

Internship Project Report

Banking System

SQL Database Project

Submitted By:

Username : CT_CSI_SQ_6138

Name: Ritika Kumari

Roll Number: 2241011228

Domain: SQL

College/University: Siksha `O` Anusandhan

Course: B.Tech (Computer Science and Engineering)

Semester: 6th Semester

Passing Out Year: 2026

Batch: batch 2

Submitted To:

Organization: Celebal Technologies

Mentor Name: Anurag Yadav, Shubham Jaunjal

Designation: Senior Data Engineer

Project Duration: 02 June 2025 – 03 August 2025

Submission Date:

Date: 22 July 2025

Problem Statement:

Banks handle large volumes of sensitive data related to customers, accounts, and transactions. Without a well-structured database, issues such as data inconsistency, redundancy, and inefficiency can arise. To ensure reliable and secure banking operations, there is a need for a robust relational database system that supports key functions like managing customer information, creating accounts, recording transactions, and maintaining accurate financial records.

Goal:

The goal of this project is to design and implement a normalized, secure, and efficient banking system database using SQL Server. It will include table structures for customers, accounts, and transactions, along with stored procedures for operations such as creating customers, opening accounts, depositing and withdrawing money, transferring funds, and viewing transaction history. The solution will also ensure data integrity, performance, and be thoroughly documented for future maintenance and scalability.

1. Requirement Gathering – Understanding System Requirements

1.1 Identify Stakeholders

- **Bank Customers** – Individuals utilizing banking services.
- **Bank Staff (Tellers/Admins)** – Personnel responsible for managing accounts and processing transactions.
- **Database Administrators/Developers** – Professionals in charge of building and maintaining the database infrastructure.
-

1.2 Functional Requirements

These requirements outline the core actions that the system must support:

Feature	Description
Create Customer	Register a new customer with personal details.
Open Account	Initiate a savings or current account for an existing customer.
Deposit Money	Add funds to an existing account.
Withdraw Money	Deduct funds from an account, with validation for sufficient balance.
Transfer Money	Transfer funds from one account to another.
View Transaction History	Display the transaction history associated with a particular account.

Additional Considerations:

- Account Status (e.g., Active, Closed)
- Account Type (e.g., Savings, Current)
- Minimum Balance Requirements (optional enhancement)

1.3 Non-Functional Requirements

These are system-level attributes and constraints:

Requirement	Description
Data Integrity	Ensure valid data entry; prevent anomalies such as negative balances.
Security	Restrict access to authorized users only.
Performance	Ensure efficient data retrieval and update operations, especially for transactions.
Scalability	Maintain performance levels with an increasing number of users and accounts.
Auditability	Maintain logs of all balance changes and transaction records.

1.4 Required Tables (Initial)

Initial list of key database entities:

- **Customers** – Stores personal and contact information of account holders.
- **Accounts** – Contains account details and links to customers.
- **Transactions** – Records all financial operations, including deposits, withdrawals, and transfers.

1.5 Sample Use Case Scenarios

- **Use Case 1: Opening a New Account**
 - Register customer → Create account → Perform initial deposit
- **Use Case 2: Transferring Money**
 - Deduct amount from sender's account → Add amount to receiver's account → Log debit and credit transactions

1.6 Deliverables After Step 1

Deliverable	Description
Functional Requirements	Comprehensive list of supported database operations
Table Requirements	Defined list of entities and their attributes
Use Case Scenarios	Descriptions of key operational flows in the system
Constraints to Enforce	Rules such as no overdrafts, unique customer IDs, valid account types

2. Define Entities and Attributes – Banking System Database

This involves identifying the core entities (tables), listing relevant attributes (columns), and defining their data types, constraints, and relationships. This information is essential for designing the Entity-Relationship (ER) diagram and writing corresponding CREATE TABLE SQL scripts.

♦ Entity 1: Customers

Represents personal details of each bank customer.

Column Name	Data Type	Constraints	Description
CustomerID	INT	PRIMARY KEY, IDENTITY	Unique identifier for each customer
FullName	VARCHAR(100)	NOT NULL	Customer's full name
Email	VARCHAR(100)	UNIQUE, NOT NULL	Email address
Phone	VARCHAR(15)	UNIQUE, NOT NULL	Mobile number
DOB	DATE	NOT NULL	Date of birth

Address	VARCHAR(255)		Home address
CreatedDate	DATETIME	DEFAULT GETDATE()	Record creation date

◇ Entity 2: Accounts

Represents account details associated with customers.

Column Name	Data Type	Constraints	Description
AccountID	INT	PRIMARY KEY, IDENTITY(1001,1)	Unique identifier for each account
CustomerID	INT	FOREIGN KEY → Customers	References the owning customer
AccountType	VARCHAR(20)	CHECK (AccountType IN ('SAVINGS', 'CURRENT'))	Type of account
Balance	DECIMAL(10,2)	DEFAULT 0, CHECK (Balance >= 0)	Current balance
OpenDate	DATE	DEFAULT GETDATE()	Date when account was created
Status	VARCHAR(10)	DEFAULT 'ACTIVE'	Status of account: 'ACTIVE' or 'CLOSED'

◇ Entity 3: Transactions

Captures financial activities such as deposits, withdrawals, and transfers.

Column Name	Data Type	Constraints	Description
TransactionID	INT	PRIMARY KEY, IDENTITY	Unique transaction identifier
AccountID	INT	FOREIGN KEY → Accounts	References the related account
Type	VARCHAR(20)	NOT NULL, CHECK (Type IN ('DEPOSIT', 'WITHDRAW', 'TRANSFER'))	Nature of the transaction
Amount	DECIMAL(10,2)	CHECK (Amount > 0)	Transaction amount
TransactionDate	DATETIME	DEFAULT GETDATE()	Date and time of the transaction
Description	VARCHAR(255)		Optional details (e.g., reason for transfer)
RelatedAccountID	INT	NULLABLE, FOREIGN KEY → Accounts	Used in transfers to indicate receiving account

◇ Summary of Entity Relationships

- **One-to-Many:**
 - *Customers* → *Accounts* (A customer can hold multiple accounts)
 - *Accounts* → *Transactions* (Each account can have multiple transactions)
- **Optional Many-to-Many** (via RelatedAccountID):
 - Used to model transfers involving two accounts

3. Entities and Attributes

The Entity-Relationship (ER) diagram visually models the structure and relationships of key components in the banking system. It includes six main entities—**Bank**, **Branch**, **Customer**, **Account**, **Loan**, and their interconnections—designed to ensure a normalized and efficient relational database.

1. Bank

- a. Attributes: Name, Code, Address
- b. Relationship: A bank **has** multiple branches.

2. Branch

- a. Attributes: Branch_id, Name, Address
- b. Relationship:
 - i. Each branch **belongs to** a single bank.
 - ii. Each branch can **offer** multiple loans.
 - iii. Each branch can **maintain** multiple accounts.

3. Customer

- a. Attributes: Custid, Name, Phone, Address
- b. Relationship:
 - i. A customer can **hold** multiple accounts.
 - ii. A customer can **avail** multiple loans.

4. Account

- a. Attributes: Account_No, Acc_Type, Balance
- b. Relationship:
 - i. An account is **held by** one or more customers (many-to-many).
 - ii. An account is **maintained** by one branch.

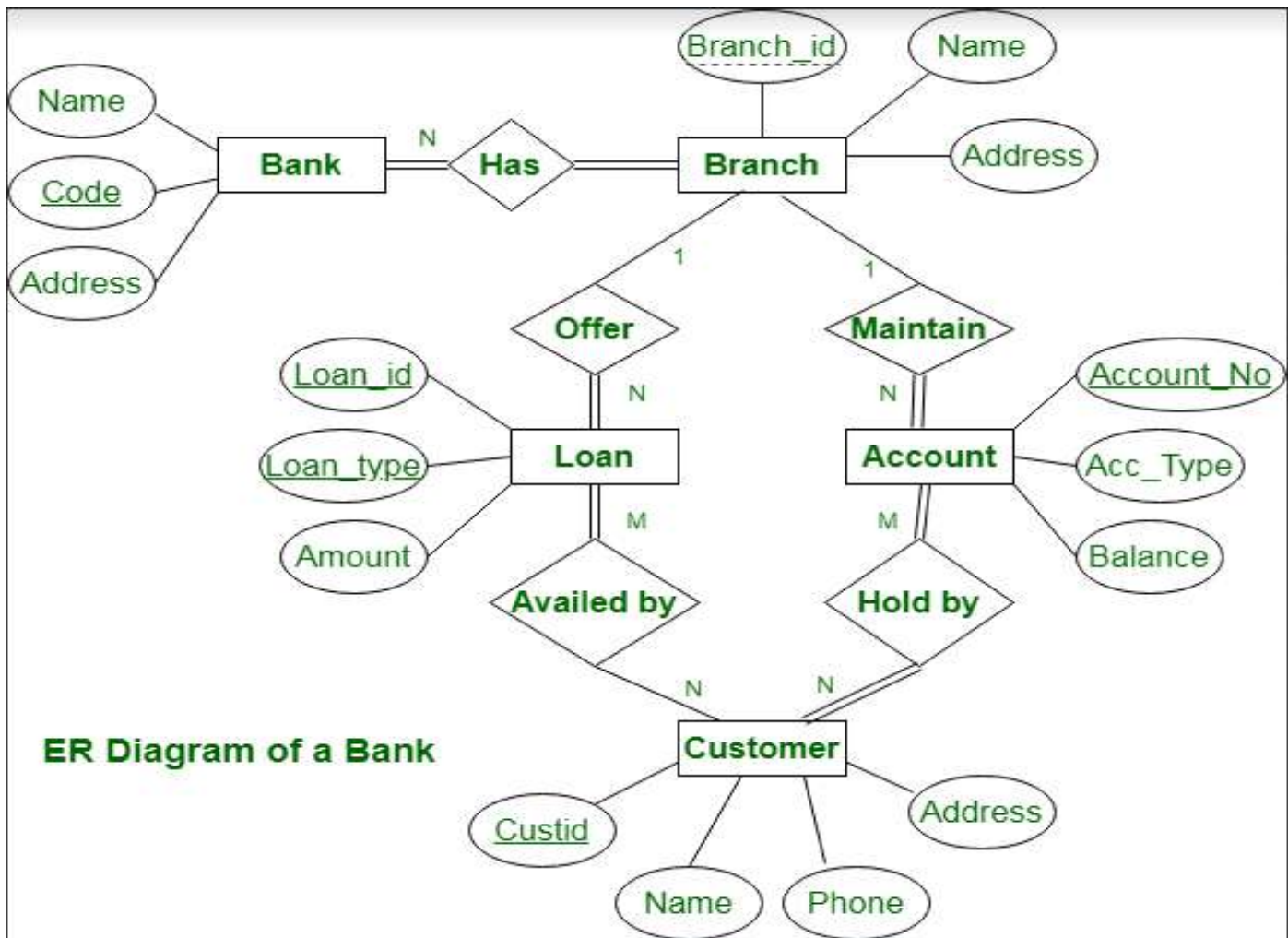
5. Loan

- a. Attributes: Loan_id, Loan_type, Amount
- b. Relationship:
 - i. A loan is **availed by** one or more customers.
 - ii. A loan is **offered** by one branch.

Relationships

Relationship	Description
Has (Bank–Branch)	One bank has many branches (1:N).
Offer (Branch–Loan)	A branch offers multiple types of loans (1:N).
Maintain (Branch–Account)	A branch maintains multiple accounts (1:N).
Hold by (Customer–Account)	A customer may hold multiple accounts, and an account may be held by multiple customers (M:N).
Availed by (Customer–Loan)	A customer may avail multiple loans, and a loan may be availed by multiple customers (M:N).

ER Diagram:

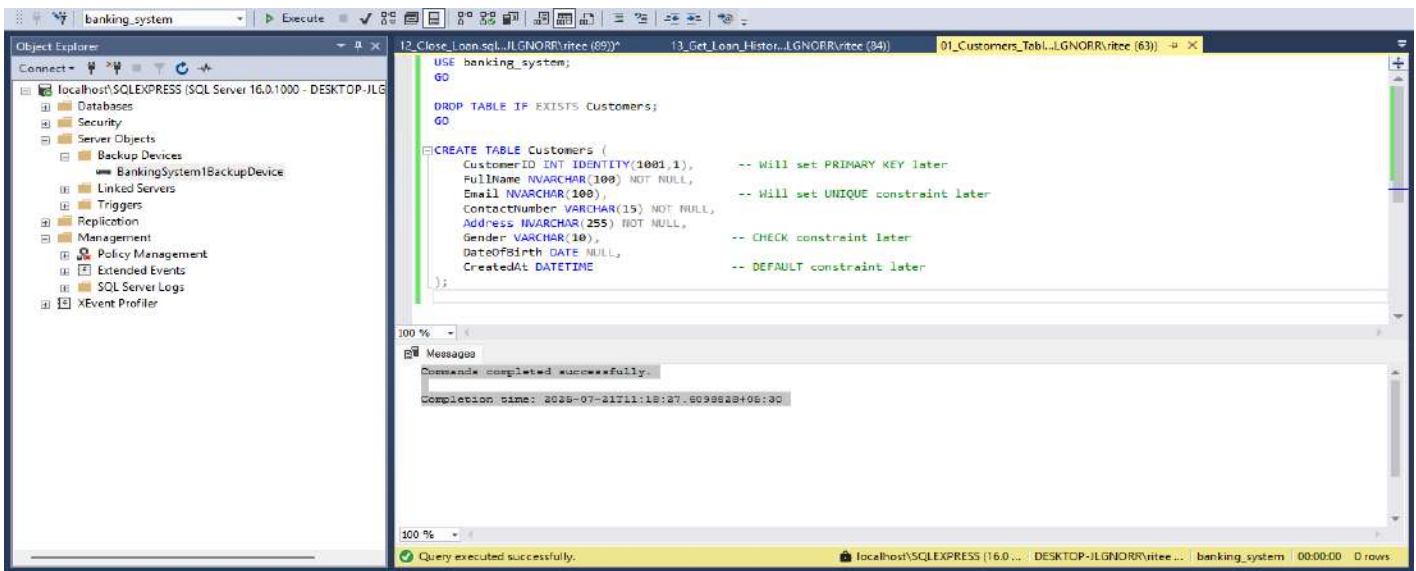


4. Database Schema

01_Customers_Table.sql

```
USE banking_system;
GO
DROP TABLE IF EXISTS Customers;
GO

CREATE TABLE Customers (
    CustomerID INT IDENTITY(1001,1),      -- Will set PRIMARY KEY later
    FullName NVARCHAR(100) NOT NULL,
    Email NVARCHAR(100),                  -- Will set UNIQUE constraint later
    ContactNumber VARCHAR(15) NOT NULL,
    Address NVARCHAR(255) NOT NULL,
    Gender VARCHAR(10),                   -- CHECK constraint later
    DateOfBirth DATE NULL,
    CreatedAt DATETIME                     -- DEFAULT constraint later
);
```



02_Accounts_Table.sql

```
USE banking_system;
```

```
GO
```

```
CREATE TABLE Accounts (
```

```
    AccountID INT IDENTITY(5001,1),    -- Will become PRIMARY KEY later
```

```
    CustomerID INT,                    -- Will be FOREIGN KEY to Customers later
```

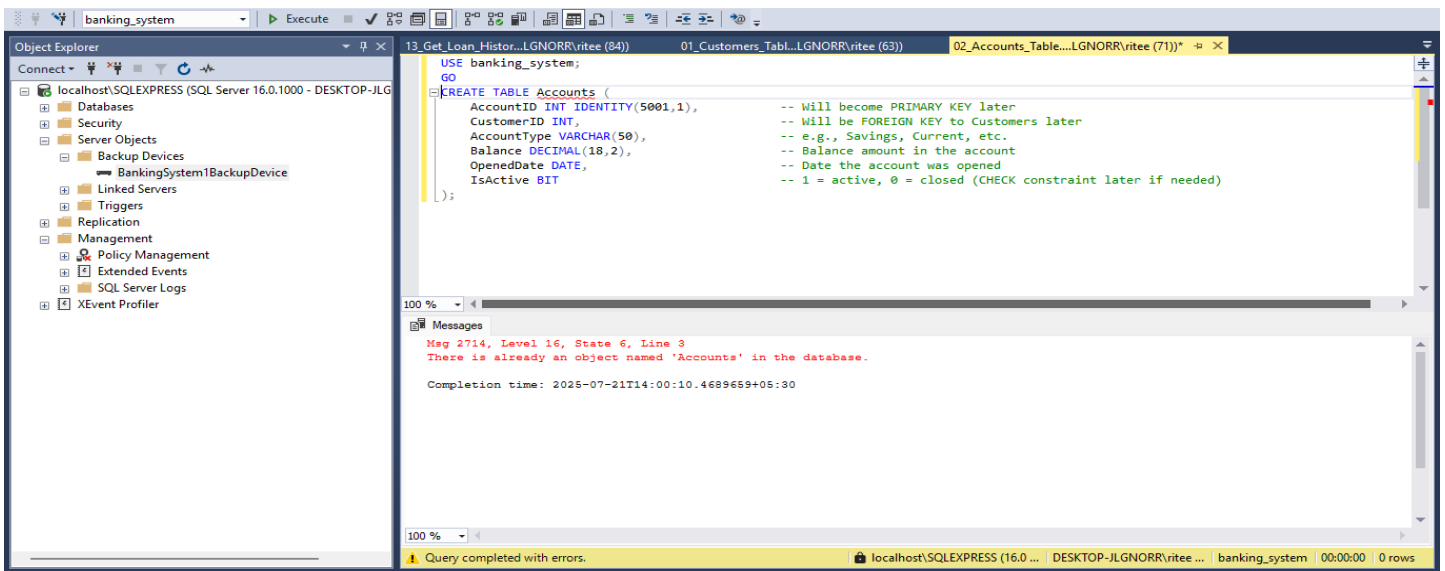
```
    AccountType VARCHAR(50),          -- e.g., Savings, Current, etc.
```

```
    Balance DECIMAL(18,2),            -- Balance amount in the account
```

```
    OpenedDate DATE,                  -- Date the account was opened
```

```
    IsActive BIT                      -- 1 = active, 0 = closed (CHECK constraint later if needed)
```

```
);
```



03_Transactions_Table.sql

USE banking_system;

GO

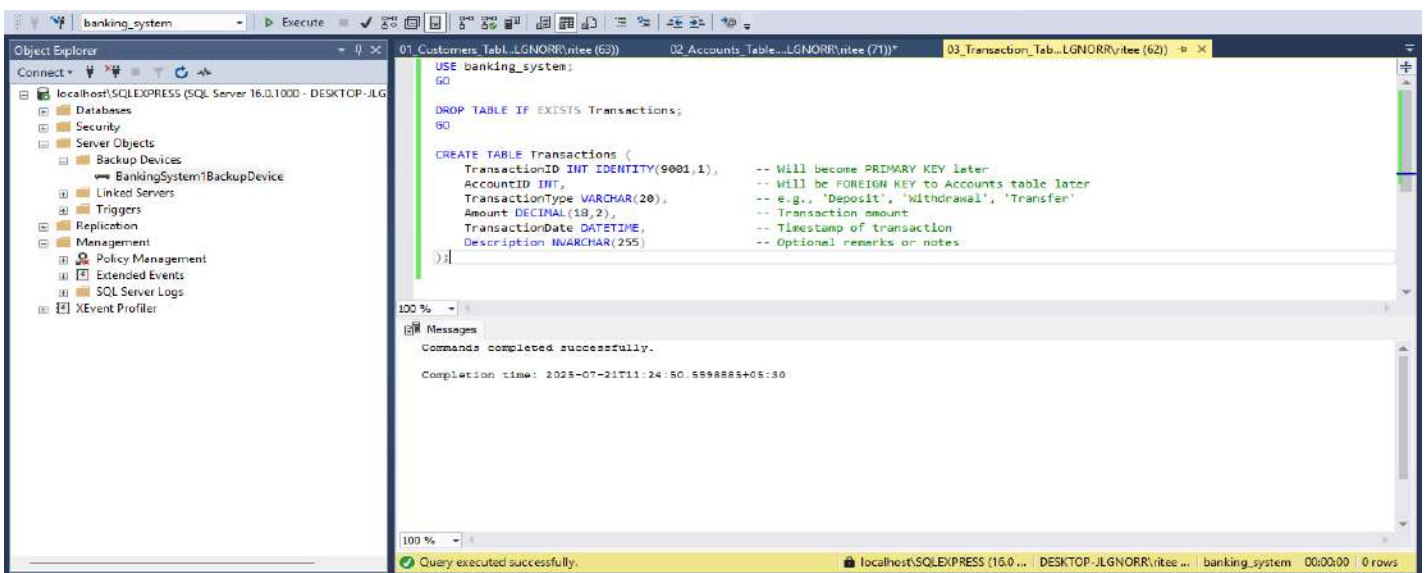
DROP TABLE IF EXISTS Transactions;

GO

```

CREATE TABLE Transactions (
    TransactionID INT IDENTITY(9001,1), -- Will become PRIMARY KEY later
    AccountID INT, -- Will be FOREIGN KEY to Accounts table later
    TransactionType VARCHAR(20), -- e.g., 'Deposit', 'Withdrawal', 'Transfer'
    Amount DECIMAL(18,2), -- Transaction amount
    TransactionDate DATETIME, -- Timestamp of transaction
    Description NVARCHAR(255) -- Optional remarks or notes
);

