2;
```

3.Delete the Phone record:

```
DELETE FROM Products
WHERE ProductID = 2;
```

LESSON #

#2

SQL Server Command Classification: Hands-On Activity

Part 2: Hands-On SQL Practice Exercise:

Open SQL Server Management Studio (SSMS) and run the following exercises.

DCL Exercise:

1.Grant SELECT permission to user user1:
GRANT SELECT ON Products TO user1;
2. Revoke INSERT permission from user1:
REVOKE INSERT ON Products FROM user1;

TCL Exercise:

1.Start a transaction:
BEGIN TRANSACTION;
2. Insert a record:
INSERT INTO Products (ProductID, ProductName, Price)
VALUES (3, 'Tablet', 299.99);
3. Rollback the transaction:
ROLLBACK;
4. Check if the Tablet record still exists:
SELECT * FROM Products;

Reflection Questions:

1.What happens when you use ROLLBACK versus COMMIT?
2.Why would you use GRANT and REVOKE commands in a real-world database?
3.How does ALTER TABLE differ from UPDATE?

LESSON #

#2

Stored Procedures and Functions

Stored Procedures and Functions

Definition:

•**Stored Procedures** are precompiled collections of SQL statements and control-flow logic stored in the database. They are executed by calling their name and can take input parameters, process logic, and produce output.

•Stored procedures help to:

- Automate repetitive tasks.
- Reduce client-server communication by running logic on the server.
- Enforce business rules at the database level.

•**Functions** are similar to stored procedures but differ in key ways:

- Functions must return a value (or a result set in some databases).
- They can be used within SQL expressions, like in SELECT statements.
- They are typically used for calculations and data transformation.

Key Differences:

Feature	Stored Procedure	Function
Return Value	Optional (can return multiple values via OUT parameters)	Mandatory (must return a single value or result set)
Usage in Queries	Cannot be used in SELECT statements	Can be used directly in SELECT statements
Transaction Support	Can contain transaction control statements (COMMIT, ROLLBACK)	Cannot contain transaction control statements
Invocation	Called with CALL or EXECUTE	Used like built-in SQL functions (in SELECT, WHERE, etc.)

Example:

```
-- Stored Procedure to get employee details by department
DELIMITER $$
CREATE PROCEDURE GetEmployeesByDept(IN dept_name VARCHAR(50))
BEGIN
    SELECT * FROM Employees WHERE department = dept_name;
END $$
DELIMITER ;

-- Call the procedure
CALL GetEmployeesByDept('Sales');
```

Explanation:

•IN dept_name VARCHAR(50) is an input parameter that specifies the department.

•The SELECT statement retrieves employee data filtered by the given department.

```
-- Stored Procedure to count employees in a department
DELIMITER $$
CREATE PROCEDURE CountEmployeesByDept(IN dept_name VARCHAR(50), OUT
emp_count INT)
BEGIN
    SELECT COUNT(*) INTO emp_count FROM Employees WHERE department =
dept_name;
END $$
DELIMITER ;

-- Declare a variable to store the output
CALL CountEmployeesByDept('Sales', @count);
SELECT @count;
```

Explanation:

•The OUT emp_count INT is an output parameter that returns the employee count.

•The result is stored in @count and displayed with a SELECT.

SQL SERVER:

```
CREATE PROCEDURE GetEmployeesByDept
    @DeptName VARCHAR(50)
AS
BEGIN
    SELECT * FROM Employees WHERE department = @DeptName;
END;
GO

EXEC GetEmployeesByDept 'Sales';
```

LESSON #

#2

Stored Procedure Practice Activity:

Managing Product Inventory with Stored Procedures (SQL Server)

Objective:

- Understand what a stored procedure is and how to create, execute, and modify stored procedures in SQL.
- Learn how to use input parameters and implement logic within stored procedures.

Scenario:
You are managing a small store's inventory database. The database has a table called **Products** with the following structure:

Step 1: Create the Database and Table

Create a database named StoreDB and a Products table:

```
-- Create the database
CREATE DATABASE StoreDB;
GO

-- Use the database
USE StoreDB;
GO

-- Create the Products table
CREATE TABLE Products (
    ProductID INT PRIMARY KEY IDENTITY(1,1),
    ProductName NVARCHAR(100),
    Quantity INT,
    Price DECIMAL(10, 2)
);
GO

-- Insert sample data
INSERT INTO Products (ProductName, Quantity, Price)
VALUES
('Laptop', 10, 55000.00),
('Mouse', 50, 500.00),
('Keyboard', 30, 1500.00),
('Monitor', 15, 7000.00);
GO

```

LESSON #

#2

Stored Procedure Practice Activity:

Managing Product Inventory with Stored Procedures (SQL Server)

Step 2: Create a Stored Procedure

Create a stored procedure : UpdateProductQuantity:

```
-- Create stored procedure
CREATE OR ALTER PROCEDURE UpdateProductQuantity
    @ProductID INT,
    @NewQuantity INT
AS
BEGIN
    -- Update product quantity
    UPDATE Products
    SET Quantity = @NewQuantity
    WHERE ProductID = @ProductID;

    -- Display confirmation message
    PRINT CONCAT('Product ', @ProductID, '
                 quantity updated to ', @NewQuantity);
END;
GO

```

Step 3: Execute the Stored Procedure

To run the stored procedure and update the quantity of a product:

```
-- Execute the stored procedure
EXEC UpdateProductQuantity @ProductID = 1,
                           @NewQuantity = 20;
GO
```

LESSON #

#2

Stored Procedure Practice Activity:

Managing Product Inventory with Stored Procedures (SQL Server)

Step 4: Add Validation (Advanced Task)

Enhance the procedure to validate the input ensuring the quantity is not negative:

```sql
-- Create or alter stored procedure with validation
CREATE OR ALTER PROCEDURE UpdateProductQuantityWithValidation
    @ProductID INT,
    @NewQuantity INT
AS
BEGIN
    IF @NewQuantity < 0
    BEGIN
        -- Throw an error if quantity is negative
        THROW 50000, 'Quantity cannot be negative.', 1;
    END
    ELSE
    BEGIN
        -- Update product quantity
        UPDATE Products
        SET Quantity = @NewQuantity
        WHERE ProductID = @ProductID;

        -- Print success message
        PRINT CONCAT('Product ', @ProductID, ' quantity updated to ',
                     @NewQuantity);
    END
END;
GO
```

Step 5: Test Validation

```sql
-- Try with a valid quantity
EXEC UpdateProductQuantityWithValidation
@ProductID = 2, @NewQuantity = 25;
GO

-- Try with an invalid quantity (should throw an error)
EXEC UpdateProductQuantityWithValidation
@ProductID = 2, @NewQuantity = -5;
GO
```

LESSON #

#2

Stored Procedure Practice Activity:

Managing Product Inventory with Stored Procedures (SQL Server)

Explanation:

Stored Procedure: A saved collection of SQL statements that can be executed repeatedly with different parameters.

Benefits:

•**Efficiency:** SQL Server compiles the procedure once and reuses the execution plan.

•**Security:** Users can run stored procedures without having direct access to the tables.

Key Concepts:

•**CREATE OR ALTER PROCEDURE:** Ensures you can easily update an existing stored procedure.

•**@Parameter:** Declares input parameters.

•**EXEC:** Executes a stored procedure.

•**THROW:** Raises custom error messages.

LESSON #

#2

Triggers

In **Advanced Database Systems**, **triggers** are powerful tools used to automatically execute predefined actions in response to certain events on a table or view.

Introduction to Triggers

•**Definition:** Triggers are special types of stored procedures in MS SQL Server that automatically execute in response to specific events on a table or a view.

•**Purpose:** Used for enforcing business rules, auditing, logging, and validating data.

Types of Triggers in MS SQL Server

AFTER Triggers (FOR Triggers)

- Executed after the triggering event (INSERT, UPDATE, DELETE) has completed.
- Cannot be used with views.
- Use Case: Logging product additions.

Syntax:

```
CREATE TRIGGER trg_AfterInsert
ON Products
AFTER INSERT
AS
BEGIN
    PRINT 'A new product was added!';
END;
```

INSTEAD OF Triggers

•Executed instead of the triggering event.
•Can be used on tables and views.
•Use case: Preventing deletion of important records.

Syntax:

```
CREATE TRIGGER trg_InsteadOfDelete
ON Products
INSTEAD OF DELETE
AS
BEGIN
    PRINT 'Deletion prevented!';
END;
```

LESSON #

#2

Triggers

DDL Triggers (Data Definition Language Triggers)

- Fire in response to changes to database schema (CREATE, ALTER, DROP).
- Use case: Auditing schema changes

Syntax:

```
CREATE TRIGGER trg_DDL
ON DATABASE
FOR CREATE_TABLE, ALTER_TABLE, DROP_TABLE
AS
BEGIN
    PRINT 'Schema change detected!';
END;
```

Logon Triggers

- Execute in response to a LOGON event.
- Use case: Monitoring login activity.

Syntax:

```
CREATE TRIGGER trg_Logon
ON ALL SERVER
FOR LOGON
AS
BEGIN
    PRINT 'User logged in!';
END;
```

LESSON #

#2

Triggers

Simulating "BEFORE" Triggers

•MS SQL Server does not have "BEFORE" triggers like other databases.

•You can use **INSTEAD OF triggers** to achieve similar behavior.

Example: Prevent negative prices before insertion.

```
CREATE TRIGGER trg_BeforeInsert
ON Products
INSTEAD OF INSERT
AS
BEGIN
    IF EXISTS (SELECT 1 FROM inserted WHERE Price < 0)
    BEGIN
        RAISERROR ('Price cannot be negative!', 16, 1);
    END
    ELSE
    BEGIN
        INSERT INTO Products (ProductID, ProductName, Price)
        SELECT ProductID, ProductName, Price FROM inserted;
    END
END;
```

Summary

•**AFTER triggers:** Used for logging and auditing.
•**INSTEAD OF triggers:** Override actions, useful for custom processing.
•**DDL triggers:** Monitor changes to database structure.
•**Logon triggers:** Track user sessions.

LESSON #

#2

Practice Activity

Create the necessary tables to run all the triggers:

Employees — where the main data operations happen (INSERT, UPDATE, DELETE).

AuditLog — to store records of data changes (used in AFTER triggers).

SchemaChangesLog — for logging DDL changes like creating or altering tables.

EMPLOYEE TABLE

```
CREATE TABLE Employees (
    EmployeeID INT PRIMARY KEY,
    Name NVARCHAR(100),
    Salary DECIMAL(10, 2)
);
```

•EmployeeID: Unique identifier for each employee.

•Name: Employee's name.

•Salary: Employee's salary (can't be negative for the INSTEAD OF triggers to work).

AUDITLOG TABLE

```
CREATE TABLE AuditLog (
    LogID INT PRIMARY KEY IDENTITY(1,1),
    Action NVARCHAR(50),
    EmployeeID INT,
    OldSalary DECIMAL(10, 2) NULL,
    NewSalary DECIMAL(10, 2) NULL,
    Timestamp DATETIME DEFAULT GETDATE()
);
```

•LogID: Auto-incremented primary key.

•Action: Records whether it's an INSERT, UPDATE, or DELETE action.

•EmployeeID: References the employee affected by the change.

•OldSalary: Used for updates (null for insert/delete).

•NewSalary: Used for updates (null for delete).

•Timestamp: Automatically records the time of the action.

•SCHEMACHANGESLOG TABLE

