Lab 9: Automated Loan Management System with Email Notification Using Stored Procedures

Introduction:

In this lab, we focus on advanced SQL concepts including stored procedures, automated data processing, and relational database management for a loan management system. Stored procedures are precompiled SQL statements that can be executed multiple times, providing better performance, security, and code reusability. This lab demonstrates the creation of a stored procedure that automatically generates email notifications for customers with overdue loan payments.

The lab covers:

- **CREATE PROCEDURE** – Creating stored procedures for business logic automation
- **Date Functions** – Using GETDATE () for current date operations
- **Window Functions** – Using ROW_NUMBER () for sequential numbering
- **JOIN Operations** – Combining data from multiple related tables
- **Conditional Data Insertion** – Inserting data based on specific criteria
- **Foreign Key Constraints** – Maintaining referential integrity in procedural contexts

Step 1: Creating the Database Tables

First, we establish the foundation by creating four interconnected tables: customerdetails, loantype, loandetails, and emaildetails.

SQL Code:

```sql
-- 1 Drop existing tables if they exist
DROP TABLE IF EXISTS emaildetails;
DROP TABLE IF EXISTS loandetails;
DROP TABLE IF EXISTS loantype;
DROP TABLE IF EXISTS customerdetails;

-- 2 Create customerdetails table
CREATE TABLE customerdetails (
    cid INT NOT NULL PRIMARY KEY,
    customername VARCHAR(100),
    email VARCHAR(100),
    phoneno BIGINT,
    address VARCHAR(100)
);

-- 3 Create loantype table
CREATE TABLE loantype (
    sno INT NOT NULL PRIMARY KEY,
    loantype CHAR(50) UNIQUE -- unique so it can be referenced
);

-- 4 Create loandetails table with EMI tracking
CREATE TABLE loandetails (
    sno INT NOT NULL PRIMARY KEY,
    cid INT, -- link to customer
    loanname VARCHAR(100),
    loantype CHAR(50),
    interest INT,
    due_date DATE,
    paid_date DATE NULL, -- NULL if unpaid
    FOREIGN KEY (loantype) REFERENCES loantype(loantype),
    FOREIGN KEY (cid) REFERENCES customerdetails(cid)
);

-- 5 Create emaildetails table
CREATE TABLE emaildetails (
    sno INT NOT NULL,
    [From] VARCHAR(100),
    [To] VARCHAR(100),
    cc VARCHAR(100),
    bcc VARCHAR(100),
    emailbody VARCHAR(100),
    status VARCHAR(100)
);
```

Step 2: Inserting Sample Data

```
-- Insert sample customers
INSERT INTO customerdetails (cid, customername, email, phoneno, address)
VALUES
(1, 'Jonash Chatuat', 'jonashchataut@gmail.com', 9877894562, 'Nepal'),
(2, 'Mohit Singh', 'mohit@gmial.com', 9178956778, 'China'),
(3, 'Binod Kumar', 'binod@gmail.com', 9459988578, 'USA');

-- Insert loan types
INSERT INTO loantype (sno, loantype)
VALUES
(1, 'Home Loan'),
(2, 'Car Loan'),
(3, 'Personal Loan');

-- Insert loans with EMI info
INSERT INTO loandetails (sno, cid, loanname, loantype, interest, due_date, paid_date)
VALUES
(1, 1, 'Hilux loan', 'Car Loan', 8, '2025-05-21', NULL), -- overdue
(2, 2, 'House building', 'Home Loan', 5, '2025-08-15', '2025-08-10'), -- paid
(3, 3, 'Vacation', 'Personal Loan', 12, '2025-08-05', NULL); -- overdue
```

Step 3: Creating the Stored Procedure
We create a stored procedure named 'xyzNotification' that automatically generates email notifications for overdue loans.