```



04_Branch_Table.sql

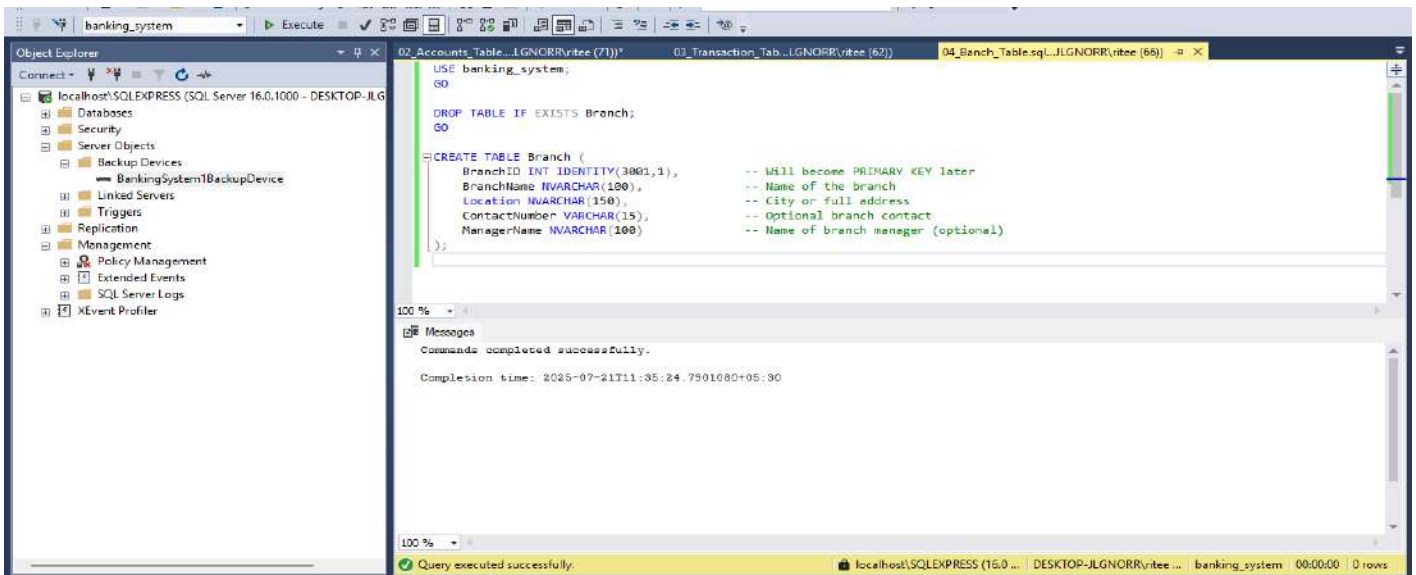
```

USE banking_system;
GO

DROP TABLE IF EXISTS Branch;
GO

CREATE TABLE Branch (
    BranchID INT IDENTITY(3001,1),    -- Will become PRIMARY KEY later
    BranchName NVARCHAR(100),        -- Name of the branch
    Location NVARCHAR(150),          -- City or full address
    ContactNumber VARCHAR(15),       -- Optional branch contact
    ManagerName NVARCHAR(100)       -- Name of branch manager (optional)
);

```



05_Loans_Table.sql

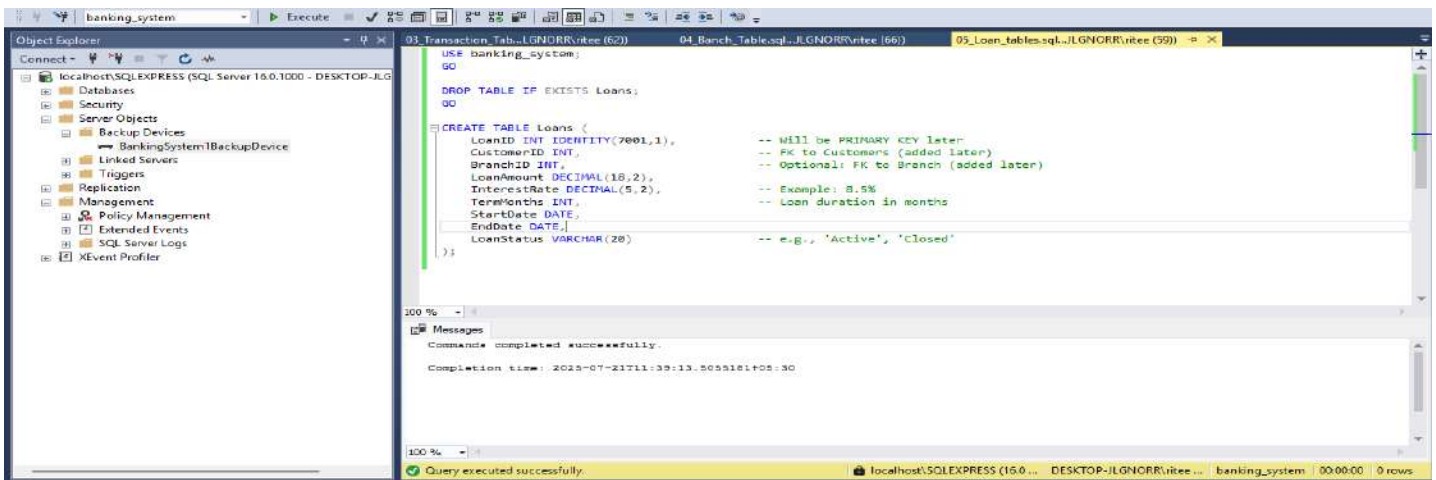
```

USE banking_system;
GO

DROP TABLE IF EXISTS Loans;
GO

CREATE TABLE Loans (
    LoanID INT IDENTITY(7001,1),    -- Will be PRIMARY KEY later
    CustomerID INT,                 -- FK to Customers (added later)
    BranchID INT,                   -- Optional: FK to Branch (added later)
    LoanAmount DECIMAL(18,2),
    InterestRate DECIMAL(5,2),      -- Example: 8.5%
    TermMonths INT,                 -- Loan duration in months
    StartDate DATE,
    EndDate DATE,
    LoanStatus VARCHAR(20)          -- e.g., 'Active', 'Closed'
);

```



06_Constraints_And_Indexes.sql

```
USE banking_system; GO
```

-- 1: DROP FOREIGN KEY CONSTRAINTS FIRST

-- Transactions → Accounts

```
IF OBJECT_ID('FK_Transactions_Accounts', 'F') IS NOT NULL ALTER TABLE Transactions DROP CONSTRAINT FK_Transactions_Accounts;
```

-- Accounts → Customers

```
IF OBJECT_ID('FK_Accounts_Customers', 'F') IS NOT NULL ALTER TABLE Accounts DROP CONSTRAINT FK_Accounts_Customers;
```

-- Loans → Customers

```
IF OBJECT_ID('FK_Loans_Customers', 'F') IS NOT NULL ALTER TABLE Loans DROP CONSTRAINT FK_Loans_Customers;
```

-- Loans → Branch

```
IF OBJECT_ID('FK_Loans_Branch', 'F') IS NOT NULL ALTER TABLE Loans DROP CONSTRAINT FK_Loans_Branch;
GO
```

-- 2: DROP EXISTING CONSTRAINTS IF ANY

-- Customers Table

```
IF OBJECT_ID('PK_Customers', 'PK') IS NOT NULL ALTER TABLE Customers DROP CONSTRAINT PK_Customers;
```

```
IF OBJECT_ID('UQ_Customers_Email', 'UQ') IS NOT NULL ALTER TABLE Customers DROP CONSTRAINT UQ_Customers_Email;
```

```
IF EXISTS (SELECT * FROM sys.check_constraints WHERE name = 'CHK_Customers_Gender') ALTER TABLE Customers DROP CONSTRAINT CHK_Customers_Gender;
```

```
IF EXISTS (SELECT * FROM sys.default_constraints WHERE name = 'DF_Customers_CreatedAt') ALTER TABLE Customers DROP CONSTRAINT DF_Customers_CreatedAt;
```

-- Accounts Table

```
IF OBJECT_ID('PK_Accounts', 'PK') IS NOT NULL ALTER TABLE Accounts DROP CONSTRAINT PK_Accounts;
```

-- Transactions Table

```
IF OBJECT_ID('PK_Transactions', 'PK') IS NOT NULL ALTER TABLE Transactions DROP CONSTRAINT PK_Transactions;
```

```
IF EXISTS (SELECT * FROM sys.check_constraints WHERE name = 'CHK_Transactions_Type') ALTER TABLE Transactions DROP CONSTRAINT CHK_Transactions_Type;
```

```
IF EXISTS (SELECT * FROM sys.default_constraints WHERE name = 'DF_Transactions_Date') ALTER TABLE Transactions DROP CONSTRAINT DF_Transactions_Date;
```

-- Branch Table

```
IF OBJECT_ID('PK_Branch', 'PK') IS NOT NULL ALTER TABLE Branch DROP CONSTRAINT PK_Branch;
```

-- Loans Table

```
IF OBJECT_ID('PK_Loans', 'PK') IS NOT NULL ALTER TABLE Loans DROP CONSTRAINT PK_Loans;
```

```
IF EXISTS (SELECT * FROM sys.check_constraints WHERE name = 'CHK_Loans_Status') ALTER TABLE Loans DROP CONSTRAINT CHK_Loans_Status; GO
```

3: ADD CONSTRAINTS BACK IN ORDER

-- Customers

```
ALTER TABLE Customers ADD CONSTRAINT PK_Customers PRIMARY KEY (CustomerID);
```

```
ALTER TABLE Customers ADD CONSTRAINT UQ_Customers_Email UNIQUE (Email);
```

```
ALTER TABLE Customers ADD CONSTRAINT CHK_Customers_Gender CHECK (Gender IN ('Male', 'Female', 'Other'));
```

```
ALTER TABLE Customers ADD CONSTRAINT DF_Customers_CreatedAt DEFAULT GETDATE() FOR CreatedAt;
```

-- Accounts

```
ALTER TABLE Accounts ADD CONSTRAINT PK_Accounts PRIMARY KEY (AccountID);
```

```
ALTER TABLE Accounts ADD CONSTRAINT FK_Accounts_Customers FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID);
```

-- Transactions

```
ALTER TABLE Transactions ADD CONSTRAINT PK_Transactions PRIMARY KEY (TransactionID);

ALTER TABLE Transactions ADD CONSTRAINT FK_Transactions_Accounts FOREIGN KEY (AccountID)
REFERENCES Accounts(AccountID);

ALTER TABLE Transactions ADD CONSTRAINT CHK_Transactions_Type CHECK (TransactionType IN ('Deposit',
'Withdrawal', 'Transfer'));

ALTER TABLE Transactions ADD CONSTRAINT DF_Transactions_Date DEFAULT GETDATE() FOR
TransactionDate;
```

-- Branch

```
ALTER TABLE Branch ADD CONSTRAINT PK_Branch PRIMARY KEY (BranchID);
```

-- Loans

```
ALTER TABLE Loans ADD CONSTRAINT PK_Loans PRIMARY KEY (LoanID);

ALTER TABLE Loans ADD CONSTRAINT FK_Loans_Customers FOREIGN KEY (CustomerID) REFERENCES
Customers(CustomerID);

ALTER TABLE Loans ADD CONSTRAINT FK_Loans_Branch FOREIGN KEY (BranchID) REFERENCES
Branch(BranchID);

ALTER TABLE Loans ADD CONSTRAINT CHK_Loans_Status CHECK (LoanStatus IN ('Active', 'Closed'));
```

Indexes

```
USE banking_system;

GO
```

-- Indexes for Customers

```
IF NOT EXISTS ( SELECT * FROM sys.indexes WHERE name = 'IX_Customers_Email' AND object_id =
OBJECT_ID('Customers'))

BEGIN CREATE NONCLUSTERED INDEX IX_Customers_Email ON Customers (Email); END;

IF NOT EXISTS ( SELECT * FROM sys.indexes WHERE name = 'IX_Customers_FullName' AND object_id =
OBJECT_ID('Customers')) BEGIN CREATE NONCLUSTERED INDEX IX_Customers_FullName ON Customers
(FullName); END;
```

-- Indexes for Accounts

```
IF NOT EXISTS ( SELECT * FROM sys.indexes WHERE name = 'IX_Accounts_CustomerID' AND object_id =  
OBJECT_ID('Accounts')) BEGIN CREATE NONCLUSTERED INDEX IX_Accounts_CustomerID ON Accounts  
(CustomerID); END;
```

```
IF NOT EXISTS ( SELECT * FROM sys.indexes WHERE name = 'IX_Accounts_AccountType' AND object_id =  
OBJECT_ID('Accounts')) BEGIN CREATE NONCLUSTERED INDEX IX_Accounts_AccountType ON Accounts  
(AccountType); END;
```

-- Indexes for Transactions

```
IF NOT EXISTS ( SELECT * FROM sys.indexes WHERE name = 'IX_Transactions_AccountID' AND object_id =  
OBJECT_ID('Transactions')) BEGIN CREATE NONCLUSTERED INDEX IX_Transactions_AccountID ON  
Transactions (AccountID); END;
```

```
IF NOT EXISTS ( SELECT * FROM sys.indexes WHERE name = 'IX_Transactions_Date' AND object_id =  
OBJECT_ID('Transactions')) BEGIN CREATE NONCLUSTERED INDEX IX_Transactions_Date ON Transactions  
(TransactionDate); END;
```

-- Indexes for Loans

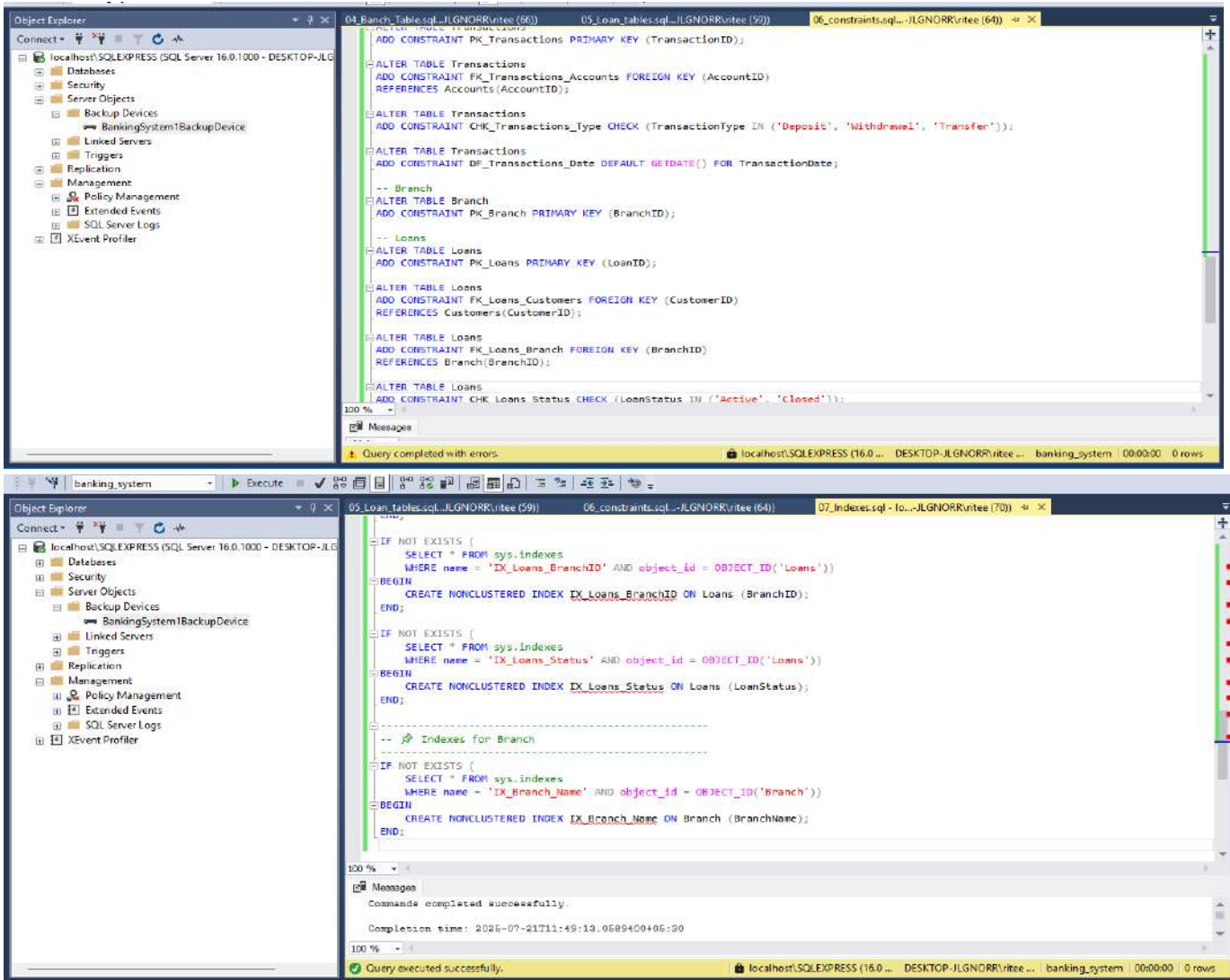
```
IF NOT EXISTS ( SELECT * FROM sys.indexes WHERE name = 'IX_Loans_CustomerID' AND object_id =  
OBJECT_ID('Loans')) BEGIN CREATE NONCLUSTERED INDEX IX_Loans_CustomerID ON Loans (CustomerID);  
END;
```

```
IF NOT EXISTS ( SELECT * FROM sys.indexes WHERE name = 'IX_Loans_BranchID' AND object_id =  
OBJECT_ID('Loans')) BEGIN CREATE NONCLUSTERED INDEX IX_Loans_BranchID ON Loans (BranchID); END;
```

```
IF NOT EXISTS ( SELECT * FROM sys.indexes WHERE name = 'IX_Loans_Status' AND object_id =  
OBJECT_ID('Loans')) BEGIN CREATE NONCLUSTERED INDEX IX_Loans_Status ON Loans (LoanStatus); END;
```

-- Indexes for Branch

```
IF NOT EXISTS ( SELECT * FROM sys.indexes WHERE name = 'IX_Branch_Name' AND object_id =  
OBJECT_ID('Branch')) BEGIN CREATE NONCLUSTERED INDEX IX_Branch_Name ON Branch (BranchName);  
END;
```



5. Stored Procedures

To streamline and secure database operations, this step implements **Stored Procedures (SPs)** for commonly performed tasks in the banking system. Each procedure encapsulates business logic, enforces validation rules, and improves reusability and maintainability.

