Stored Procedure:

SQL stored procedures are sets of SQL statements saved and stored in a database. They can be executed on demand to perform data manipulation and validation tasks, reducing the need to write repetitive SQL code for common operations. Stored procedures are helpful in database management by promoting efficiency and reusability. In addition, they support enhanced database security and maintainability. In this article, we will discuss how to create and execute SQL stored procedures, common use cases, and best practices.

Also, it is a prepared SQL code that you can save, so the code can be reused over and over again. So, if you have an SQL query that you write over and over again, save it as a stored procedure, and then just call it to execute it.You can also pass parameters to a stored procedure, so that the stored procedure can act based on the parameter value(s) that is passed.

Why use stored procedures?

- ✅ Reuse SQL logic
- ✅ Faster execution (precompiled)
- ✅ Better security (no direct table access)
- ✅ Easier maintenance

The benefits of SQL stored procedures include the following:

- **Code reusability:** Once a stored procedure is created, it can be called as many times as needed, eliminating redundancy in SQL code.
- **Enhanced performance:** Stored procedures often execute faster because they are precompiled and stored on the database server, reducing network latency and compilation time.
- **Security:** Stored procedures can improve data security and control over sensitive data access by granting users permission to execute a stored procedure without direct access to tables.

Common Uses for Stored Procedures

SQL stored procedures are useful in scenarios where repetitive complex tasks are required. The following are real-world applications of stored procedures in data management and business operations.

Basic syntax (SQL Server)

Create a stored procedure

```
CREATE PROCEDURE SelectAllCustomers
AS
SELECT * FROM Customers
GO;
```

Execute it

```
EXEC GetAllStudents;
```

Stored Procedure with One Parameter:

```
CREATE PROCEDURE SelectAllCustomers @City nvarchar(30)
AS
SELECT * FROM Customers WHERE City = @City
GO;
```

Execute it

```
EXEC SelectAllCustomers @City = 'London';
```

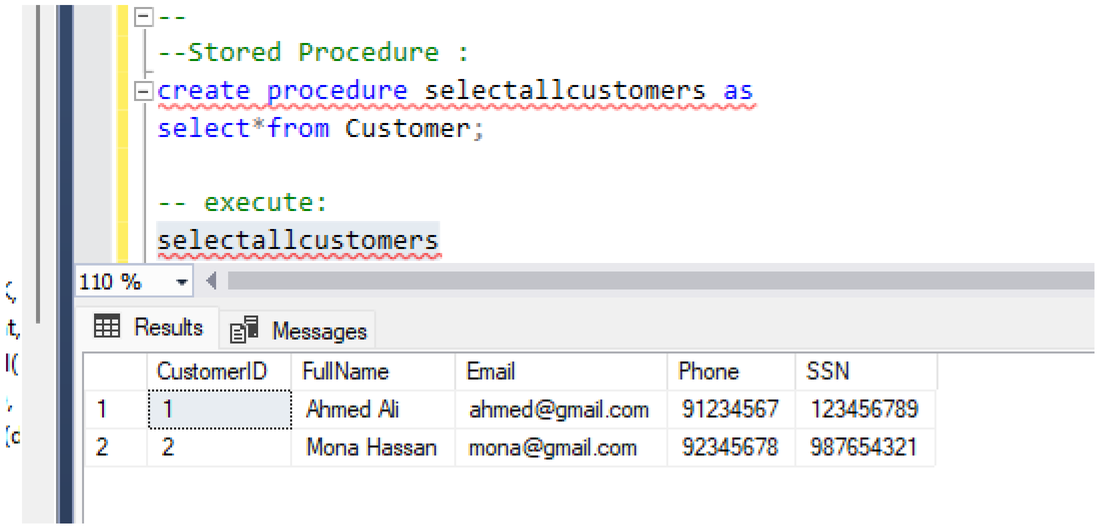
Stored Procedure with Multiple Parameters:

```
CREATE PROCEDURE SelectAllCustomers @City nvarchar(30), @PostalCode
nvarchar(10)
```

```
AS
SELECT * FROM Customers WHERE City = @City AND PostalCode = @PostalCode
GO;
```