```
CREATE TABLE SchemaChangesLog (
    ChangeID INT PRIMARY KEY IDENTITY(1,1),
    Event NVARCHAR(100),
    Timestamp DATETIME DEFAULT GETDATE()
);
```

•ChangeID: Auto-incremented ID.

•Event: Logs the type of schema change (CREATE_TABLE, ALTER_TABLE, etc.).

•Timestamp: Automatically logs the time of the schema change.

LESSON #

#2

Practice Activity

AFTER INSERT Trigger

Purpose: Executes after a new record is inserted into a table. Used for logging or validating inserted data.

Example:

```
CREATE TRIGGER trg_AfterInsert
ON Employees
AFTER INSERT
AS
BEGIN
    INSERT INTO AuditLog (Action, EmployeeID, Timestamp)
    SELECT 'INSERT', EmployeeID, GETDATE()
    FROM inserted;
END;
```

Explanation:

•The inserted table is a special table that holds the newly added rows.

•This trigger logs insertions into the Audit Log table with the action, employee ID, and timestamp.

AFTER UPDATE Trigger

Purpose: Executes after an update occurs. Useful for auditing changes.

Example:

LESSON #

#2

Practice Activity

AFTER UPDATE Trigger
Purpose: Executes after an update occurs. Useful for auditing changes.
Example:

```
CREATE TRIGGER trg_AfterUpdate
ON Employees
AFTER UPDATE
AS
BEGIN
    INSERT INTO AuditLog (Action, EmployeeID, OldSalary, NewSalary, Timestamp)
    SELECT 'UPDATE', i.EmployeeID, d.Salary, i.Salary, GETDATE()
    FROM inserted i
    JOIN deleted d ON i.EmployeeID = d.EmployeeID;
END;
```

Explanation:
•The inserted table contains the new values, and the deleted table contains the old values.
•This trigger logs the old and new salary details after any update operation.

LESSON #

#2

Practice Activity

AFTER DELETE Trigger
Purpose: Executes after a record is deleted. Helps maintain a history of deleted records.
Example:

```
CREATE TRIGGER trg_AfterDelete
ON Employees
AFTER DELETE
AS
BEGIN
    INSERT INTO AuditLog (Action, EmployeeID, Timestamp)
    SELECT 'DELETE', EmployeeID, GETDATE()
    FROM deleted;
END;
```

Explanation:
•The deleted table stores rows that have been removed.
•This trigger logs which records were deleted into the Audit Log.

LESSON #

#2

Practice Activity

INSTEAD OF INSERT Trigger
Purpose: Overrides the default action for insert operations. Useful for custom validation before allowing an insert.
Example:

```
CREATE TRIGGER trg_InsteadOfInsert
ON Employees
INSTEAD OF INSERT
AS
BEGIN
    IF EXISTS (SELECT 1 FROM inserted WHERE Salary < 0)
    BEGIN
        RAISERROR('Salary cannot be negative.', 16, 1);
        ROLLBACK;
    END
    ELSE
    BEGIN
        INSERT INTO Employees (EmployeeID, Name, Salary)
        SELECT EmployeeID, Name, Salary FROM inserted;
    END
END;
```

Explanation:
•Prevents the insertion of records with negative salaries.
•If validation passes, it inserts the new records manually.

LESSON #

#2

Practice Activity

INSTEAD OF UPDATE Trigger
Purpose: Used to validate or modify updates before they happen.
Example:

```
CREATE TRIGGER trg_InsteadOfUpdate
ON Employees
INSTEAD OF UPDATE
AS
BEGIN
    IF EXISTS (SELECT 1 FROM inserted WHERE Salary < 0)
    BEGIN
        RAISERROR('Salary cannot be negative.', 16, 1);
        ROLLBACK;
    END
    ELSE
    BEGIN
        UPDATE Employees
        SET Salary = i.Salary
        FROM Employees e
        JOIN inserted i ON e.EmployeeID = i.EmployeeID;
    END
END;
```

Explanation:
•Stops updates where the new salary is negative.
•Only valid updates are processed.

LESSON #

#2

Practice Activity

DDL Triggers (FOR CREATE/DROP/ALTER)
Purpose: Used to track or prevent changes to database schema.
Example:

```
CREATE TRIGGER trg_DDL_Changes
ON DATABASE
FOR CREATE_TABLE, ALTER_TABLE, DROP_TABLE
AS
BEGIN
    PRINT 'Table schema change detected.';
    INSERT INTO SchemaChangesLog (Event, Timestamp)
    VALUES (EVENTDATA().value('(/EVENT_INSTANCE/EventType)[1]',
    'NVARCHAR(100)'), GETDATE());
END;
```

Explanation:
•Triggers when any table is created, altered, or dropped.
•Logs the type of schema change and timestamp.

LESSON #

#2

Activity

Practical Exercises

1. **Create an AFTER INSERT trigger** that logs every new entry in a "Logs" table.
2. **Develop an INSTEAD OF DELETE trigger** that prevents record deletion.
3. **Design a DDL trigger** to capture schema modifications.
4. **Implement a Logon trigger** to track user logins.

Q&A and Discussion

1. How can triggers enhance database security?
2. What are the limitations of triggers?
3. When should you use triggers versus other stored procedures?

LESSON #

#2

Query Optimization

In **MS SQL Server**, **query optimization** is the process of enhancing the efficiency of a query to ensure it runs as quickly and efficiently as possible.

How Query Optimization Works:

1.Query Parsing: SQL Server parses the query to check for syntax errors.

2.Query Compilation: It generates multiple execution plans.

3.Execution Plan Selection: The Query Optimizer chooses the "cheapest" plan in terms of resource cost.

Best Practices for Query Optimization:

1.Use Indexes Wisely:

- **Clustered Index:** Sorts rows in the table based on the key column (one per table).
- **Non-clustered Index:** Contains pointers to the data (multiple per table).
- Avoid over-indexing, as it can slow down INSERT, UPDATE, and DELETE.

2. **Avoid SELECT *:

- Specify only the columns you need:

```
SELECT name, age FROM Customers;
```

LESSON #

#2

Query Optimization

3. Use Joins Efficiently:

- **INNER JOIN** is generally faster than OUTER JOIN.
- Use indexed columns in JOIN conditions.

4. Filter Early Using WHERE Clauses:

- Reduce the number of rows processed as soon as possible:

```
SELECT * FROM Orders
WHERE OrderDate >= '2024-01-01';
```

5. Use Stored Procedures:

- Precompiled execution plans save time.

6. Avoid Functions in WHERE Clauses:

- Instead of:

```
WHERE YEAR(OrderDate) = 2024
```

- Use:

```
WHERE OrderDate >= '2024-01-01'
AND OrderDate < '2025-01-01';
```

LESSON #

#2

Query Optimization

7. Use EXISTS Instead of IN (if possible):

```
SELECT * FROM Customers
WHERE EXISTS (
  SELECT 1 FROM Orders
  WHERE Orders.CustomerID = Customers.CustomerID);
```

8. Analyze Execution Plans:

•Use **SSMS (SQL Server Management Studio)** to check **Estimated Execution Plans** (Ctrl + L).

9. Optimize Temp Tables and CTEs:

•Avoid unnecessary temp tables.

•Use **Common Table Expressions (CTEs)** for readability and performance:

```
WITH RecentOrders AS (
    SELECT * FROM Orders WHERE OrderDate >= '2024-01-01'
)
SELECT * FROM RecentOrders WHERE Amount > 100;
```

10. Update Statistics:

•Ensure statistics are current to help the optimizer make better decisions:

```
UPDATE STATISTICS Customers;
```