01_Add_New_Customer.sql

```
USE banking_system;
GO
```

```
IF EXISTS ( SELECT 1 FROM sys.foreign_keys
WHERE name = 'FK_Accounts_Customers' )
BEGIN ALTER TABLE Accounts DROP CONSTRAINT FK_Accounts_Customers;
END
GO
```

```
IF EXISTS ( SELECT 1 FROM sys.foreign_keys
WHERE name = 'FK_Loans_Customers' )
BEGIN ALTER TABLE Loans DROP CONSTRAINT FK_Loans_Customers;
END
GO
```

```
IF EXISTS ( SELECT 1 FROM sys.key_constraints
WHERE name = 'PK_Customers' ) BEGIN ALTER TABLE Customers
DROP CONSTRAINT PK_Customers;
END GO
```

```
ALTER TABLE Customers ADD CONSTRAINT PK_Customers
PRIMARY KEY (CustomerID);
GO
```

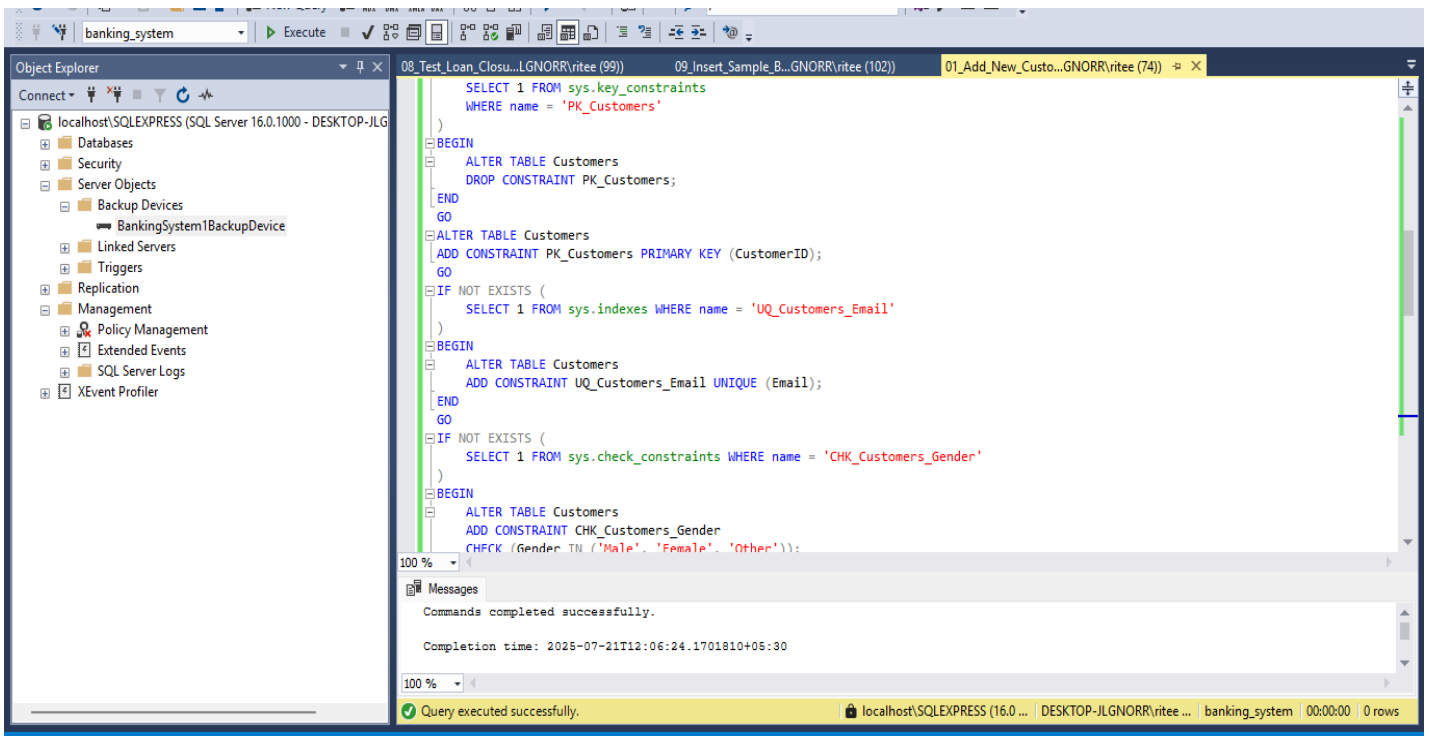
```
IF NOT EXISTS ( SELECT 1 FROM sys.indexes
WHERE name = 'UQ_Customers_Email' )
BEGIN ALTER TABLE Customers
ADD CONSTRAINT UQ_Customers_Email UNIQUE (Email);
END GO
```

```
IF NOT EXISTS ( SELECT 1 FROM sys.check_constraints
WHERE name = 'CHK_Customers_Gender' )
BEGIN ALTER TABLE Customers ADD CONSTRAINT
CHK_Customers_Gender CHECK (Gender IN ('Male', 'Female', 'Other'));
END
GO
```

```
IF NOT EXISTS ( SELECT 1 FROM sys.default_constraints
WHERE parent_object_id = OBJECT_ID('Customers')
AND COL_NAME(parent_object_id, parent_column_id) = 'CreatedAt' )
BEGIN ALTER TABLE Customers
ADD CONSTRAINT DF_Customers_CreatedAt DEFAULT GETDATE() FOR CreatedAt;
END
GO
```

```
IF NOT EXISTS ( SELECT 1 FROM sys.foreign_keys
WHERE name = 'FK_Accounts_Customers' )
BEGIN ALTER TABLE Accounts
ADD CONSTRAINT FK_Accounts_Customers
FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID);
END
GO
```

```
IF NOT EXISTS ( SELECT 1 FROM sys.foreign_keys
WHERE name = 'FK_Loans_Customers' )
BEGIN ALTER TABLE Loans
ADD CONSTRAINT FK_Loans_Customers
FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID);
END
GO
```



02_Open_New_Account.sql

```

USE banking_system;
GO

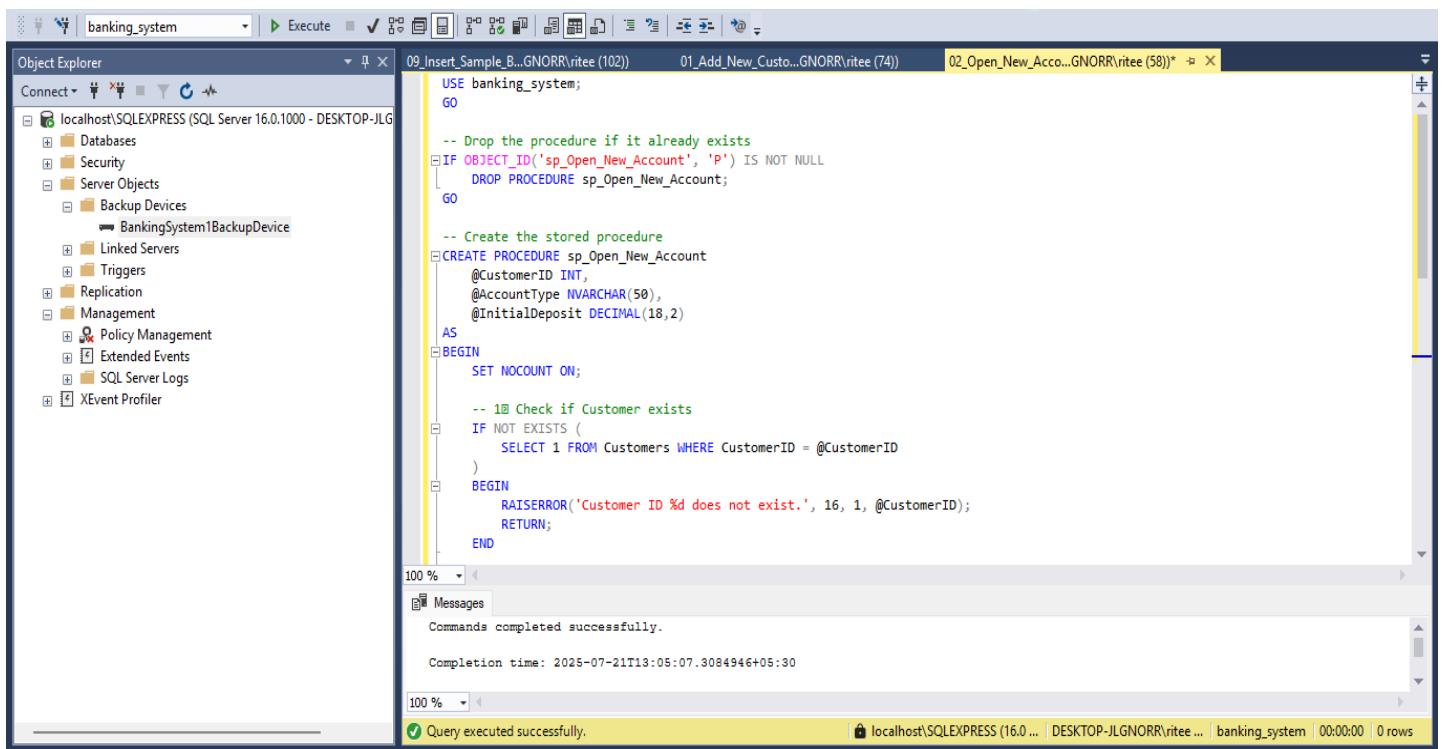
IF OBJECT_ID('sp_Open_New_Account', 'P') IS NOT NULL
DROP PROCEDURE sp_Open_New_Account;
GO

CREATE PROCEDURE sp_Open_New_Account
@CustomerID INT,
@AccountType NVARCHAR(50),
@InitialDeposit DECIMAL(18,2) AS BEGIN SET NOCOUNT ON;
IF NOT EXISTS (
    SELECT 1 FROM Customers WHERE CustomerID = @CustomerID
)
BEGIN
    RAISERROR('Customer ID %d does not exist.', 16, 1, @CustomerID);
    RETURN;
END

IF @InitialDeposit < 1000
BEGIN
    RAISERROR('Minimum initial deposit is ₹1000.', 16, 1);
    RETURN;
END

INSERT INTO Accounts (CustomerID, AccountType, Balance, OpenedDate, IsActive)
VALUES (@CustomerID, @AccountType, @InitialDeposit, GETDATE(), 1);
  
```

```
PRINT ' Account opened successfully.';
END; GO
```



03_Deposit_Amount.sql

```
USE banking_system;
GO

IF OBJECT_ID('sp_Deposit_Amount', 'P')
IS NOT NULL DROP PROCEDURE sp_Deposit_Amount; GO

CREATE PROCEDURE sp_Deposit_Amount @AccountID INT,
@Amount DECIMAL(18,2),
@Description NVARCHAR(255) = NULL AS BEGIN SET NOCOUNT ON;

IF NOT EXISTS (
    SELECT 1 FROM Accounts
    WHERE AccountID = @AccountID AND IsActive = 1
)
BEGIN
    RAISERROR('Account ID %d does not exist or is not active.', 16, 1, @AccountID);
    RETURN;
END
```

```

IF @Amount <= 0
BEGIN
    RAISERROR('Deposit amount must be greater than 0.', 16, 1);
    RETURN;
END

UPDATE Accounts
SET Balance = Balance + @Amount
WHERE AccountID = @AccountID;

INSERT INTO Transactions (AccountID, TransactionType, Amount, Description)
VALUES (@AccountID, 'Deposit', @Amount, @Description);

PRINT 'Deposit successful.';

END; GO

```

The screenshot displays the SQL Server Enterprise Manager interface. The left pane shows the 'Object Explorer' with the 'BankingSystem1BackupDevice' folder expanded. The right pane shows the 'Messages' window with the following text:

```

Commands completed successfully.

Completion time: 2025-07-21T12:27:44.7396768+05:30

```

The status bar at the bottom indicates 'Query executed successfully.' and '0 rows'.

04_Withdraw_Amount.sql

```

USE banking_system;
GO

-- Drop the procedure if it exists
IF OBJECT_ID('sp_Withdraw_Amount', 'P') IS NOT NULL
DROP PROCEDURE sp_Withdraw_Amount;
GO

CREATE PROCEDURE sp_Withdraw_Amount
@AccountID INT,
@Amount DECIMAL(18, 2),
@Description NVARCHAR(255) = NULL AS BEGIN SET NOCOUNT ON;

```

```

BEGIN TRY

    IF NOT EXISTS (
        SELECT 1 FROM Accounts
        WHERE AccountID = @AccountID AND IsActive = 1
    )
    BEGIN
        RAISERROR('Account not found or is not active.', 16, 1);
        RETURN;
    END

    DECLARE @CurrentBalance DECIMAL(18,2);
    SELECT @CurrentBalance = Balance
    FROM Accounts
    WHERE AccountID = @AccountID;

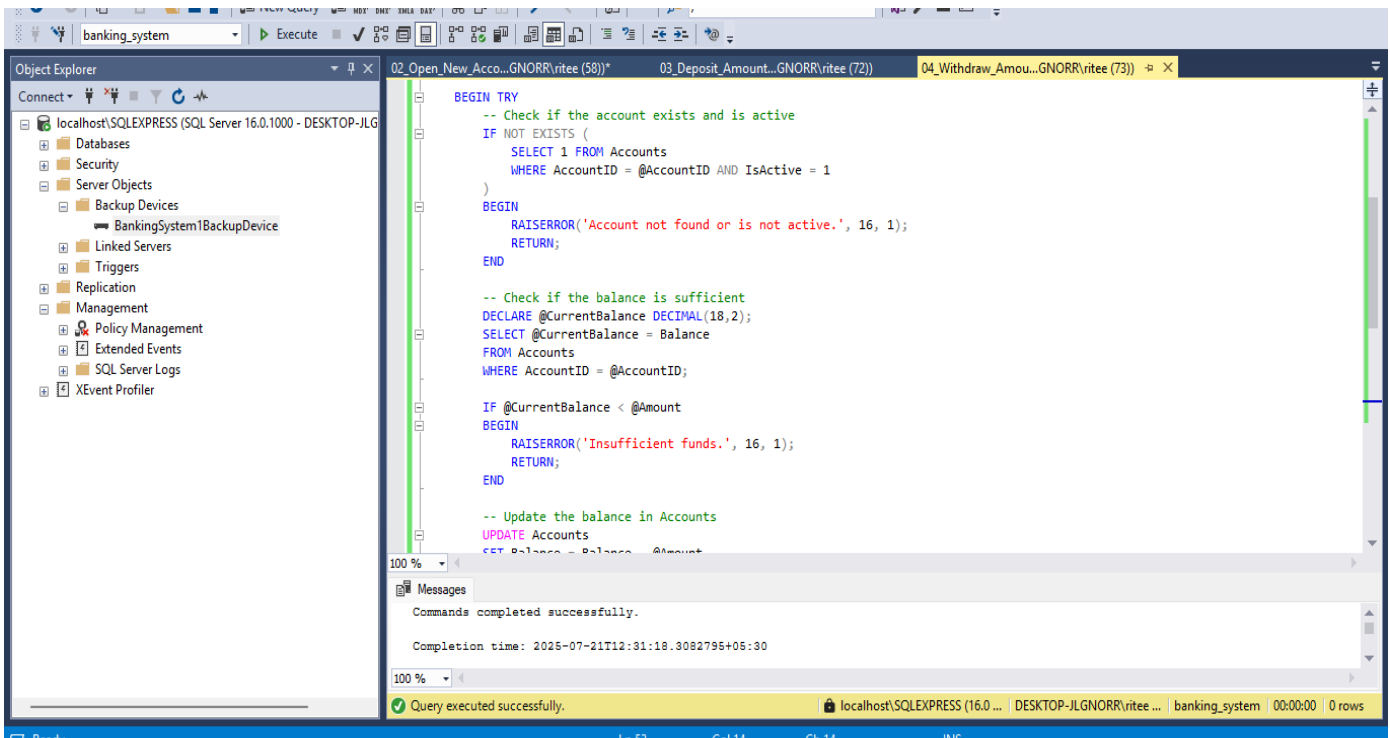
    IF @CurrentBalance < @Amount
    BEGIN
        RAISERROR('Insufficient funds.', 16, 1);
        RETURN;
    END

    UPDATE Accounts
    SET Balance = Balance - @Amount
    WHERE AccountID = @AccountID;

    INSERT INTO Transactions (AccountID, TransactionType, Amount, Description)
    VALUES (@AccountID, 'Withdrawal', @Amount, @Description);
END
TRY
BEGIN CATCH
    DECLARE @ErrMsg NVARCHAR(4000), @ErrSeverity INT;
    SELECT @ErrMsg = ERROR_MESSAGE(), @ErrSeverity = ERROR_SEVERITY();
    RAISERROR(@ErrMsg, @ErrSeverity, 1);
END
CATCH

END;
GO

```



05_Transfer_Amount.sql

USE banking_system; GO

IF OBJECT_ID('sp_Transfer_Amount', 'P') IS NOT NULL DROP PROCEDURE sp_Transfer_Amount; GO

CREATE PROCEDURE sp_Transfer_Amount @FromAccountID INT, @ToAccountID INT, @Amount DECIMAL(18, 2), @Description NVARCHAR(255) = NULL AS BEGIN SET NOCOUNT ON;

BEGIN TRY

-- Validate both accounts are active

IF NOT EXISTS (

SELECT 1 FROM Accounts WHERE AccountID = @FromAccountID AND IsActive = 1

)

BEGIN

RAISERROR('Sender account not found or inactive.', 16, 1);

RETURN;

END

IF NOT EXISTS (

SELECT 1 FROM Accounts WHERE AccountID = @ToAccountID AND IsActive = 1

)

BEGIN

RAISERROR('Receiver account not found or inactive.', 16, 1);

RETURN;

END

-- Check if sender has sufficient balance

DECLARE @FromBalance DECIMAL(18, 2);

SELECT @FromBalance = Balance FROM Accounts WHERE AccountID = @FromAccountID;

```

IF @FromBalance < @Amount
BEGIN
    RAISERROR('Insufficient funds in sender account.', 16, 1);
    RETURN;
END

-- Start transaction block
BEGIN TRANSACTION;

-- Deduct from sender
UPDATE Accounts
SET Balance = Balance - @Amount
WHERE AccountID = @FromAccountID;

-- Add to receiver
UPDATE Accounts
SET Balance = Balance + @Amount
WHERE AccountID = @ToAccountID;

-- Log sender transaction
INSERT INTO Transactions (AccountID, TransactionType, Amount, TransactionDate,
Description)
VALUES (
    @FromAccountID, 'Transfer', @Amount, GETDATE(),
    ISNULL(@Description, '') + ' (Transferred to Account ' + CAST(@ToAccountID AS
NVARCHAR) + ')')
);

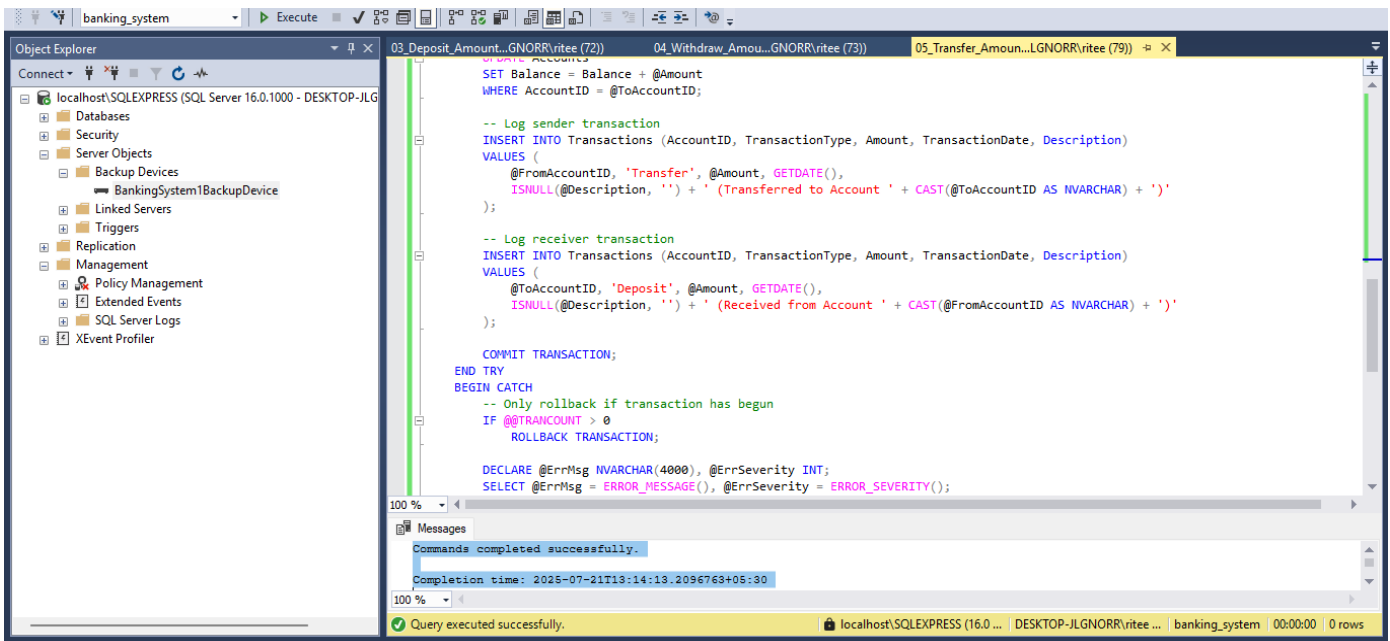
-- Log receiver transaction
INSERT INTO Transactions (AccountID, TransactionType, Amount, TransactionDate,
Description)
VALUES (
    @ToAccountID, 'Deposit', @Amount, GETDATE(),
    ISNULL(@Description, '') + ' (Received from Account ' + CAST(@FromAccountID AS
NVARCHAR) + ')')
);

COMMIT TRANSACTION;
END TRY
BEGIN CATCH
    -- Only rollback if transaction has begun
    IF @@TRANCOUNT > 0
        ROLLBACK TRANSACTION;

    DECLARE @ErrMsg NVARCHAR(4000), @ErrSeverity INT;
    SELECT @ErrMsg = ERROR_MESSAGE(), @ErrSeverity = ERROR_SEVERITY();
    RAISERROR(@ErrMsg, @ErrSeverity, 1);
END CATCH

END; GO

```



06_Close_Account.sql

```
USE banking_system; GO
```

```
-- Drop the procedure if it already exists IF OBJECT_ID('sp_Close_Account', 'P') IS NOT NULL DROP PROCEDURE
sp_Close_Account; GO
```

```
-- Create the procedure CREATE PROCEDURE sp_Close_Account @AccountID INT AS BEGIN SET NOCOUNT
ON;
```

```
-- Check if the account exists and is active
```

```
IF NOT EXISTS (
    SELECT 1 FROM Accounts WHERE AccountID = @AccountID AND IsActive = 1
)
```

```
BEGIN
```

```
    RAISERROR('Account either does not exist or is already closed.', 16, 1);
```

```
    RETURN;
```

```
END
```

```
-- Check if balance is zero
```

```
DECLARE @Balance DECIMAL(18, 2);
```

```
SELECT @Balance = Balance FROM Accounts WHERE AccountID = @AccountID;
```

```
IF @Balance <> 0
```

```
BEGIN
```

```
    RAISERROR('Account cannot be closed. Balance must be zero.', 16, 1);
```

```
    RETURN;
```

```
END
```

```
-- Close the account
```

```
UPDATE Accounts
```