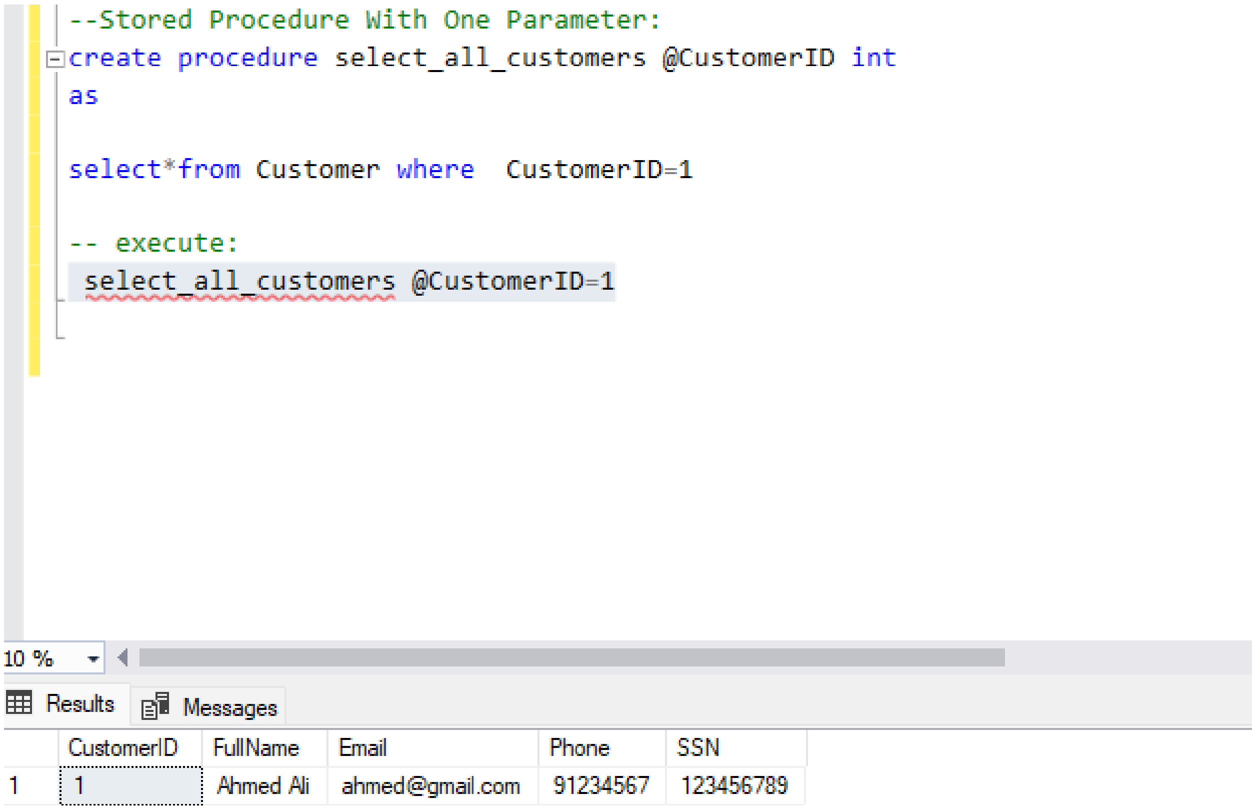
Execute it

```
EXEC SelectAllCustomers @City = 'London', @PostalCode = 'WA1 1DP';
```



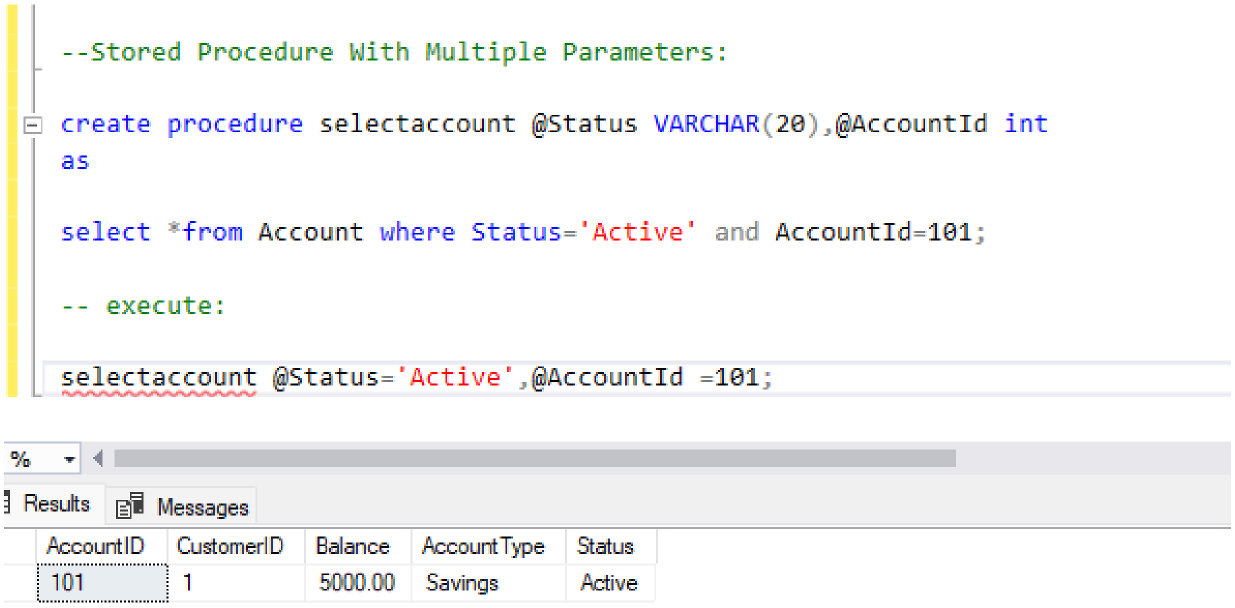
```
--
--Stored Procedure :
create procedure selectallcustomers as
select*from Customer;

-- execute:
selectallcustomers
```