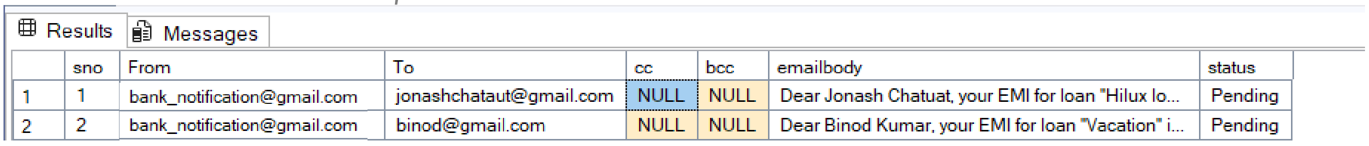
```
CREATE PROCEDURE xyzNotification
AS
BEGIN
    SET NOCOUNT ON;

    DECLARE @today DATE = GETDATE();

    INSERT INTO emaildetails (sno, [From], [To], cc, bcc, emailbody, status)
    SELECT
        ISNULL((SELECT MAX(sno) FROM emaildetails), 0)
        + ROW_NUMBER() OVER (ORDER BY c.cid) AS sno,
        'bank_notification@gmail.com' AS [From],
        c.email AS [To],
        NULL AS cc,
        NULL AS bcc,
        CONCAT('Dear ', c.customername, ', your EMI for loan "', l.loanname, '" is overdue.
Please pay immediately.') AS emailbody,
        'Pending' AS status
    FROM loandetails l
    JOIN customerdetails c ON l.cid = c.cid
    WHERE l.paid_date IS NULL
      AND l.due_date < @today;
END;
GO
```

Step 4: Executing the Stored Procedure

```
EXEC xyzNotification;
SELECT * FROM emaildetails;
```



Results | Messages

	sno	From	To	cc	bcc	emailbody	status
1	1	bank_notification@gmail.com	jonashchataut@gmail.com	NULL	NULL	Dear Jonash Chatuat, your EMI for loan "Hilux lo...	Pending
2	2	bank_notification@gmail.com	binod@gmail.com	NULL	NULL	Dear Binod Kumar, your EMI for loan "Vacation" i...	Pending

Figure: The emaildetails table shows two automated email notifications generated for customers with overdue loans.

Conclusion:

This lab successfully demonstrated the creation and execution of stored procedures that automate business processes while maintaining referential integrity. The procedure 'xyzNotification' effectively:

- **Identifies overdue loans** by comparing due dates with current date
- **Generates personalized emails** using customer and loan information
- **Automates notification process** reducing manual effort and human error
- **Maintains data consistency** through proper foreign key relationships
- **Provides scalable solution** for handling multiple customers and loan types.

Lab 10: Parameterized Stored Procedures with CRUD Operations

Introduction:

In this lab, we create a stored procedure that can perform multiple operations (SELECT and INSERT) on a database table using parameters. Instead of writing separate procedures for each operation, we use one procedure with a flag parameter to control what action to perform. This approach is commonly used in real-world applications for efficient database management.

The lab covers:

- **Parameterized Procedures** – Creating procedures with input parameters
- **Flag-Based Operations** – Using a flag to select different operations
- **Input Validation** – Checking if required data is provided
- **Error Handling** – Using TRY-CATCH to handle errors gracefully

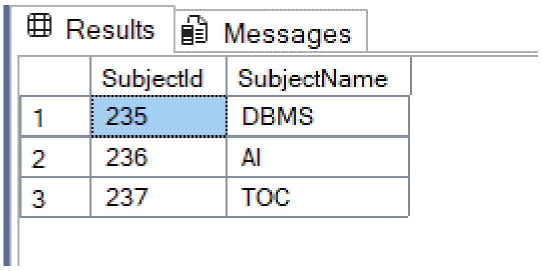
Step 1: Creating the SubjectDetails Table

```sql
-- Drop existing table if it exists
DROP TABLE IF EXISTS SubjectDetails;

-- Create SubjectDetails table
CREATE TABLE SubjectDetails (
    SubjectId INT NOT NULL PRIMARY KEY,
    SubjectName VARCHAR(100)
);

-- Insert sample data
INSERT INTO SubjectDetails (SubjectId, SubjectName) VALUES
(235, 'DBMS'),
(236, 'AI'),
(237, 'TOC');

-- Verify the data
SELECT * FROM SubjectDetails;
```



	SubjectId	SubjectName
1	235	DBMS
2	236	AI
3	237	TOC

Figure 1: The SubjectDetails table for given subs

Step 2: Creating the Stored Procedure

We create a procedure called 'sp_SubjectDetails' that can do two things:

1. **SELECT**: Retrieve data from the table (when @flag = 's')
2. **INSERT**: Add new data to the table (when @flag = 'i')

```sql
DROP PROCEDURE IF EXISTS sp_SubjectDetails;
GO
CREATE PROCEDURE sp_SubjectDetails
(
        @flag Char,
        @SubjectId Int = NULL,
        @SubjectName Varchar(100) = NULL
)
AS
BEGIN
	If @flag = 's' -- SELECT Data From Table
	BEGIN
		IF @SubjectId IS NULL
		BEGIN
			Select '0' As STATUS_CODE,
				'Data Retrived From Table' AS STATUS_MSG

			Select SubjectId,SubjectName
			From SubjectDetails
			RETURN
		END

		Select SubjectId,SubjectName
		From SubjectDetails
		WHERE SubjectId = @SubjectId
	END


	If @flag = 'i' -- INSERT DATA INTO TABLE
	BEGIN
		IF @SubjectId IS NULL
		BEGIN
			Select '100' As STATUS_CODE,
			'Subject ID is Missing' AS STATUS_MSG
			RETURN
		END

		IF @SubjectName IS NULL OR @SubjectName = ''
		BEGIN
			Select '101' As STATUS_CODE,
			'Subject Name is Missing' AS STATUS_MSG
			RETURN
		END

		BEGIN TRY
			Insert Into SubjectDetails(SubjectId, SubjectName)
			Values(@SubjectId, @SubjectName)

			Select '0' AS STATUS_CODE,
			'New Subject is Added successfully' AS STATUS_MSG

		END TRY

		BEGIN CATCH
			Select ERROR_NUMBER() AS STATUS_CODE,
			ERROR_MESSAGE() AS STATUS_MSG
		END CATCH
	END
END
```

How It Works:

Parameters:

- @flag: Tells the procedure what to do ('s' = SELECT, 'i' = INSERT)
- @SubjectId: The subject ID number
- @SubjectName: The subject name

For SELECT (@flag = 's'):

- If no SubjectId is given, it shows all subjects
- If SubjectId is given, it shows only that subject

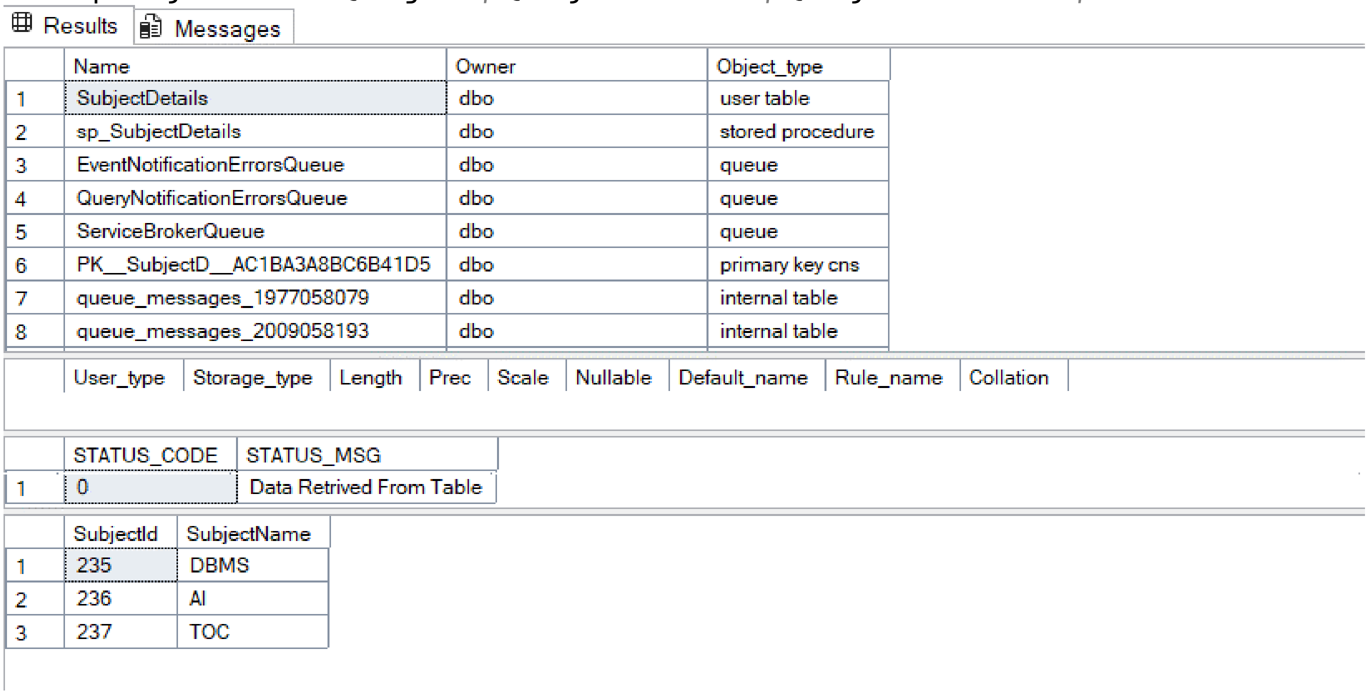
For INSERT (@flag = 'i'):