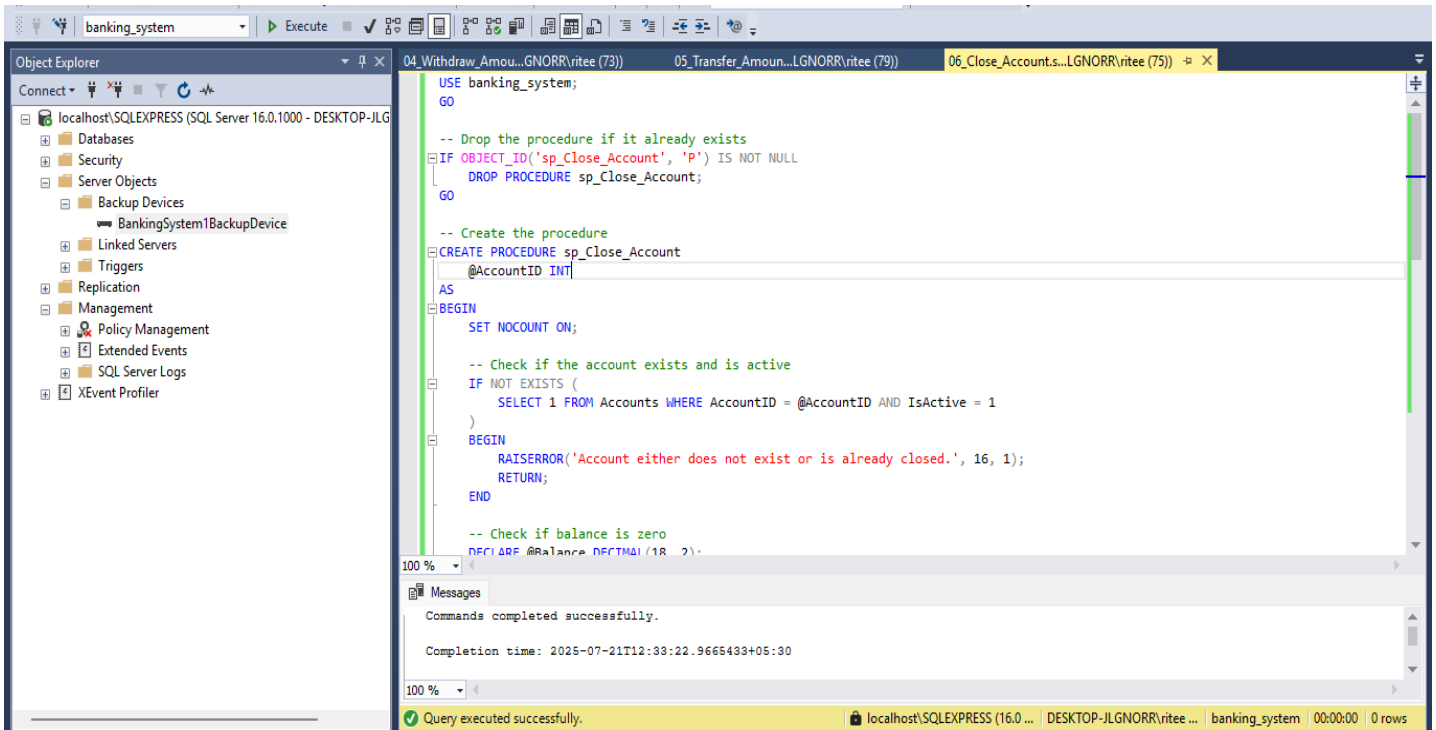
```
SET IsActive = 0
```

```
WHERE AccountID = @AccountID;
```

```
-- Optional: Log closure in Transactions table
INSERT INTO Transactions (AccountID, TransactionType, Amount, Description)
VALUES (@AccountID, 'Closure', 0.00, 'Account closed');

PRINT 'Account closed successfully.';

END; GO
```



07_Get_Transaction_History.sql

```
USE banking_system; GO

-- Drop the procedure if it already exists IF OBJECT_ID('sp_Get_Transaction_History', 'P') IS NOT NULL DROP
PROCEDURE sp_Get_Transaction_History; GO

-- Create the procedure CREATE PROCEDURE sp_Get_Transaction_History @AccountID INT, @FromDate
DATETIME = NULL, @ToDate DATETIME = NULL AS BEGIN SET NOCOUNT ON;

-- Check if the account exists
IF NOT EXISTS (
    SELECT 1 FROM Accounts WHERE AccountID = @AccountID
)
BEGIN
    RAISERROR('Account does not exist.', 16, 1);
    RETURN;
END

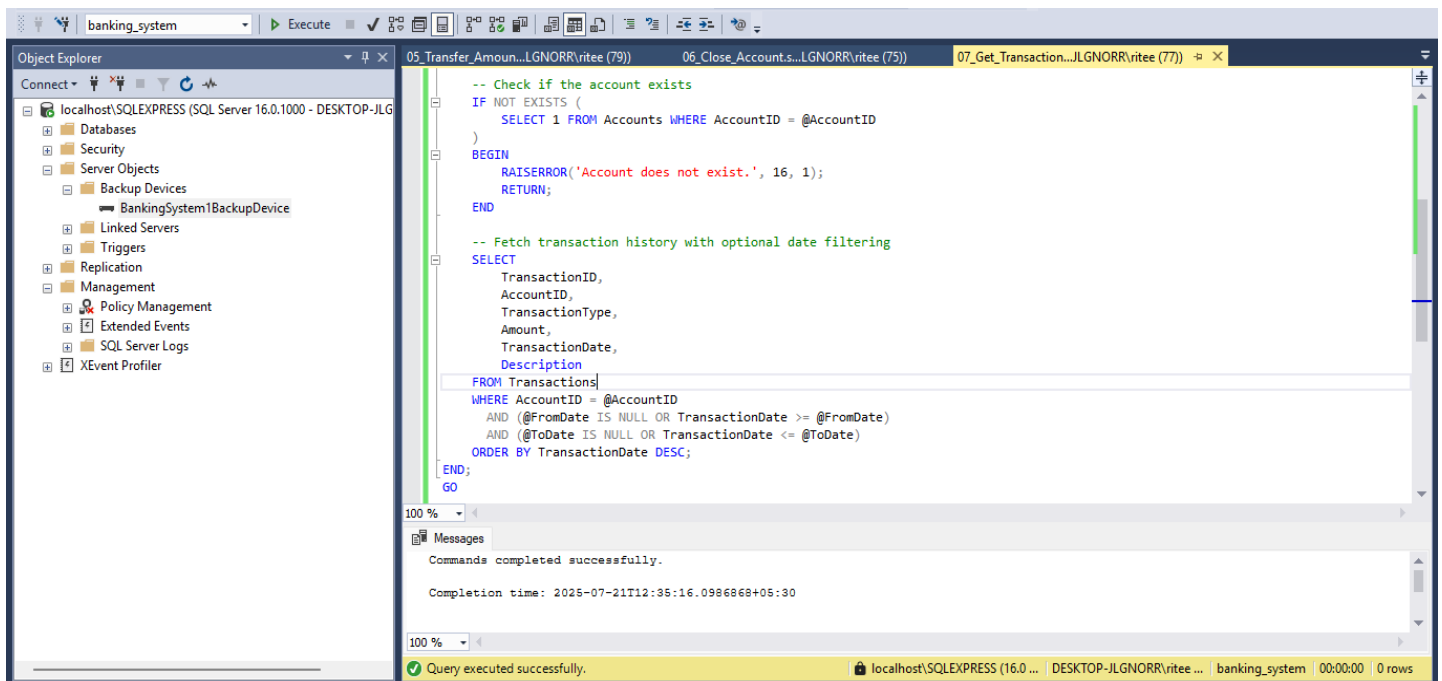
-- Fetch transaction history with optional date filtering
SELECT
```

```

TransactionID,
AccountID,
TransactionType,
Amount,
TransactionDate,
Description
FROM Transactions
WHERE AccountID = @AccountID
AND (@FromDate IS NULL OR TransactionDate >= @FromDate)
AND (@ToDate IS NULL OR TransactionDate <= @ToDate)
ORDER BY TransactionDate DESC;

END; GO

```



08_Add_New_Branch.sql

```

USE banking_system; GO

-- Drop the procedure if it already exists IF OBJECT_ID('sp_Add_New_Branch', 'P') IS NOT NULL DROP
PROCEDURE sp_Add_New_Branch; GO

-- Create the procedure CREATE PROCEDURE sp_Add_New_Branch @BranchName NVARCHAR(100),
@Location NVARCHAR(150), @ContactNumber VARCHAR(15) = NULL, @ManagerName NVARCHAR(100) =
NULL AS BEGIN SET NOCOUNT ON;

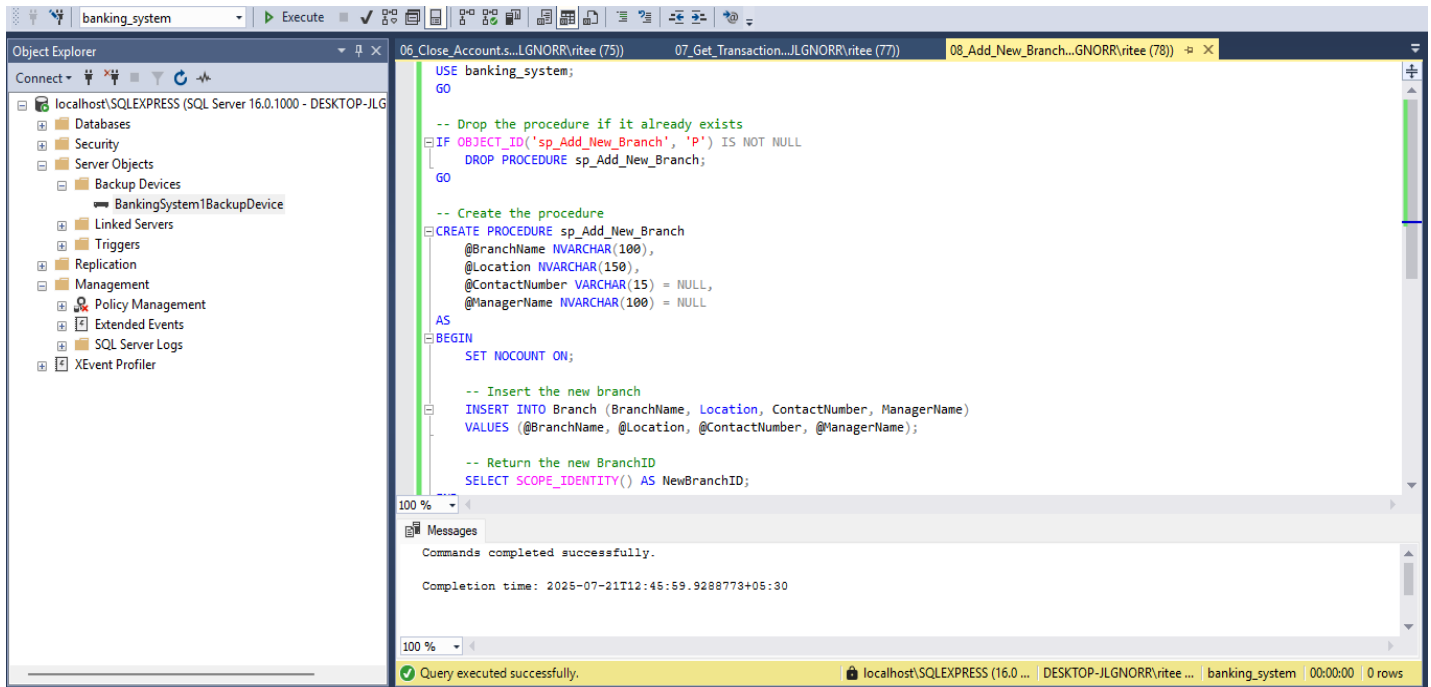
-- Insert the new branch
INSERT INTO Branch (BranchName, Location, ContactNumber, ManagerName)
VALUES (@BranchName, @Location, @ContactNumber, @ManagerName);

-- Return the new BranchID

```

```
SELECT SCOPE_IDENTITY() AS NewBranchID;
```

```
END; GO
```



09_Get_All_Branches.sql

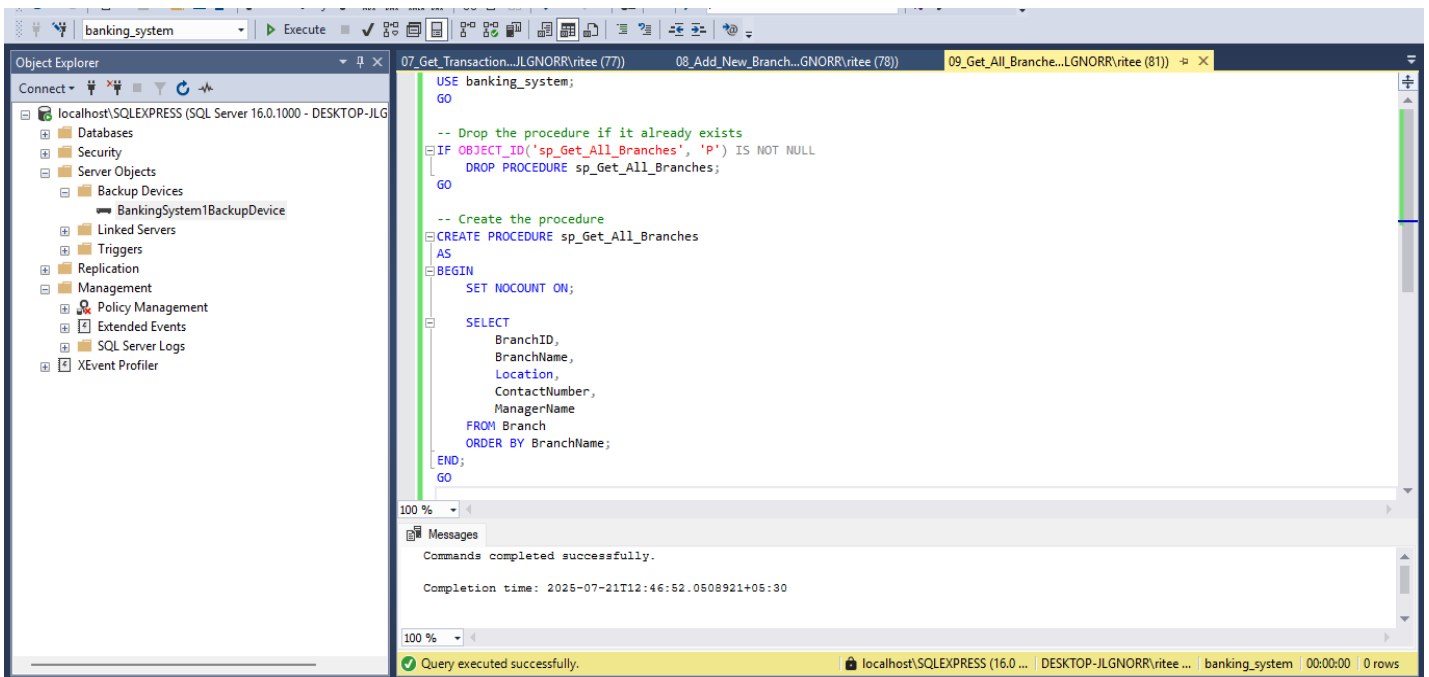
```
USE banking_system; GO
```

```
-- Drop the procedure if it already exists IF OBJECT_ID('sp_Get_All_Branches', 'P') IS NOT NULL DROP  
PROCEDURE sp_Get_All_Branches; GO
```

```
-- Create the procedure CREATE PROCEDURE sp_Get_All_Branches AS BEGIN SET NOCOUNT ON;
```

```
SELECT  
    BranchID,  
    BranchName,  
    Location,  
    ContactNumber,  
    ManagerName  
FROM Branch  
ORDER BY BranchName;
```

```
END; GO
```



10_Apply_Loan.sql

USE banking_system; GO

-- Drop the procedure if it already exists IF OBJECT_ID('sp_Apply_Loan', 'P') IS NOT NULL DROP PROCEDURE sp_Apply_Loan; GO

-- Create the procedure CREATE PROCEDURE sp_Apply_Loan @CustomerID INT, @BranchID INT, @LoanAmount DECIMAL(18,2), @InterestRate DECIMAL(5,2), @TermMonths INT AS BEGIN SET NOCOUNT ON;

DECLARE @StartDate DATE = GETDATE();

DECLARE @EndDate DATE = DATEADD(MONTH, @TermMonths, @StartDate);

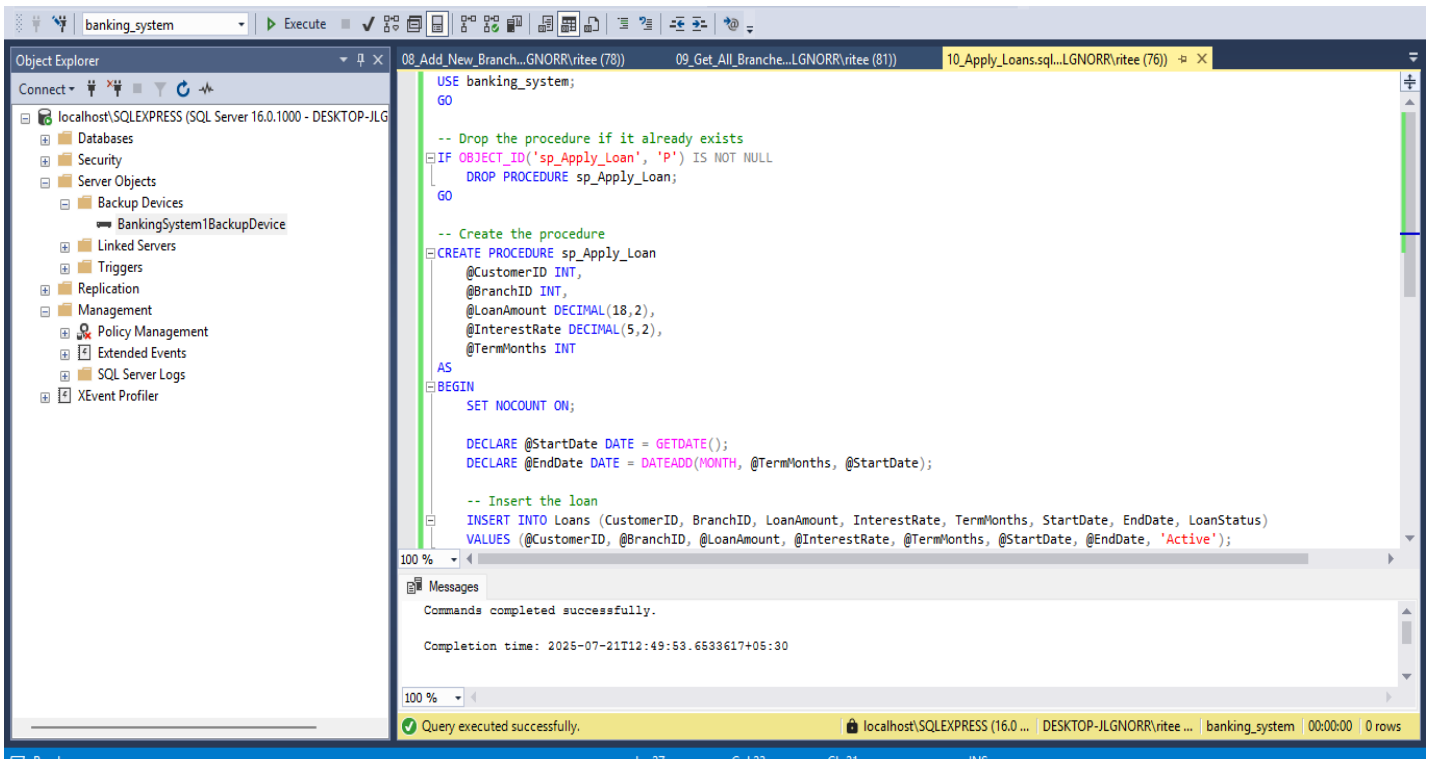
-- Insert the loan

INSERT INTO Loans (CustomerID, BranchID, LoanAmount, InterestRate, TermMonths, StartDate, EndDate, LoanStatus)

VALUES (@CustomerID, @BranchID, @LoanAmount, @InterestRate, @TermMonths, @StartDate, @EndDate, 'Active');

PRINT '✅ Loan application successful.';

END; GO



11_Approve_Loan.sql

```
USE banking_system; GO
```

```
-- Drop the procedure if it already exists IF OBJECT_ID('sp_Approve_Loan', 'P') IS NOT NULL DROP PROCEDURE  
sp_Approve_Loan; GO
```

```
-- Create the procedure CREATE PROCEDURE sp_Approve_Loan @LoanID INT AS BEGIN SET NOCOUNT ON;
```

```
-- Check if the loan exists and is not already approved
```

```
IF EXISTS (  
    SELECT 1 FROM Loans  
    WHERE LoanID = @LoanID AND LoanStatus <> 'Active'  
)
```

```
BEGIN
```

```
-- Approve the loan
```

```
UPDATE Loans
```

```
SET LoanStatus = 'Active'
```

```
WHERE LoanID = @LoanID;
```

```
PRINT ' Loan approved successfully.';
```

```
END
```

```
ELSE
```

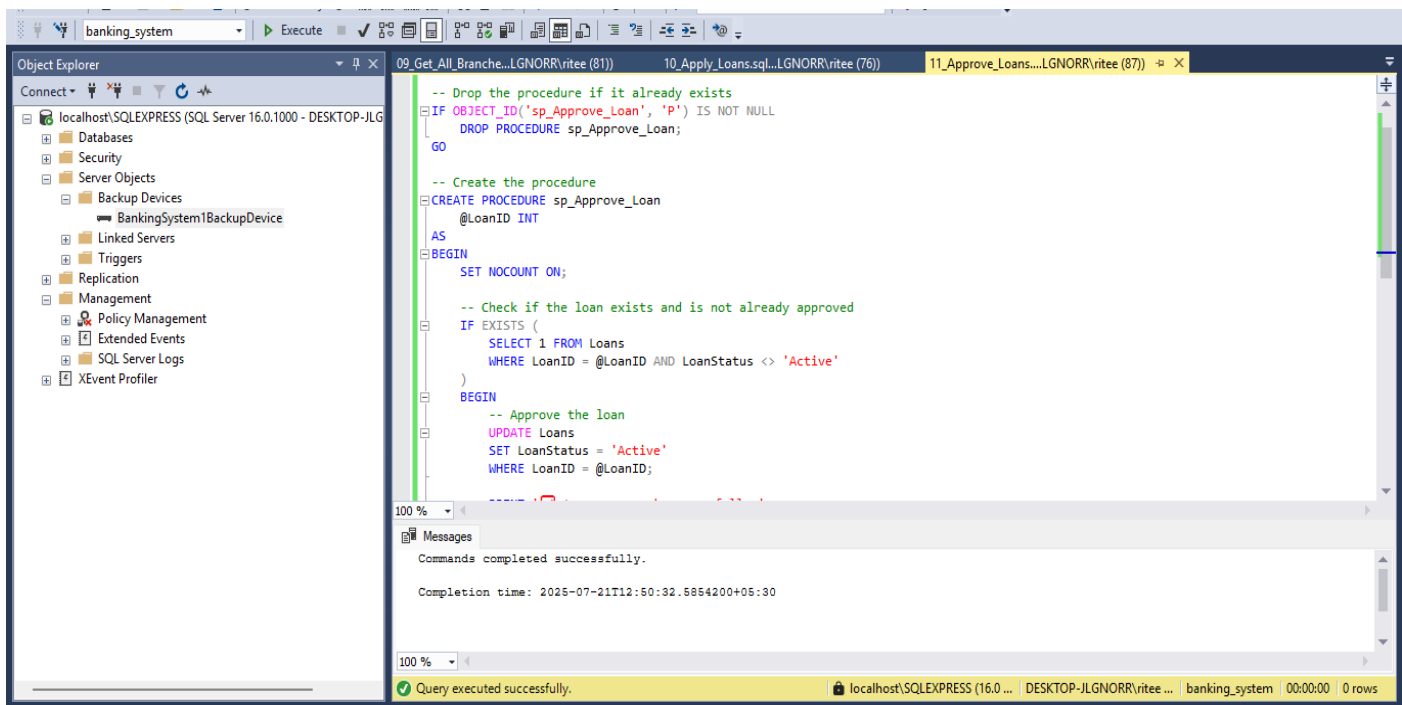
```
BEGIN
```

```
PRINT ' Loan either does not exist or is already active.';
```

```
END
```

```
END;
```

GO



12_Close_Loan.sql

```
USE banking_system; GO
```

```
-- Drop the procedure if it exists IF OBJECT_ID('sp_Close_Loan', 'P') IS NOT NULL DROP PROCEDURE  
sp_Close_Loan; GO
```

```
-- Create the stored procedure CREATE PROCEDURE sp_Close_Loan @LoanID INT AS BEGIN SET  
NOCOUNT ON;
```

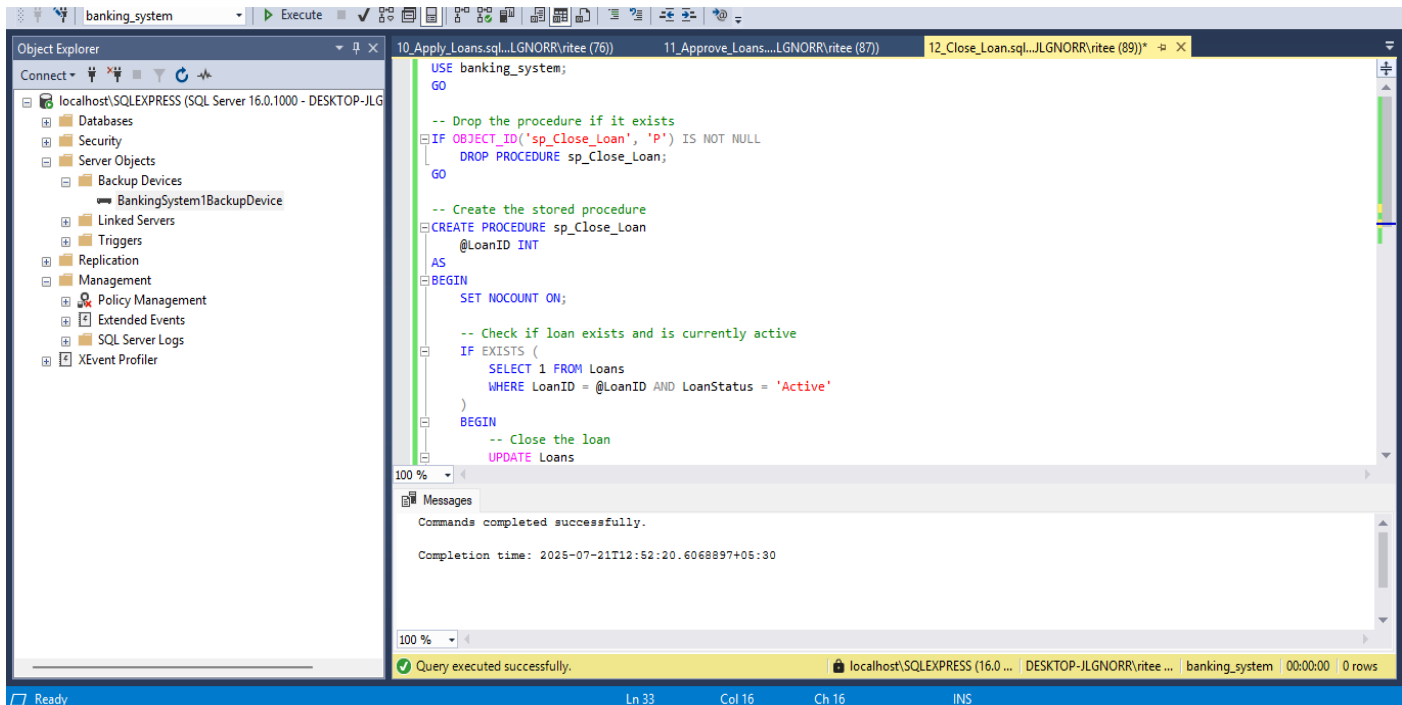
```
-- Check if loan exists and is currently active  
IF EXISTS (  
    SELECT 1 FROM Loans  
    WHERE LoanID = @LoanID AND LoanStatus = 'Active'  
)  
BEGIN  
    -- Close the loan  
    UPDATE Loans  
    SET  
        LoanStatus = 'Closed',  
        EndDate = GETDATE()  
    WHERE LoanID = @LoanID;  
  
    PRINT ' Loan closed successfully.';  
END  
ELSE
```

```

BEGIN
    PRINT ' Loan either does not exist or is not active.';
END

END; GO

```



13_Get_Loan_History.sql

```

USE banking_system; GO

-- Drop the procedure if it already exists IF OBJECT_ID('sp_Get_Loan_History', 'P') IS NOT NULL DROP
PROCEDURE sp_Get_Loan_History; GO

-- Create the procedure CREATE PROCEDURE sp_Get_Loan_History @CustomerID INT, @Status VARCHAR(20)
= NULL, -- Optional filter: 'Active' or 'Closed' @FromDate DATE = NULL, -- Optional: Filter StartDate @ToDate
DATE = NULL -- Optional: Filter EndDate AS BEGIN SET NOCOUNT ON;

-- Check if the customer exists
IF NOT EXISTS (
    SELECT 1 FROM Customers WHERE CustomerID = @CustomerID
)
BEGIN
    RAISERROR('Customer does not exist.', 16, 1);
    RETURN;
END

-- Fetch loans for the customer with optional filters
SELECT
    LoanID,

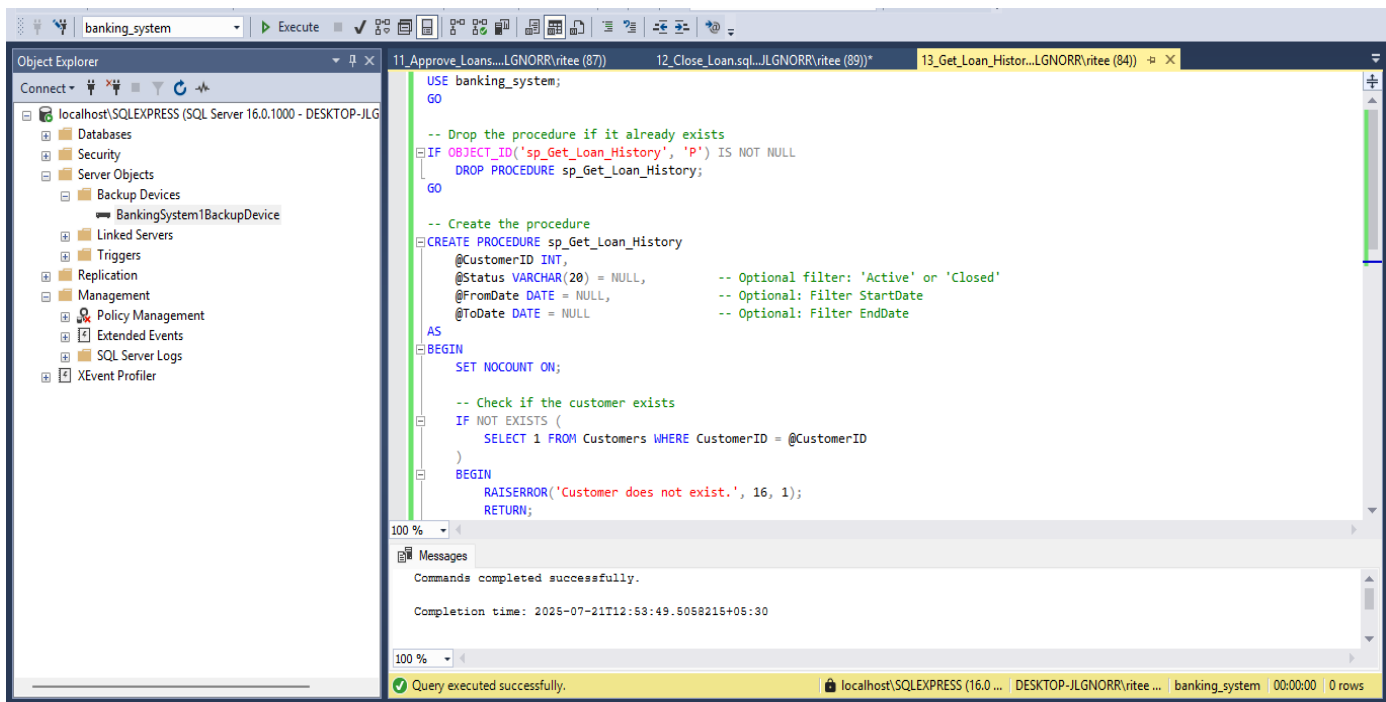
```

```

BranchID,
LoanAmount,
InterestRate,
TermMonths,
StartDate,
EndDate,
LoanStatus
FROM Loans
WHERE CustomerID = @CustomerID
AND (@Status IS NULL OR LoanStatus = @Status)
AND (@FromDate IS NULL OR StartDate >= @FromDate)
AND (@ToDate IS NULL OR StartDate <= @ToDate)
ORDER BY StartDate DESC;

END; GO

```



6. Testing and sample data

01_Insert_Sample_Customers.sql

```

USE banking_system;
GO

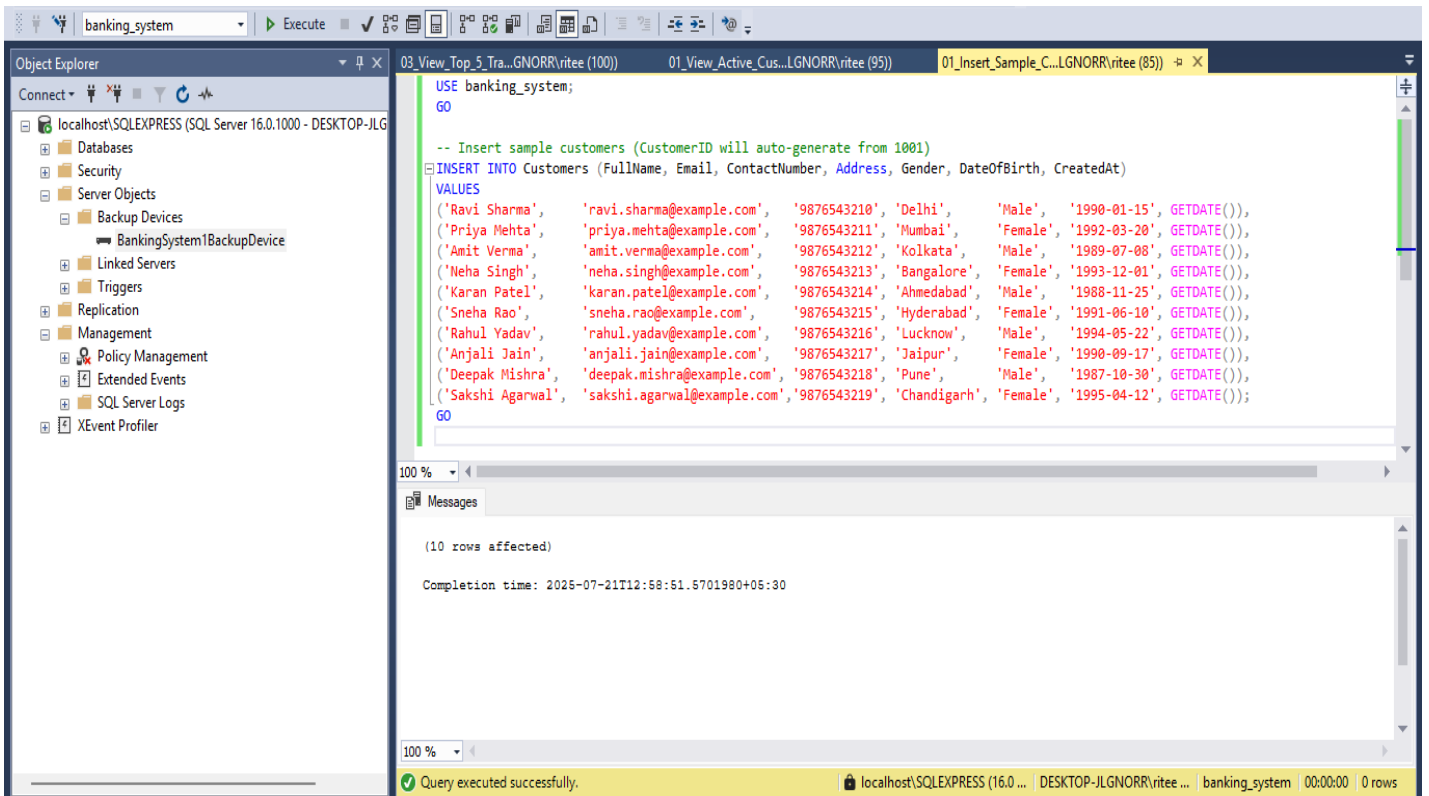
-- Insert sample customers (CustomerID will auto-generate from 1001)
INSERT INTO Customers (FullName, Email, ContactNumber, Address, Gender, DateOfBirth, CreatedAt)
VALUES
('Ravi Sharma', 'ravi.sharma@example.com', '9876543210', 'Delhi', 'Male', '1990-01-15', GETDATE()),
('Priya Mehta', 'priya.mehta@example.com', '9876543211', 'Mumbai', 'Female', '1992-03-20', GETDATE()),
('Amit Verma', 'amit.verma@example.com', '9876543212', 'Kolkata', 'Male', '1989-07-08', GETDATE()),

```

```

('Neha Singh', 'neha.singh@example.com', '9876543213', 'Bangalore', 'Female', '1993-12-01', GETDATE()),
('Karan Patel', 'karan.patel@example.com', '9876543214', 'Ahmedabad', 'Male', '1988-11-25', GETDATE()),
('Sneha Rao', 'sneha.rao@example.com', '9876543215', 'Hyderabad', 'Female', '1991-06-10', GETDATE()),
('Rahul Yadav', 'rahul.yadav@example.com', '9876543216', 'Lucknow', 'Male', '1994-05-22', GETDATE()),
('Anjali Jain', 'anjali.jain@example.com', '9876543217', 'Jaipur', 'Female', '1990-09-17', GETDATE()),
('Deepak Mishra', 'deepak.mishra@example.com', '9876543218', 'Pune', 'Male', '1987-10-30', GETDATE()),
('Sakshi Agarwal', 'sakshi.agarwal@example.com', '9876543219', 'Chandigarh', 'Female', '1995-04-12',
GETDATE());
GO

```

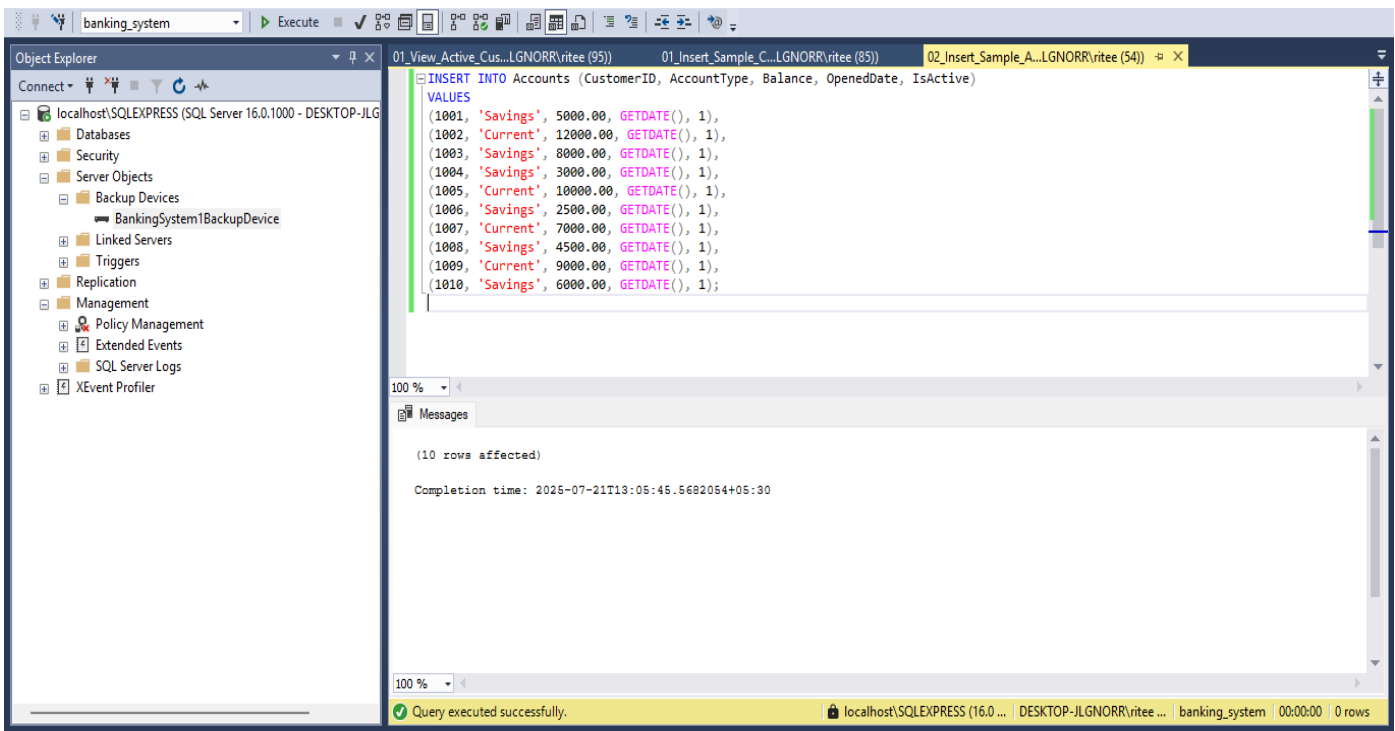


02_Insert_Sample_Accounts.sql

```

INSERT INTO Accounts (CustomerID, AccountType, Balance, OpenedDate, IsActive)
VALUES
(1001, 'Savings', 5000.00, GETDATE(), 1),
(1002, 'Current', 12000.00, GETDATE(), 1),
(1003, 'Savings', 8000.00, GETDATE(), 1),
(1004, 'Savings', 3000.00, GETDATE(), 1),
(1005, 'Current', 10000.00, GETDATE(), 1),
(1006, 'Savings', 2500.00, GETDATE(), 1),
(1007, 'Current', 7000.00, GETDATE(), 1),
(1008, 'Savings', 4500.00, GETDATE(), 1),
(1009, 'Current', 9000.00, GETDATE(), 1),
(1010, 'Savings', 6000.00, GETDATE(), 1);