- First checks if SubjectId is provided (if not, returns error code 100)
- Then checks if SubjectName is provided (if not, returns error code 101)
- If both are provided, it inserts the new subject
- Uses TRY-CATCH to handle any database errors

Step 3: Retrieving All Data (SELECT Operation)

We execute the procedure with flag 's' to get all subjects.

```sql
sp_help
EXEC sp_SubjectDetails @flag='s', @SubjectId = NULL, @SubjectName = NULL;
```



Results | Messages

	Name	Owner	Object_type
1	SubjectDetails	dbo	user table
2	sp_SubjectDetails	dbo	stored procedure
3	EventNotificationErrorsQueue	dbo	queue
4	QueryNotificationErrorsQueue	dbo	queue
5	ServiceBrokerQueue	dbo	queue
6	PK__SubjectD__AC1BA3A8BC6B41D5	dbo	primary key cns
7	queue_messages_1977058079	dbo	internal table
8	queue_messages_2009058193	dbo	internal table

	User_type	Storage_type	Length	Prec	Scale	Nullable	Default_name	Rule_name	Collation

	STATUS_CODE	STATUS_MSG
1	0	Data Retrived From Table

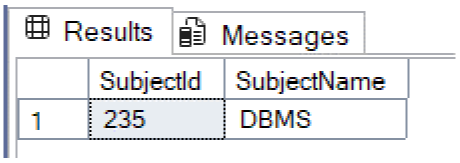
	SubjectId	SubjectName
1	235	DBMS
2	236	AI
3	237	TOC

Figure 2The procedure successfully retrieves all subjects from the table. Status code '0' means the operation was successful.

Step 4: Retrieving Specific Data

We can also get a specific subject by providing its ID.

```sql
EXEC sp_SubjectDetails @flag='s', @SubjectId = 235, @SubjectName = NULL;
```



Results | Messages

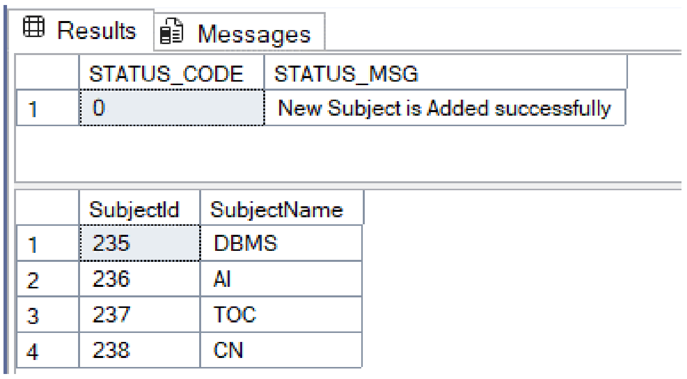
	SubjectId	SubjectName
1	235	DBMS

Step 5: Inserting New Data (INSERT Operation)

Now we use the same procedure to add a new subject to the table.

```sql
EXEC sp_SubjectDetails @flag='i',
    @SubjectId = 238,
    @SubjectName = 'CN';

-- Verify the insertion
SELECT * FROM SubjectDetails;
```



Results | Messages

	STATUS_CODE	STATUS_MSG
1	0	New Subject is Added successfully

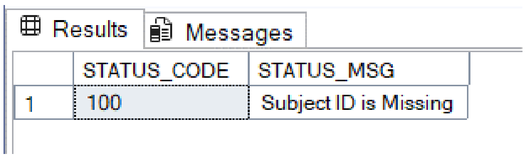
	SubjectId	SubjectName
1	235	DBMS
2	236	AI
3	237	TOC
4	238	CN

Figure 3The new subject CN is successfully added to the table with id 238.

Step 6: Testing Error Handling

Let's see what happens if we try to insert without providing the SubjectId.

```
EXEC sp_SubjectDetails @flag='i',
    @SubjectId = NULL,
    @SubjectName = 'OS';
```



Results | Messages

	STATUS_CODE	STATUS_MSG
1	100	Subject ID is Missing

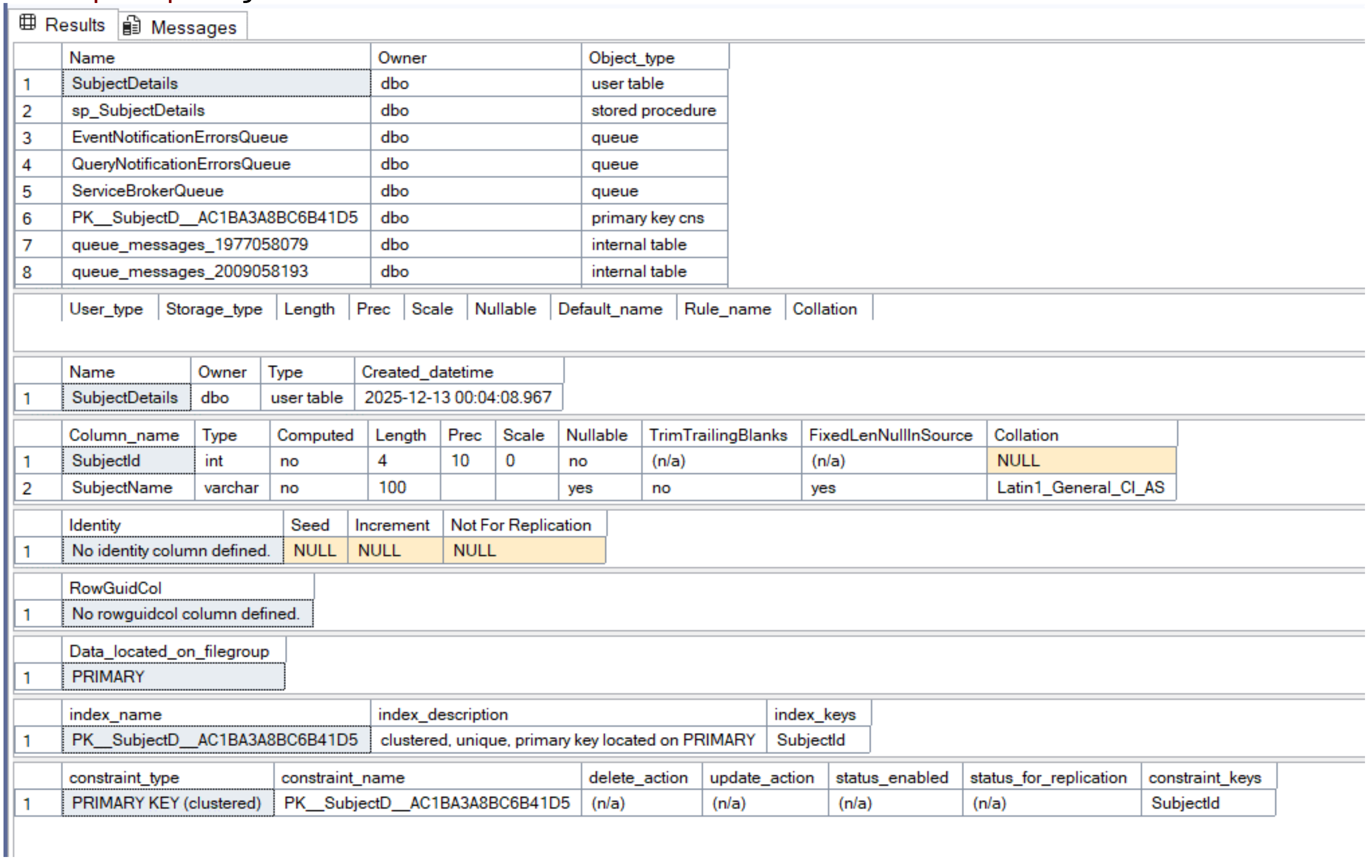
Figure 4 The procedure checks for missing data and returns an error message instead of crashing.