```



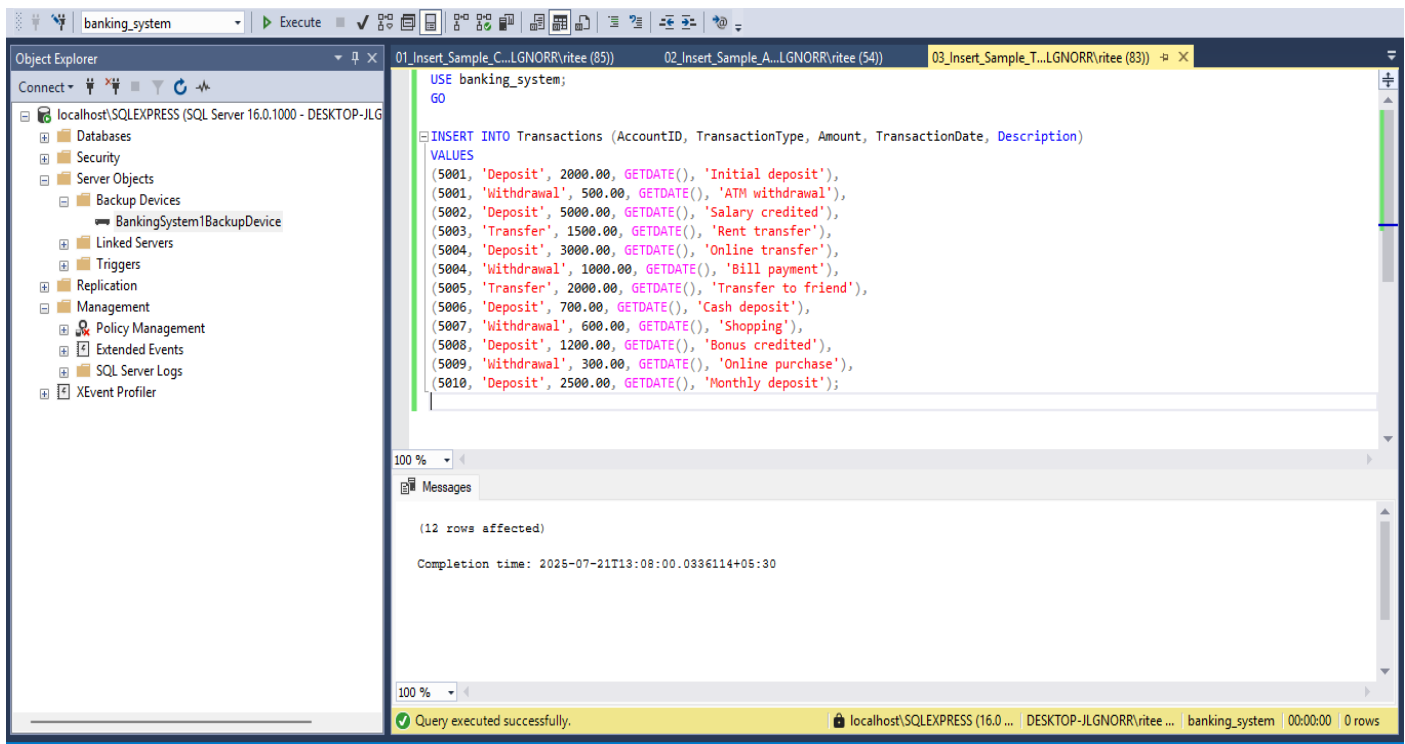
03_Insert_Sample_Transactions.sql

USE banking_system;
GO

```

INSERT INTO Transactions (AccountID, TransactionType, Amount, TransactionDate, Description)
VALUES
(5001, 'Deposit', 2000.00, GETDATE(), 'Initial deposit'),
(5001, 'Withdrawal', 500.00, GETDATE(), 'ATM withdrawal'),
(5002, 'Deposit', 5000.00, GETDATE(), 'Salary credited'),
(5003, 'Transfer', 1500.00, GETDATE(), 'Rent transfer'),
(5004, 'Deposit', 3000.00, GETDATE(), 'Online transfer'),
(5004, 'Withdrawal', 1000.00, GETDATE(), 'Bill payment'),
(5005, 'Transfer', 2000.00, GETDATE(), 'Transfer to friend'),
(5006, 'Deposit', 700.00, GETDATE(), 'Cash deposit'),
(5007, 'Withdrawal', 600.00, GETDATE(), 'Shopping'),
(5008, 'Deposit', 1200.00, GETDATE(), 'Bonus credited'),
(5009, 'Withdrawal', 300.00, GETDATE(), 'Online purchase'),
(5010, 'Deposit', 2500.00, GETDATE(), 'Monthly deposit');

```



04_Test_Transfer_Procedure.sql

```
USE banking_system;  
  
GO  
  
-- TEST CASE 1: Successful Transfer  
  
PRINT 'Test 1: Successful transfer from 5001 to 5002';  
EXEC sp_Transfer_Amount  
@FromAccountID = 5001,  
@ToAccountID = 5002,  
@Amount = 200.00,  
@Description = 'Test Case: Rent payment';  
  
-- TEST CASE 2: Insufficient Balance  
  
PRINT 'Test 2: Insufficient balance';  
EXEC sp_Transfer_Amount  
@FromAccountID = 5001,  
@ToAccountID = 5002,  
@Amount = 9999999.99,  
@Description = 'Test Case: Overdraft attempt';  
  
-- TEST CASE 3: Sender Account Does Not Exist  
  
PRINT 'Test 3: Sender account invalid';  
EXEC sp_Transfer_Amount  
@FromAccountID = 9999,  
@ToAccountID = 5002,  
@Amount = 100.00,  
@Description = 'Test Case: Invalid sender';
```

-- TEST CASE 4: Receiver Account Does Not Exist

```
PRINT ' Test 4: Receiver account invalid';
EXEC sp_Transfer_Amount
@FromAccountID = 5001,
@ToAccountID = 9999,
@Amount = 100.00,
@Description = 'Test Case: Invalid receiver';
```

-- TEST CASE 5: Receiver Account Inactive

```
PRINT ' Test 5: Receiver account inactive';
UPDATE Accounts SET IsActive = 0 WHERE AccountID = 5002;
EXEC sp_Transfer_Amount
@FromAccountID = 5001,
@ToAccountID = 5002,
@Amount = 50.00,
@Description = 'Test Case: Receiver inactive';
UPDATE Accounts SET IsActive = 1 WHERE AccountID = 5002;
```

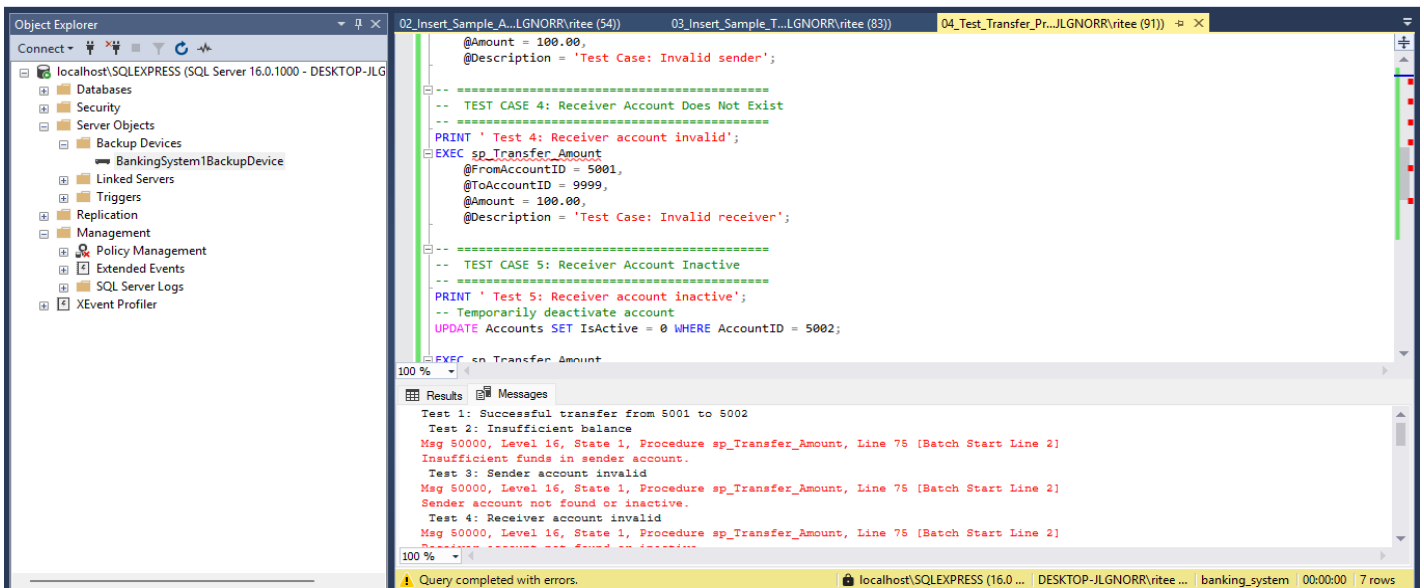
-- TEST CASE 6: Sender Account Inactive

```
PRINT ' Test 6: Sender account inactive';
-- Temporarily deactivate account
```

```
UPDATE Accounts SET IsActive = 0 WHERE AccountID = 5001;
EXEC sp_Transfer_Amount
@FromAccountID = 5001,
@ToAccountID = 5002,
@Amount = 50.00,
@Description = 'Test Case: Sender inactive';
```

-- Reactivate the account

```
UPDATE Accounts SET IsActive = 1 WHERE AccountID = 5001;
-- Final Check: Balances and Transactions
PRINT ' Final Balance Check for Accounts 5001 and 5002'; SELECT AccountID, Balance, IsActive FROM
Accounts WHERE AccountID IN (5001, 5002);
PRINT ' Last 5 Transactions'; SELECT TOP 5 * FROM Transactions ORDER BY TransactionID DESC;
```



05_Test_Account_Closure.sql

USE banking_system; GO

INSERT INTO Customers (FullName, Email, ContactNumber, Address, Gender, DateOfBirth, CreatedAt) VALUES ('John Doe', 'john.doe@example.com', '1234567890', '123 Main St', 'Male', '1990-01-01', GETDATE()); DECLARE @Cust1 INT = SCOPE_IDENTITY();

INSERT INTO Customers (FullName, Email, ContactNumber, Address, Gender, DateOfBirth, CreatedAt) VALUES ('Jane Smith', 'jane.smith@example.com', '9876543210', '456 Elm St', 'Female', '1992-05-15', GETDATE()); DECLARE @Cust2 INT = SCOPE_IDENTITY();

-- Test 1: Non-existent account

PRINT 'Test 1: Non-existing account'; EXEC sp_Close_Account @AccountID = 9999;

-- Should fail with error

-- Test 2: Account with balance \neq 0

PRINT 'Test 2: Account with non-zero balance'; INSERT INTO Accounts (CustomerID, AccountType, Balance, OpenedDate, IsActive) VALUES (@Cust1, 'Savings', 1000.00, GETDATE(), 1); DECLARE @AccWithBalance INT = SCOPE_IDENTITY();

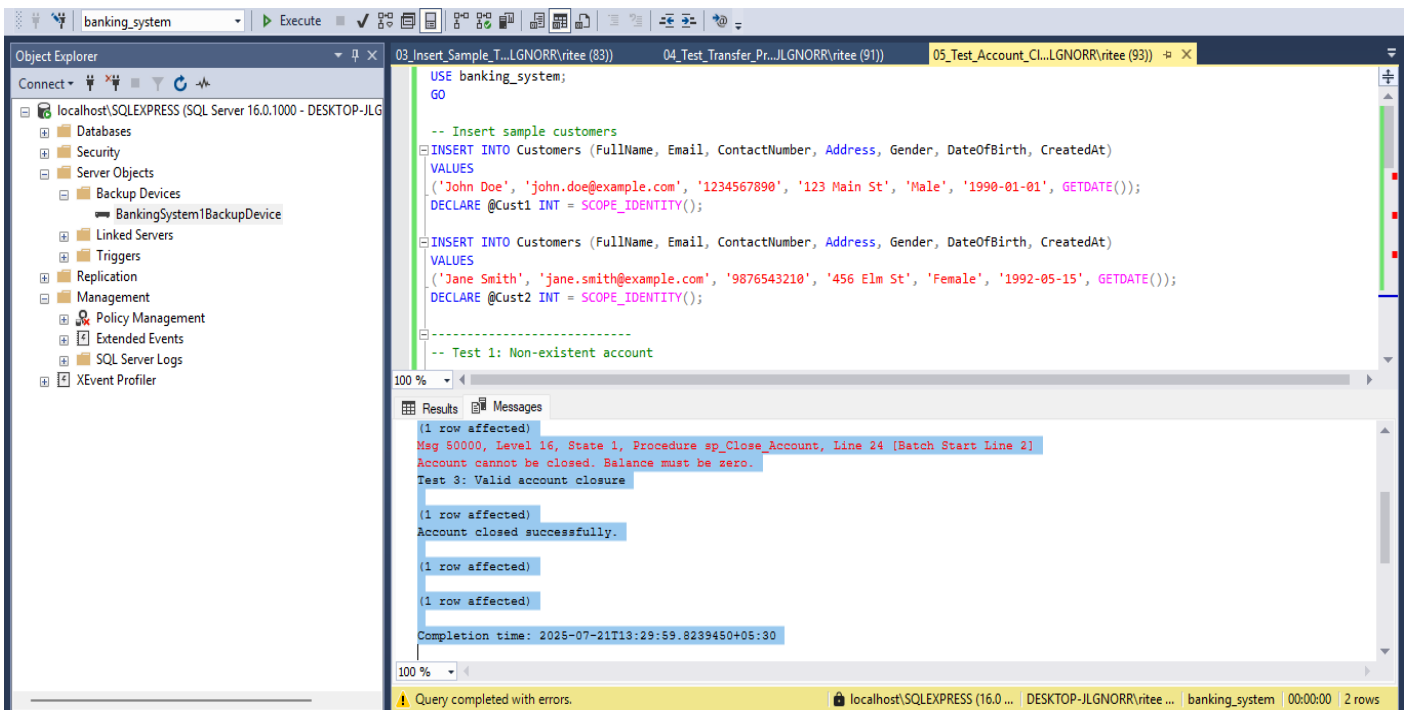
EXEC sp_Close_Account @AccountID = @AccWithBalance; -- Should fail due to non-zero balance

-- Test 3: Account with balance = 0 (valid closure)

PRINT 'Test 3: Valid account closure'; INSERT INTO Accounts (CustomerID, AccountType, Balance, OpenedDate, IsActive) VALUES (@Cust2, 'Checking', 0.00, GETDATE(), 1); DECLARE @ZeroBalanceAcc INT = SCOPE_IDENTITY();

EXEC sp_Close_Account @AccountID = @ZeroBalanceAcc; -- Should succeed SELECT AccountID, IsActive FROM Accounts WHERE AccountID = @ZeroBalanceAcc;

SELECT * FROM Transactions WHERE AccountID = @ZeroBalanceAcc AND TransactionType = 'Closure';



06_Test_Loan_Application.sql

```
USE banking_system; GO
-- Drop the procedure if it already exists IF OBJECT_ID('sp_Apply_Loan', 'P') IS NOT NULL DROP PROCEDURE
sp_Apply_Loan; GO
-- Create the loan application procedure CREATE PROCEDURE sp_Apply_Loan @CustomerID INT, @BranchID
INT, @LoanAmount DECIMAL(18,2), @InterestRate DECIMAL(5,2), @TermMonths INT AS BEGIN SET NOCOUNT
ON;
-- Validate customer exists
IF NOT EXISTS (SELECT 1 FROM Customers WHERE CustomerID = @CustomerID)
BEGIN
    RAISERROR(' Customer does not exist.', 16, 1);
    RETURN;
END

-- Validate branch exists
IF NOT EXISTS (SELECT 1 FROM Branch WHERE BranchID = @BranchID)
BEGIN
    RAISERROR(' Branch does not exist.', 16, 1);
    RETURN;
END

-- Validate positive values
IF @LoanAmount <= 0 OR @InterestRate <= 0 OR @TermMonths <= 0
BEGIN
    RAISERROR(' LoanAmount, InterestRate, and TermMonths must be positive.', 16, 1);
    RETURN;
END

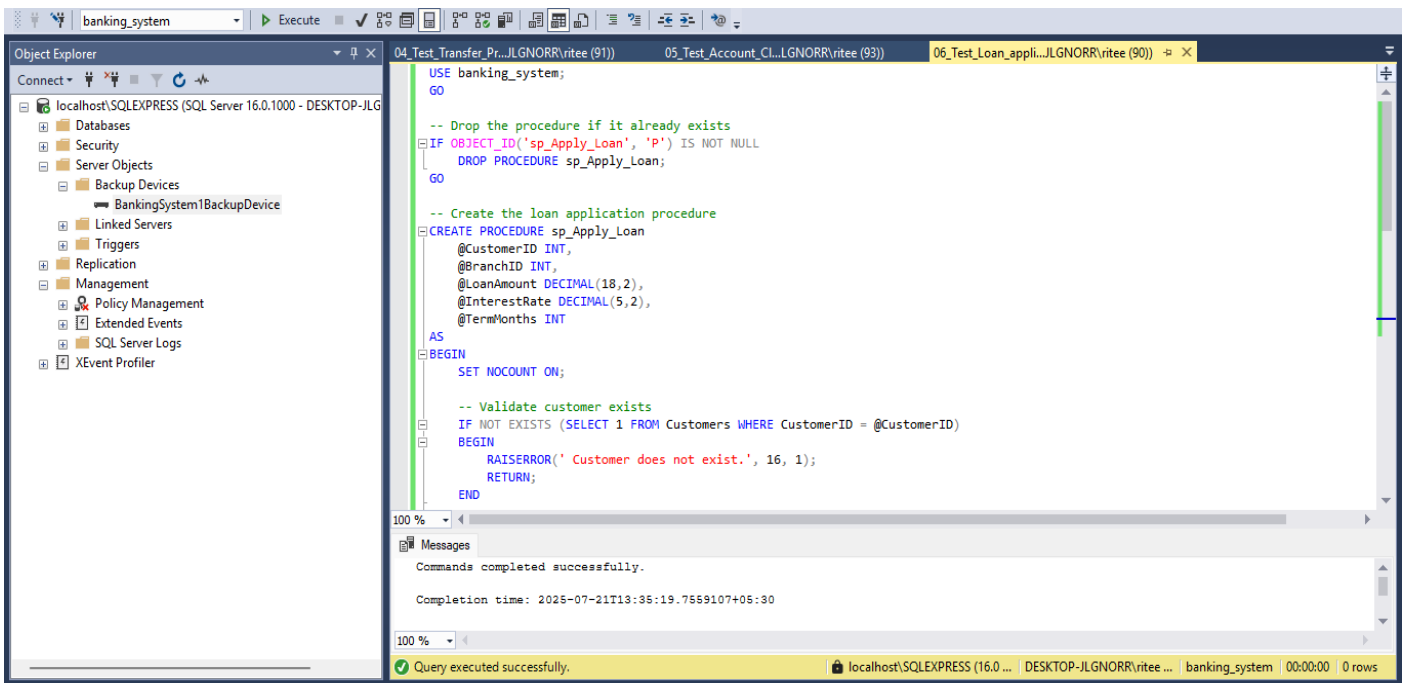
DECLARE @StartDate DATE = GETDATE();
DECLARE @EndDate DATE = DATEADD(MONTH, @TermMonths, @StartDate);

-- Insert loan application
```

```
INSERT INTO Loans (CustomerID, BranchID, LoanAmount, InterestRate, TermMonths,
StartDate, EndDate, LoanStatus)
VALUES (@CustomerID, @BranchID, @LoanAmount, @InterestRate, @TermMonths, @StartDate,
@EndDate, 'Active');
```

```
PRINT ' Loan application successful.';
```

```
END; GO
```



07_Test_Loan_Approval.sql

```
USE banking_system;
```

```
GO
```

```
PRINT 'Test 1: Approving a Loan for an Existing Customer';
```

```
DECLARE @CustomerID INT = 1001;
```

```
DECLARE @BranchID INT = 1;
```

```

INSERT INTO Loans (

    CustomerID, BranchID, LoanAmount, InterestRate, TermMonths, StartDate, EndDate, LoanStatus

)

VALUES (

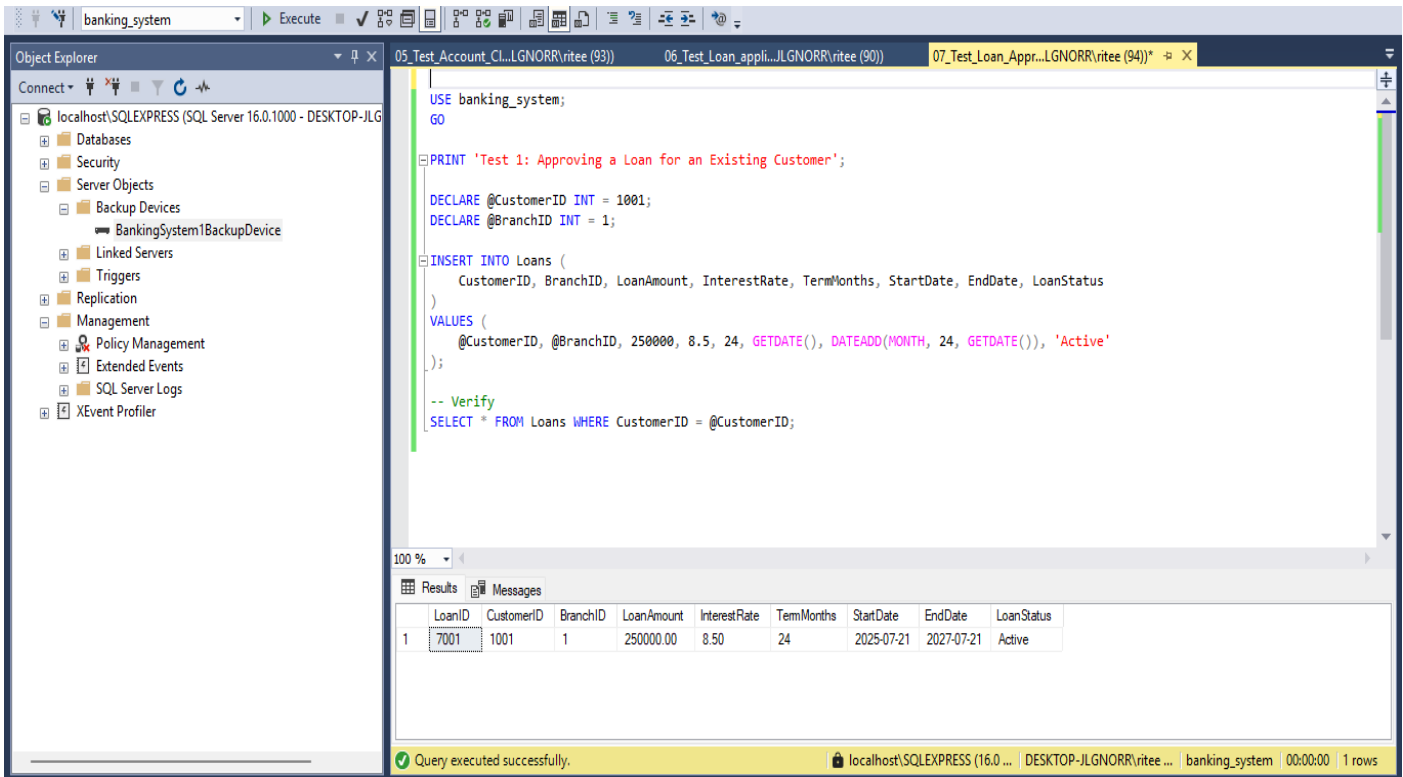
    @CustomerID, @BranchID, 250000, 8.5, 24, GETDATE(), DATEADD(MONTH, 24, GETDATE()), 'Active'

);

-- Verify

SELECT * FROM Loans WHERE CustomerID = @CustomerID;

```



The screenshot shows the SQL Server Enterprise Manager interface. The Object Explorer on the left shows the 'banking_system' database. The Query window on the right contains the following SQL code:

```

USE banking_system;
GO

PRINT 'Test 1: Approving a Loan for an Existing Customer';

DECLARE @CustomerID INT = 1001;
DECLARE @BranchID INT = 1;

INSERT INTO Loans (
    CustomerID, BranchID, LoanAmount, InterestRate, TermMonths, StartDate, EndDate, LoanStatus
)
VALUES (
    @CustomerID, @BranchID, 250000, 8.5, 24, GETDATE(), DATEADD(MONTH, 24, GETDATE()), 'Active'
);

-- Verify
SELECT * FROM Loans WHERE CustomerID = @CustomerID;

```

The Results pane at the bottom shows the output of the query:

LoanID	CustomerID	BranchID	LoanAmount	InterestRate	TermMonths	StartDate	EndDate	LoanStatus
7001	1001	1	250000.00	8.50	24	2025-07-21	2027-07-21	Active

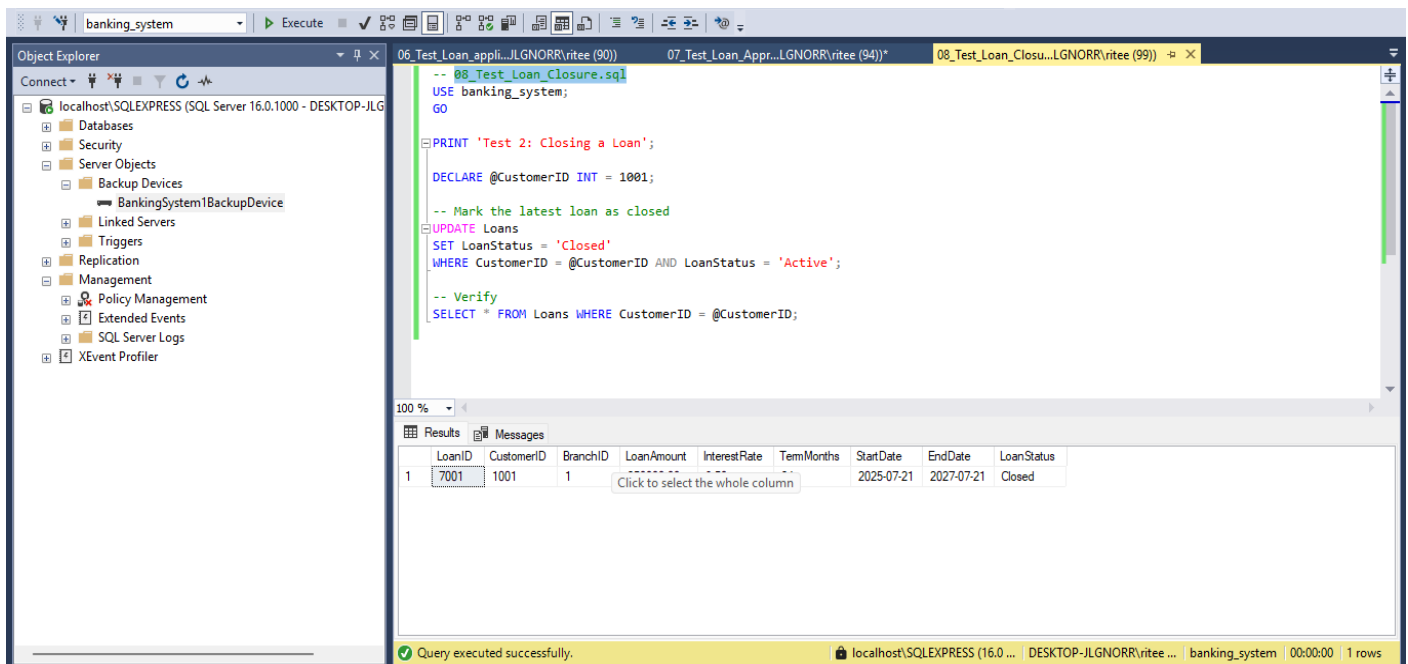
The Messages pane at the bottom shows the message: 'Query executed successfully.'

08_Test_Loan_Closure.sql

```

-- 08_Test_Loan_Closure.sql USE banking_system; GO
PRINT 'Test 2: Closing a Loan';
DECLARE @CustomerID INT = 1001;
-- Mark the latest loan as closed UPDATE Loans SET LoanStatus = 'Closed' WHERE CustomerID = @CustomerID
AND LoanStatus = 'Active';
-- Verify SELECT * FROM Loans WHERE CustomerID = @CustomerID;

```



09_Insert_Sample_Branches.sql

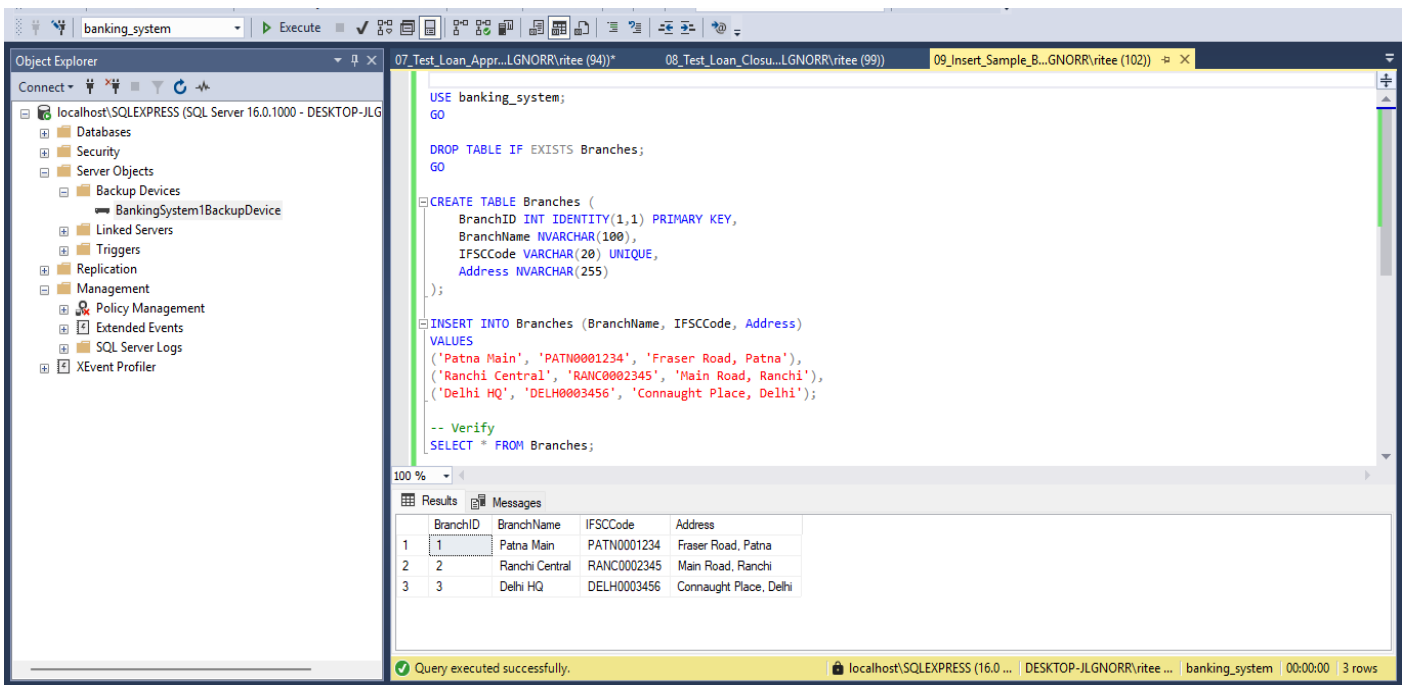
```
USE banking_system;
GO
```

```
DROP TABLE IF EXISTS Branches;
GO
```

```
CREATE TABLE Branches (
    BranchID INT IDENTITY(1,1) PRIMARY KEY,
    BranchName NVARCHAR(100),
    IFSCCode VARCHAR(20) UNIQUE,
    Address NVARCHAR(255)
);
```

```
INSERT INTO Branches (BranchName, IFSCCode, Address)
VALUES
('Patna Main', 'PATN0001234', 'Fraser Road, Patna'),
('Ranchi Central', 'RANC0002345', 'Main Road, Ranchi'),
('Delhi HQ', 'DELH0003456', 'Connaught Place, Delhi');
```

```
-- Verify
SELECT * FROM Branches;
```



7. Views and Reports

01_View_Active_Customers.sql

This view provides a consolidated list of all active bank accounts along with corresponding customer and branch details. It helps staff to quickly identify operational accounts and manage them effectively.

```
USE banking_system;
GO

-- Tracking table for transaction logs
IF OBJECT_ID('TransactionLogs') IS NOT NULL
    DROP TABLE TransactionLogs;
GO

CREATE TABLE TransactionLogs (
    LogID INT IDENTITY(1,1) PRIMARY KEY,
    TransactionID INT,
    AccountID INT,
    TransactionType VARCHAR(20),
    Amount DECIMAL(18,2),
    LoggedAt DATETIME DEFAULT GETDATE()
);
GO

-- Trigger to log every new transaction
CREATE OR ALTER TRIGGER trg_TrackTransaction
ON Transactions
AFTER INSERT
AS
BEGIN
```

```

INSERT INTO TransactionLogs (TransactionID, AccountID, TransactionType, Amount)
SELECT
    i.TransactionID, i.AccountID, i.TransactionType, i.Amount
FROM inserted i;
END;
GO

```

-- Test the trigger (insert a transaction, then SELECT * FROM TransactionLogs)

The screenshot shows the SQL Server Enterprise Manager interface. The left pane displays the 'Object Explorer' with the 'banking_system' database selected. The right pane shows the 'Query Editor' with the following SQL script:

```

USE banking_system;
GO

CREATE OR ALTER VIEW View_Active_Customers AS
SELECT
    c.CustomerID,
    c.FullName,
    c.Email,
    c.ContactNumber,
    a.AccountID,
    a.AccountType,
    a.Balance,
    a.OpenedDate,
    CASE
        WHEN a.IsActive = 1 THEN 'Active'
        ELSE 'Closed'
    END AS AccountStatus
FROM Customers c JOIN Accounts a ON c.CustomerID = a.CustomerID
WHERE a.IsActive = 1;

```

Below the script, the 'Results' pane displays a table with 11 rows of data:

CustomerID	FullName	Email	ContactNumber	AccountID	AccountType	Balance	OpenedDate	AccountStatus
1001	Ravi Sharma	ravi.sharma@example.com	9876543210	5001	Savings	500.00	2025-07-21	Active
1003	Amit Verma	amit.verma@example.com	9876543212	5003	Savings	8000.00	2025-07-21	Active
1004	Neha Singh	neha.singh@example.com	9876543213	5004	Savings	3000.00	2025-07-21	Active
1005	Karan Patel	karan.patel@example.com	9876543214	5005	Current	10000.00	2025-07-21	Active
1006	Sneha Rao	sneha.rao@example.com	9876543215	5006	Savings	2500.00	2025-07-21	Active
1007	Rahul Yadav	rahul.yadav@example.com	9876543216	5007	Current	7000.00	2025-07-21	Active
1008	Anjali Jain	anjali.jain@example.com	9876543217	5008	Savings	4500.00	2025-07-21	Active
1009	Deepak Mishra	deepak.mishra@example.com	9876543218	5009	Current	9000.00	2025-07-21	Active
1010	Sakshi Agarwal	sakshi.agarwal@example.com	9876543219	5010	Savings	6000.00	2025-07-21	Active
1001	Ravi Sharma	ravi.sharma@example.com	9876543210	5011	Savings	5000.00	2025-07-21	Active
1002	Priya Mehta	priya.mehta@example.com	9876543211	5012	Current	12000.00	2025-07-21	Active

The status bar at the bottom indicates 'Query executed successfully.' and '20 rows'.

02_View_Account_Summary.sql

```

USE banking_system;
GO

```

```

CREATE OR ALTER VIEW View_Active_Customers AS SELECT c.CustomerID, c.FullName, c.Email,
c.ContactNumber, a.AccountID, a.AccountType, a.Balance, a.OpenedDate, CASE

```

```

WHEN a.IsActive = 1 THEN 'Active' ELSE 'Closed' END AS AccountStatus
FROM Customers c JOIN Accounts a ON c.CustomerID = a.CustomerID
WHERE a.IsActive = 1;
GO

```

-- Test the view SELECT * FROM View_Active_Customers;

The screenshot shows the SQL Server Enterprise Manager interface. The left pane displays the Object Explorer with the 'banking_system' database selected. The right pane shows the SQL Editor with the following code:

```
USE banking_system;
GO

CREATE OR ALTER VIEW View_Account_Summary AS
SELECT
    a.AccountID,
    c.FullName AS CustomerName,
    a.AccountType,
    a.Balance,
    a.OpenedDate,
    CASE
        WHEN a.IsActive = 1 THEN 'Active'
        ELSE 'Closed'
    END AS AccountStatus
FROM
    Accounts a
```

Below the code, the 'Results' tab shows the output of the query, displaying 11 rows of account data:

AccountID	CustomerName	AccountType	Balance	OpenedDate	AccountStatus
5001	Ravi Sharma	Savings	500.00	2025-07-21	Active
5002	Priya Mehta	Current	0.00	2025-07-21	Closed
5003	Amit Vema	Savings	8000.00	2025-07-21	Active
5004	Neha Singh	Savings	3000.00	2025-07-21	Active
5005	Karan Patel	Current	10000.00	2025-07-21	Active
5006	Sneha Rao	Savings	2500.00	2025-07-21	Active
5007	Rahul Yadav	Current	7000.00	2025-07-21	Active
5008	Anjali Jain	Savings	4500.00	2025-07-21	Active
5009	Deepak Mehra	Current	9000.00	2025-07-21	Active
5010	Sakshi Agarwal	Savings	6000.00	2025-07-21	Active
5011	Ravi Sharma	Savings	5000.00	2025-07-21	Active

The status bar at the bottom indicates 'Query executed successfully.' and '22 rows'.

03_View_Top_5_Transactions.sql

```
USE banking_system; GO
```

```
CREATE OR ALTER VIEW View_Top_5_Transactions
```

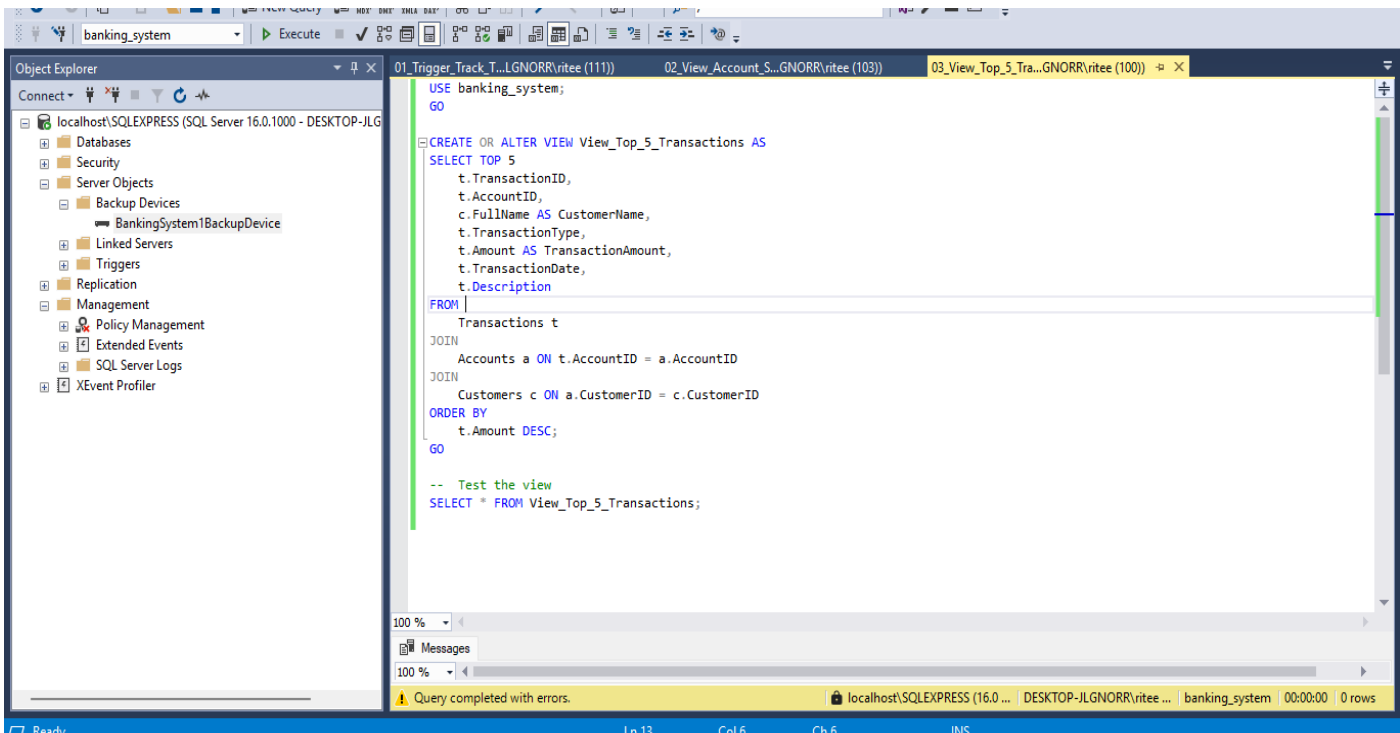
```
AS SELECT TOP 5 t.TransactionID, t.AccountID, c.FullName AS CustomerName, t.TransactionType, t.Amount
AS TransactionAmount, t.TransactionDate, t.Description
```

```
FROM Transactions t JOIN Accounts a ON t.AccountID = a.AccountID JOIN Customers c
```

```
ON a.CustomerID = c.CustomerID ORDER BY t.Amount DESC;
```

```
GO
```

```
-- Test the view SELECT * FROM View_Top_5_Transactions;
```



8. Triggers and Security

01_Trigger_Track_Transaction.sql This trigger automatically prevents any update operation that would result in a negative account balance. It ensures no overdraft is allowed, thereby enforcing business rules and data integrity.

```
USE banking_system;
GO

-- Tracking table for transaction logs
IF OBJECT_ID('TransactionLogs') IS NOT NULL
    DROP TABLE TransactionLogs;
GO

CREATE TABLE TransactionLogs (
    LogID INT IDENTITY(1,1) PRIMARY KEY,
    TransactionID INT,
    AccountID INT,
    TransactionType VARCHAR(20),
    Amount DECIMAL(18,2),
    LoggedAt DATETIME DEFAULT GETDATE()
);
GO

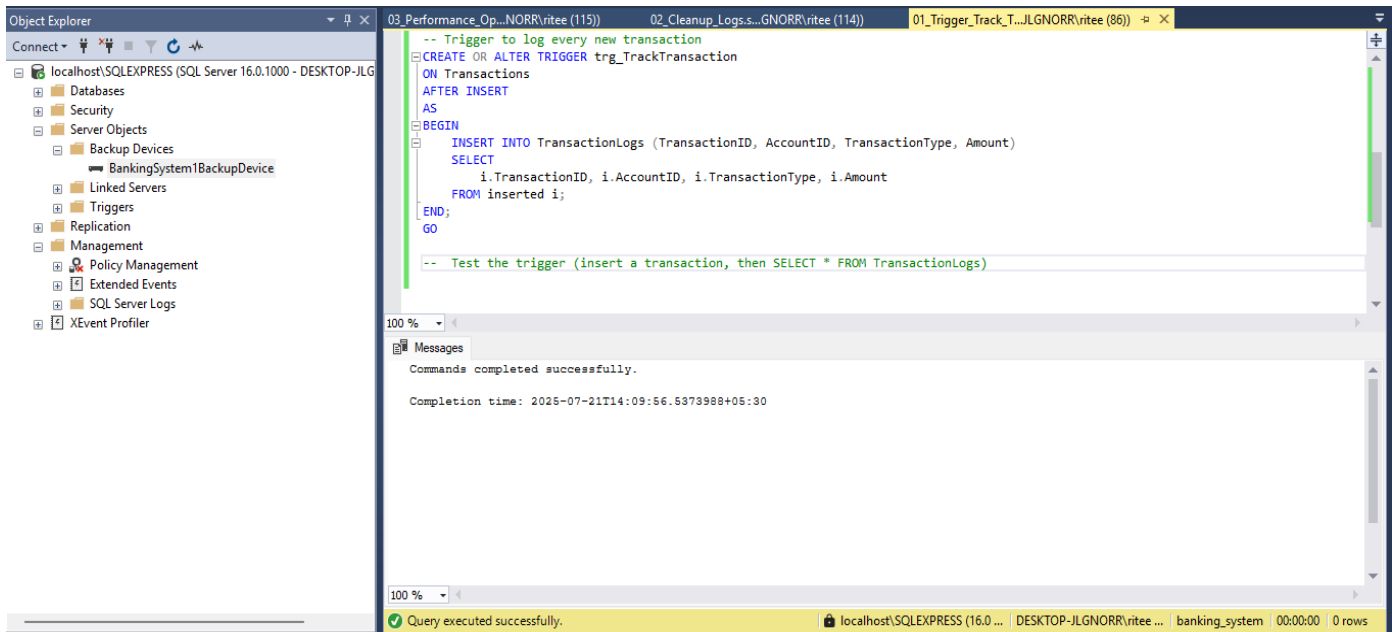
-- Trigger to log every new transaction
CREATE OR ALTER TRIGGER trg_TrackTransaction
ON Transactions
AFTER INSERT
AS
BEGIN
    INSERT INTO TransactionLogs (TransactionID, AccountID, TransactionType, Amount)
    SELECT
```

```

i.TransactionID, i.AccountID, i.TransactionType, i.Amount
FROM inserted i;
END;
GO

-- Test the trigger (insert a transaction, then SELECT * FROM TransactionLogs)