Step 7: Using sp_help Command

We can use the built-in sp_help command to see information about our table.

```
sp_help

exec sp_help SubjectDetails
```



Results | Messages

	Name	Owner	Object_type
1	SubjectDetails	dbo	user table
2	sp_SubjectDetails	dbo	stored procedure
3	EventNotificationErrorsQueue	dbo	queue
4	QueryNotificationErrorsQueue	dbo	queue
5	ServiceBrokerQueue	dbo	queue
6	PK__SubjectD__AC1BA3A8BC6B41D5	dbo	primary key cns
7	queue_messages_1977058079	dbo	internal table
8	queue_messages_2009058193	dbo	internal table

User_type	Storage_type	Length	Prec	Scale	Nullable	Default_name	Rule_name	Collation

	Name	Owner	Type	Created_datetime
1	SubjectDetails	dbo	user table	2025-12-13 00:04:08.967

	Column_name	Type	Computed	Length	Prec	Scale	Nullable	TrimTrailingBlanks	FixedLenNullInSource	Collation
1	SubjectId	int	no	4	10	0	no	(n/a)	(n/a)	NULL
2	SubjectName	varchar	no	100			yes	no	yes	Latin1_General_CI_AS

	Identity	Seed	Increment	Not For Replication
1	No identity column defined.	NULL	NULL	NULL

	RowGuidCol
1	No rowguidcol column defined.

	Data_located_on_filegroup
1	PRIMARY

	index_name	index_description	index_keys
1	PK__SubjectD__AC1BA3A8BC6B41D5	clustered, unique, primary key located on PRIMARY	SubjectId

	constraint_type	constraint_name	delete_action	update_action	status_enabled	status_for_replication	constraint_keys
1	PRIMARY KEY (clustered)	PK__SubjectD__AC1BA3A8BC6B41D5	(n/a)	(n/a)	(n/a)	(n/a)	SubjectId

Figure 5 The sp help command shows detailed information about the SubjectDetails table including column names, data types, and constraints.

Conclusion:

This lab successfully demonstrated how to create a parameterized stored procedure that handles multiple operations. The procedure 'sp_SubjectDetails' effectively:

- Performs SELECT operations to retrieve all or specific subjects
- Performs INSERT operations to add new subjects to the table
- Validates input data before processing to ensure data quality
- Handles errors gracefully using TRY-CATCH blocks
- Returns status codes to indicate success or failure

This approach is practical and efficient for real-world applications where we need to perform multiple database operations through a single, reusable procedure.

Lab 11: Transaction Management in Banking System Using BEGIN TRANSACTION, COMMIT, and ROLLBACK

Introduction:

In this lab, we implement a banking transaction system that demonstrates the critical concept of database transactions. A transaction is a set of SQL operations that must be executed as a single unit - either all operations succeed together, or all fail together. This is essential in banking systems where money transfer between accounts must be atomic (all-or-nothing) to maintain data consistency and prevent financial errors.

The lab covers:

- **Transaction Control** – Using BEGIN TRANSACTION, COMMIT, and ROLLBACK
- **ACID Properties** – Ensuring Atomicity, Consistency, Isolation, and Durability
- **Balance Validation** – Preventing negative account balances
- **Error Handling** – Rolling back transactions when conditions fail
- **Stored Procedures** – Encapsulating transaction logic for reusability

Scenario: Transfer money from Account A (source) to Account B (destination). If Account A has insufficient balance, the entire transaction should be cancelled.

Step 1: Creating the Database and Table

First, we create a new database and the Accounts table to store customer account information.

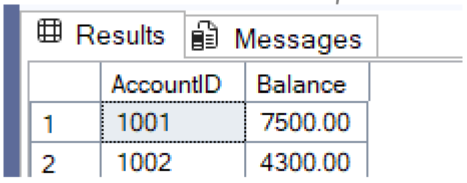
```sql
-- Create a new database
CREATE DATABASE txns;
GO

-- Use the database
USE txns;
GO

-- Create the Accounts table
CREATE TABLE Accounts (
    AccountID INT PRIMARY KEY,
    Balance DECIMAL(10, 2) NOT NULL
);

-- Insert sample data
INSERT INTO Accounts (AccountID, Balance)
VALUES (1001, 7500.00), (1002, 4300.00);

-- Verify the data
SELECT * FROM Accounts;
```



	AccountID	Balance
1	1001	7500.00
2	1002	4300.00

Figure 6 The Accounts table is created with two accounts.

Step 2: Creating the Transaction Stored Procedure

We create a stored procedure that handles money transfer between two accounts using transaction control.

```
CREATE PROCEDURE TransferAmount
    @fromAccount INT,
    @toAccount INT,
    @transferAmount DECIMAL(10, 2)
AS
BEGIN
    DECLARE @balanceA DECIMAL(10, 2);

    -- Start the transaction
    BEGIN TRANSACTION;

    BEGIN TRY
        -- Check the balance of the source account
        SELECT @balanceA = Balance
        FROM Accounts
        WHERE AccountID = @fromAccount;

        -- Perform the transaction if balance is sufficient
        IF @balanceA >= @transferAmount
        BEGIN
            -- Debit from source account
            UPDATE Accounts
            SET Balance = Balance - @transferAmount
            WHERE AccountID = @fromAccount;

            -- Credit to target account
            UPDATE Accounts
            SET Balance = Balance + @transferAmount
            WHERE AccountID = @toAccount;

            -- Commit the transaction
            COMMIT TRANSACTION;
            SELECT 'Transaction completed successfully.' AS Message;
        END
        ELSE
        BEGIN
            -- Rollback the transaction if insufficient balance
            ROLLBACK TRANSACTION;
            SELECT 'Transaction failed: Insufficient balance in the source account.' AS
Message;
        END
    END TRY
    BEGIN CATCH
        -- Rollback in case of any error
        IF @@TRANCOUNT > 0
            ROLLBACK TRANSACTION;
        SELECT ERROR_MESSAGE() AS Message;
    END CATCH
END;
GO
```

How the Procedure Works:

1. Parameters:

- @fromAccount: Source account ID (where money is debited)
- @toAccount: Destination account ID (where money is credited)
- @transferAmount: Amount to transfer

Note: SQL Server uses @ prefix for parameters, unlike MySQL which uses IN keyword.

2. Transaction Steps:

- **BEGIN TRANSACTION**: Begins a new transaction
- **SELECT @balanceA = Balance**: Checks the current balance of source account

- **IF @balanceA >= @transferAmount**: Validates if transfer is possible
- **UPDATE (Debit)**: Subtracts amount from source account
- **UPDATE (Credit)**: Adds amount to destination account
- **COMMIT TRANSACTION**: Makes all changes permanent if successful
- **ROLLBACK TRANSACTION**: Cancels all changes if validation fails
- **TRY-CATCH**: Handles any unexpected errors during the transaction

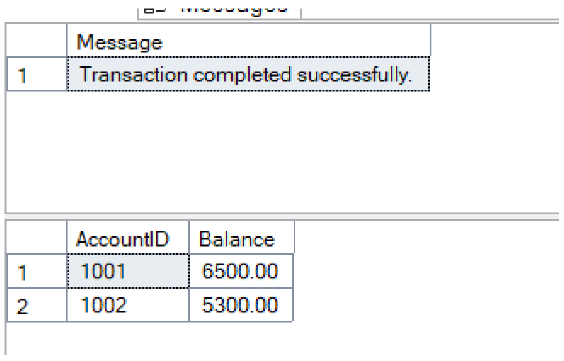
Step 3: Testing Successful Transaction

We test the procedure by transferring 200.00 from Account 1 to Account 2.

```sql
-- Transfer 200.00 from Account 1 to Account 2
EXEC TransferAmount @fromAccount = 1001, @toAccount = 1002, @transferAmount = 1000.00;

-- Check the updated balances

SELECT * FROM Accounts;
```



	Message
1	Transaction completed successfully.