```



02_Trigger_Alert_On_Overdraft.sql

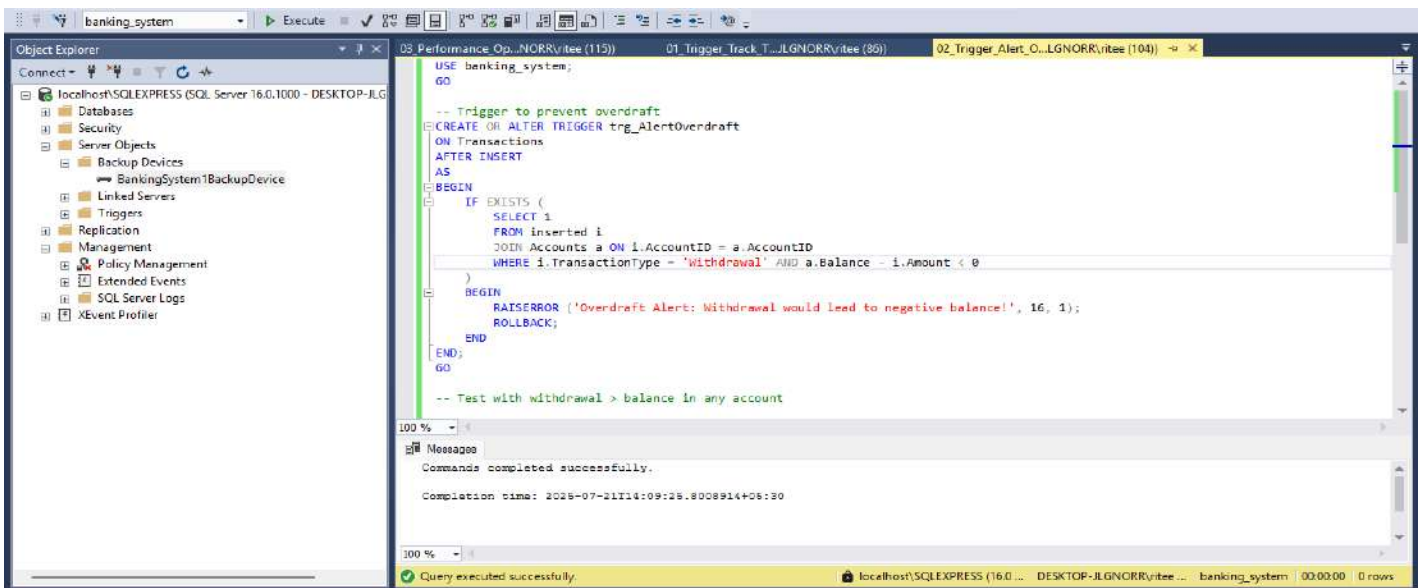
```

USE banking_system;
GO

-- Trigger to prevent overdraft
CREATE OR ALTER TRIGGER trg_AlertOverdraft
ON Transactions
AFTER INSERT
AS
BEGIN
    IF EXISTS (
        SELECT 1
        FROM inserted i
        JOIN Accounts a ON i.AccountID = a.AccountID
        WHERE i.TransactionType = 'Withdrawal' AND a.Balance - i.Amount < 0
    )
    BEGIN
        RAISERROR ('Overdraft Alert: Withdrawal would lead to negative balance!', 16, 1);
        ROLLBACK;
    END
END;
GO

-- Test with withdrawal > balance in any account

```



03_Role-Based_Access_Security.sql

USE banking_system;

GO

-- Create roles

CREATE ROLE Teller;

CREATE ROLE BranchManager;

-- Grant basic Teller permissions

GRANT SELECT, INSERT ON Customers TO Teller;

GRANT SELECT, INSERT ON Accounts TO Teller;

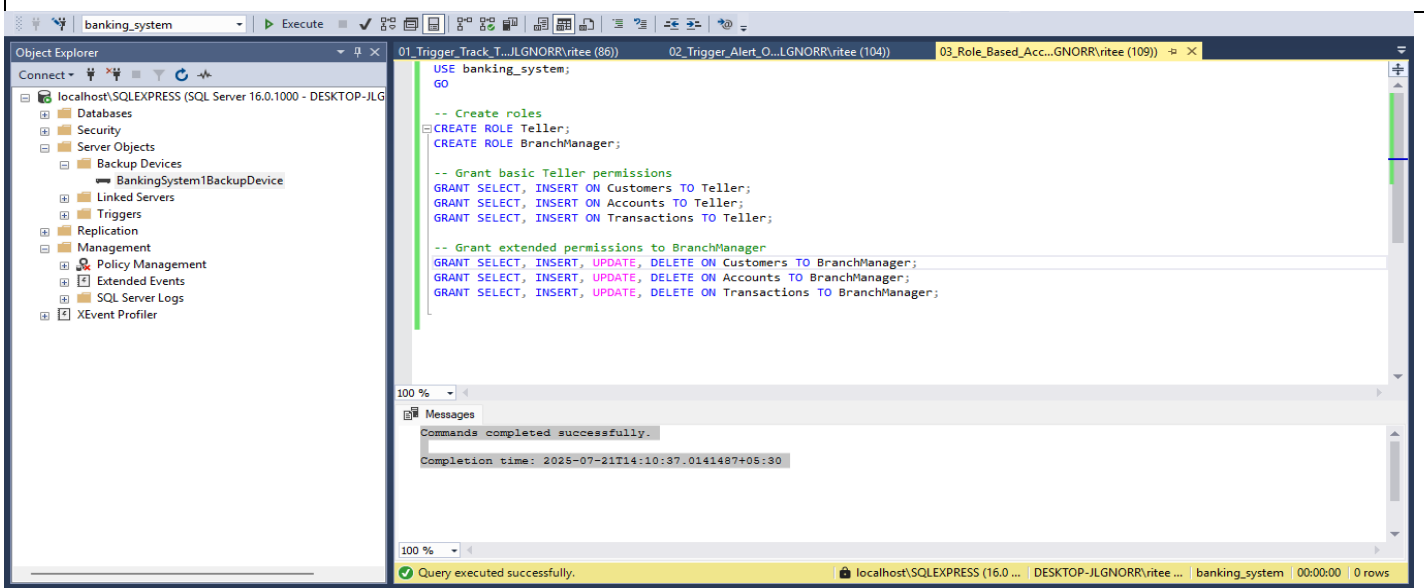
GRANT SELECT, INSERT ON Transactions TO Teller;

-- Grant extended permissions to BranchManager

GRANT SELECT, INSERT, UPDATE, DELETE ON Customers TO BranchManager;

GRANT SELECT, INSERT, UPDATE, DELETE ON Accounts TO BranchManager;

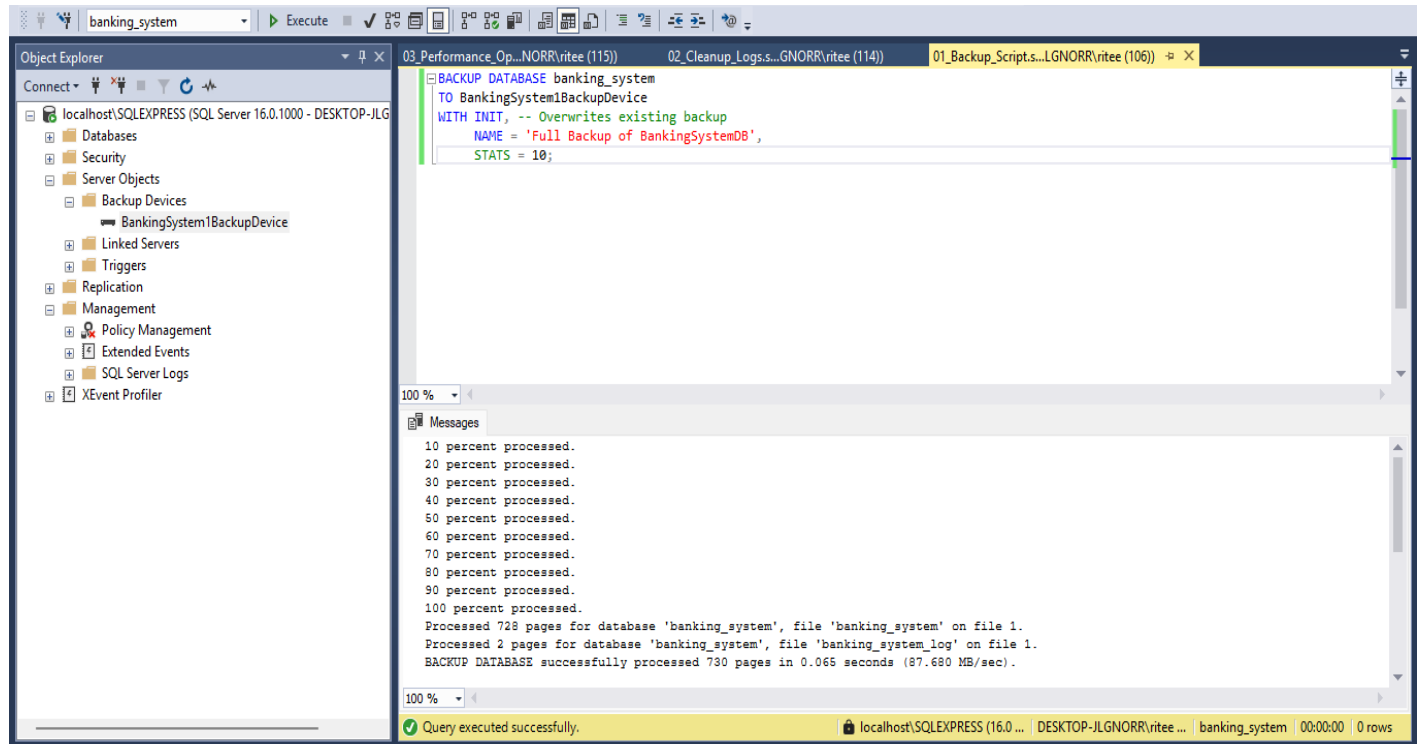
GRANT SELECT, INSERT, UPDATE, DELETE ON Transactions TO BranchManager;



9. Backup and Maintenance

To ensure data durability, availability, and recoverability, regular backup and maintenance of the banking system database is essential. These tasks are critical for protecting sensitive financial and customer data against accidental loss, hardware failures, and security breaches.

01_Backup_Script.sql



The screenshot shows the SQL Server Enterprise Manager interface. The left pane displays the Object Explorer with the 'banking_system' database selected. The right pane shows the execution of the '01_Backup_Script.sql' script. The script content is as follows:

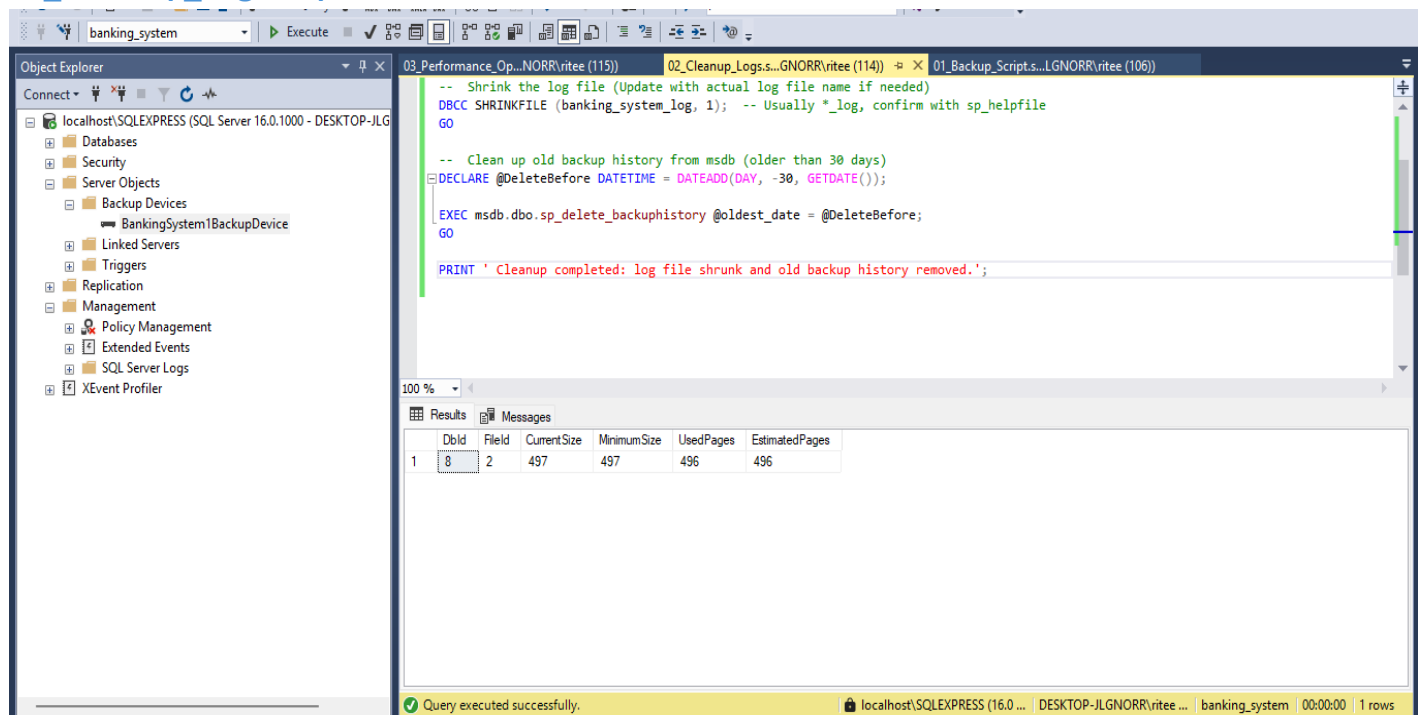
```
BACKUP DATABASE banking_system
TO BankingSystem1BackupDevice
WITH INIT, -- Overwrites existing backup
NAME = 'Full Backup of BankingSystemDB',
STATS = 10;
```

The Messages pane shows the progress of the backup operation:

```
10 percent processed.
20 percent processed.
30 percent processed.
40 percent processed.
50 percent processed.
60 percent processed.
70 percent processed.
80 percent processed.
90 percent processed.
100 percent processed.
Processed 728 pages for database 'banking_system', file 'banking_system' on file 1.
Processed 2 pages for database 'banking_system', file 'banking_system_log' on file 1.
BACKUP DATABASE successfully processed 730 pages in 0.065 seconds (87.680 MB/sec).
```

The status bar at the bottom indicates 'Query executed successfully.' and 'localhost\SQLEXPRESS (16.0.1000 - DESKTOP-JLGNORR\ritee ... banking_system 00:00:00 0 rows'.

02_Cleanup_Logs.sql



The screenshot shows the SQL Server Enterprise Manager interface. The left pane displays the Object Explorer with the 'banking_system' database selected. The right pane shows the execution of the '02_Cleanup_Logs.sql' script. The script content is as follows:

```
-- Shrink the log file (Update with actual log file name if needed)
DBCC SHRINKFILE (banking_system_log, 1); -- Usually *_log, confirm with sp_helpfile
GO

-- Clean up old backup history from msdb (older than 30 days)
DECLARE @DeleteBefore DATETIME = DATEADD(DAY, -30, GETDATE());
EXEC msdb.dbo.sp_delete_backuphistory @oldest_date = @DeleteBefore;
GO

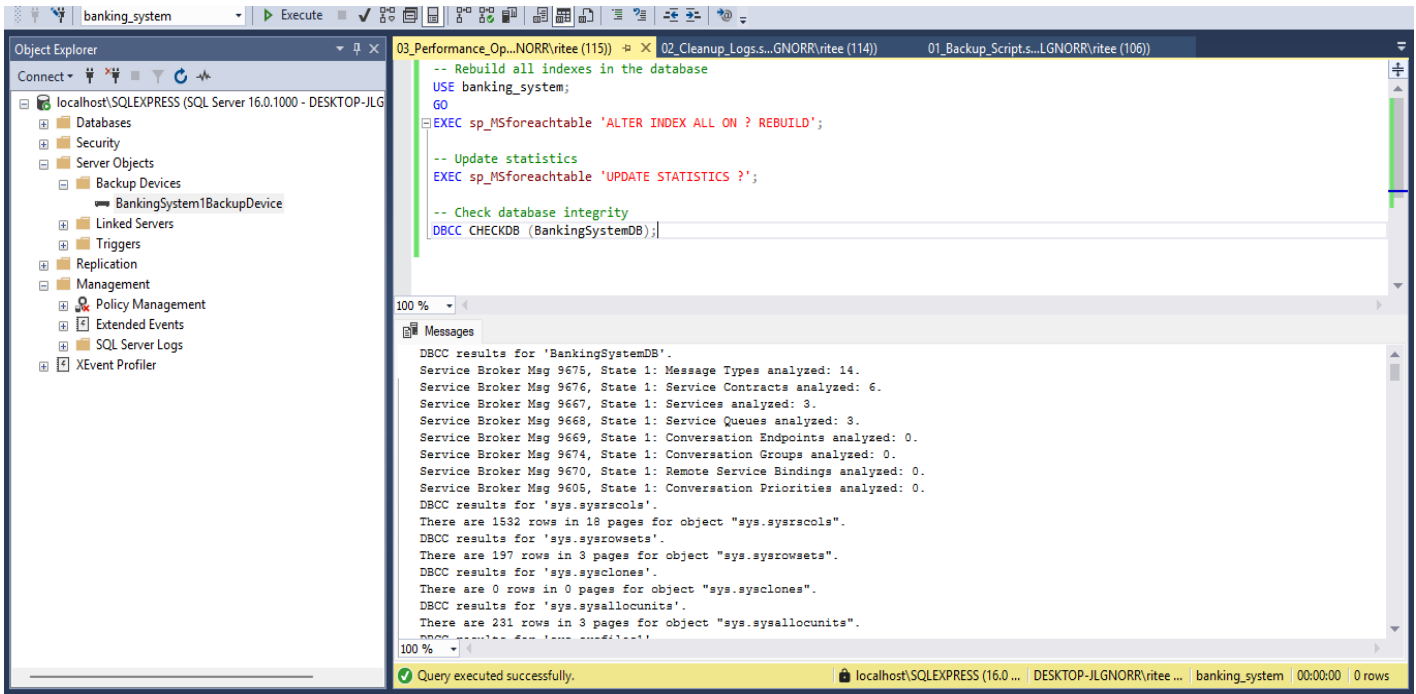
PRINT 'Cleanup completed: log file shrunk and old backup history removed.';
```

The Results pane shows the output of the script:

	Dbid	Field	CurrentSize	MinimumSize	UsedPages	EstimatedPages
1	8	2	497	497	496	496

The status bar at the bottom indicates 'Query executed successfully.' and 'localhost\SQLEXPRESS (16.0.1000 - DESKTOP-JLGNORR\ritee ... banking_system 00:00:00 1 rows'.

03_Performance_Optimization



10. Conclusion

The Banking System Database was successfully designed and implemented using SQL Server to manage core banking operations involving customers, accounts, transactions, loans, and branches. Key components such as stored procedures, views, triggers, and role-based access were developed to ensure data integrity, automation, and security. The system meets the requirements of a functional and efficient relational banking database.

It provides a solid foundation for real-time banking applications.

The project also demonstrates practical knowledge of SQL and real-world database design principles.

11. Future Enhancements

- Add loan repayment tracking and EMI schedules
- Include cheque management features
- Develop a user interface for real-time interaction
- Implement notifications for key events
- Introduce data archiving and basic analytics
- Consider cloud deployment for scalability

12. Appendix

◇ Tools Used

- **Database Engine:** Microsoft SQL Server
- **Query Tool:** SQL Server Management Studio (SSMS)
- **ER Diagram Tool:** dbdiagram.io / Draw.io
- **Documentation:** Microsoft Word

◇ Assumptions

- Each customer can have multiple accounts and loans.
- Every account is linked to only one branch.
- Transfers are allowed only between valid, active accounts.
- Negative balances and overdrafts are restricted by triggers.

◇ Limitations

- No frontend or UI has been integrated with this version.
- The system currently supports basic banking operations only.
- Loan repayment tracking is not yet implemented.
- SMS/email notifications are not included in this version.