	AccountID	Balance
1	1001	6500.00
2	1002	5300.00

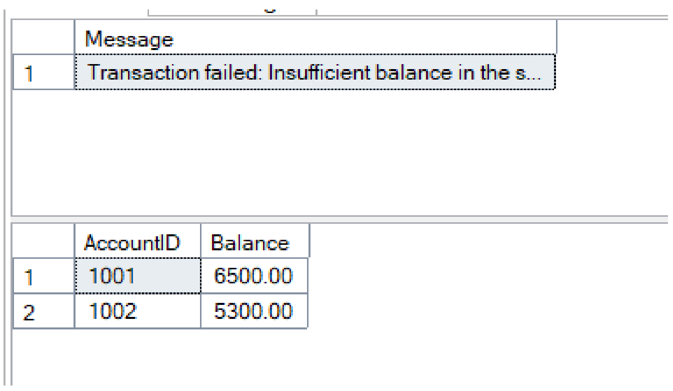
Figure 7 The transaction is successful. Account 1001 balance decreased from 7500.00 to 6500.00 (debited 1000.00), and Account 1002 balance increased from 4300.00 to 5300.00 (credited 1000.00).

Step 4: Testing Failed Transaction (Insufficient Balance)

Now we test what happens when trying to transfer more money than available in the source account.

```sql
-- Attempt to transfer 75000.00 from Account 1001 (which only has 6500.00)
EXEC TransferAmount @fromAccount = 1001, @toAccount = 1002, @transferAmount = 7500.00;

-- Check the balances (should remain unchanged)
SELECT * FROM Accounts;
```



	Message
1	Transaction failed: Insufficient balance in the s...

	AccountID	Balance
1	1001	6500.00
2	1002	5300.00

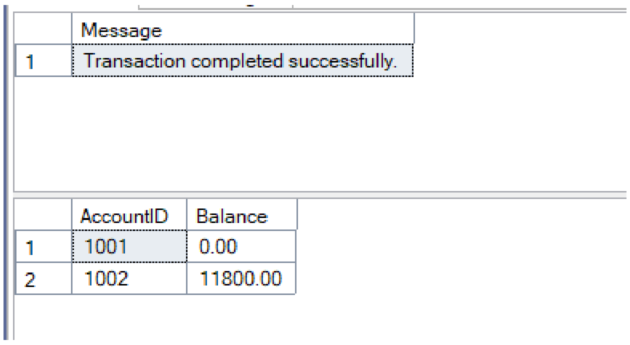
Figure 8 The transaction fails. The ROLLBACK command cancels any changes, so both account balances remain unchanged. This demonstrates transaction atomicity

Step 5: Testing Edge Case (Exact Balance Transfer)

Let's test transferring the exact balance available in an account.

```sql
-- Transfer exactly 6500.00 from Account 1 (empties the account)
EXEC TransferAmount @fromAccount = 1001, @toAccount = 1002, @transferAmount = 6500.00;

-- Check the updated balances
SELECT * FROM Accounts;
```



	Message
1	Transaction completed successfully.

	AccountID	Balance
1	1001	0.00
2	1002	11800.00

Figure 9 The transaction successfully transfers all 6500.00 from Account 1001 to Account 1002. Account 1001 now has zero balance, which is acceptable (not negative).

Conclusion:

This lab successfully demonstrated transaction management in a banking system using stored procedures with transaction control commands. The TransferAmount procedure effectively:

- Ensures atomicity by using START TRANSACTION, COMMIT, and ROLLBACK
- Validates business rules by checking sufficient balance before transfer
- Prevents data inconsistency by rolling back failed transactions
- Maintains data integrity by preventing negative account balances
- Provides clear feedback through success/failure messages

The implementation showcases essential database concepts that are critical for any financial or business-critical application. Understanding transaction management is fundamental for building reliable systems that handle sensitive data and operations where partial completion is not acceptable. This approach ensures that the database always remains in a consistent state, even when operations fail.