	CustomerID	FullName	Email	Phone	SSN
1	1	Ahmed Ali	ahmed@gmail.com	91234567	123456789
2	2	Mona Hassan	mona@gmail.com	92345678	987654321



```
--Stored Procedure With One Parameter:
create procedure select_all_customers @CustomerID int
as

select*from Customer where  CustomerID=1

-- execute:
 select_all_customers @CustomerID=1
```

	CustomerID	FullName	Email	Phone	SSN
1	1	Ahmed Ali	ahmed@gmail.com	91234567	123456789



```
--Stored Procedure With Multiple Parameters:

create procedure selectaccount @Status VARCHAR(20),@AccountId int
as

select *from Account where Status='Active' and AccountId=101;

-- execute:

selectaccount @Status='Active',@AccountId =101;
```

	AccountID	CustomerID	Balance	AccountType	Status
	101	1	5000.00	Savings	Active

Data migration and ETL processes

Stored procedures are also used to load, transform, and migrate data between systems. A stored procedure can automate data extraction from a source database, transform it as needed, and insert it into a target table, simplifying data integration for reporting or analysis.

```
CREATE PROCEDURE ETLProcess
AS
BEGIN
    -- Extract
    INSERT INTO StagingTable
    SELECT * FROM SourceTable WHERE Condition;
    -- Transform
    UPDATE StagingTable SET ColumnX =
TransformationLogic(ColumnX);
    -- Load
    INSERT INTO TargetTable
    SELECT * FROM StagingTable;
END;
```

Finally, SQL stored procedures enhance code reusability and performance optimization in database management. Stored procedures also enhance database security through controlled access and ensuring data integrity. As a data practitioner, I encourage you to practice creating and executing stored procedures to master the best database management practices.

Transaction:

A Transaction in SQL is a sequence of one or more SQL statements that are executed as a single unit of work. Either all statements succeed, or none of them apply.

Also, transactions in SQL are a sequence of operations performed as a single unit of work. It ensures that either all operations within the transaction are successfully executed or none of them are applied, maintaining the integrity and consistency of the database. Transactions are crucial for scenarios like bank transfers, where partial updates can lead to inconsistencies.

Key Properties of Transactions (ACID)

Atomicity: Ensures that all operations in a transaction are completed successfully, or none are applied. If one operation fails, the entire transaction is rolled back.

Consistency: Guarantees that the database transitions from one valid state to another, preserving data integrity.

Isolation: Ensures that concurrent transactions do not interfere with each other, maintaining accuracy.

Durability: Once a transaction is committed, its changes are permanent, even in the event of a system failure.

Transaction Control Commands

1. BEGIN TRANSACTION

Marks the start of a transaction. All subsequent operations are part of this transaction until a COMMIT or ROLLBACK is issued.

```
BEGIN TRANSACTION;

UPDATE accounts SET balance = balance - 100 WHERE account_id = 1;

UPDATE accounts SET balance = balance + 100 WHERE account_id = 2;
```

2. COMMIT

Saves all changes made during the transaction permanently to the database.

```
COMMIT;
```

3. ROLLBACK

Reverts all changes made during the transaction to the state before BEGIN TRANSACTION.

```
ROLLBACK;
```

4. SAVEPOINT

Creates a checkpoint within a transaction, allowing partial rollbacks to a specific point.

```
SAVEPOINT sp1;

UPDATE accounts
```

SET balance = balance - 50 WHERE account_id = 1;

ROLLBACK TO sp1; -- Reverts changes after SAVEPOINT sp1

5. RELEASE SAVEPOINT

Deletes a previously created savepoint, making it unavailable for rollbacks.

RELEASE SAVEPOINT sp1;

6. SET TRANSACTION

Defines transaction properties like isolation level or read-only status.

SET TRANSACTION ISOLATION LEVEL READ COMMITTED;

Example: Bank Transfer

BEGIN TRANSACTION;

UPDATE accounts

SET balance = balance - 150

WHERE account_id = 'A';

UPDATE accounts

SET balance = balance + 150

WHERE account_id = 'B';

COMMIT;

If any operation fails, a ROLLBACK ensures no partial updates occur.

SQL transactions are essential for maintaining data consistency and reliability in relational databases. By leveraging commands like BEGIN TRANSACTION, COMMIT, and ROLLBACK, developers can ensure robust and error-free database operations.

Why use Transaction?

- Ensures data accuracy
- Maintains database consistency
- Prevents partial updates
- Allows rollback if an error occurs

Basic Syntax (SQL Server)

```
BEGIN TRANSACTION;

UPDATE Accounts

SET Balance = Balance - 500

WHERE AccountID = 1;

UPDATE Accounts

SET Balance = Balance + 500

WHERE AccountID = 2;
```

COMMIT;

✓ If both updates succeed, the changes are saved.

Using ROLLBACK

```
BEGIN TRANSACTION;

UPDATE Accounts
SET Balance = Balance - 500
WHERE AccountID = 1;

IF @@ERROR <> 0
BEGIN
    ROLLBACK;
    RETURN;
END

UPDATE Accounts
SET Balance = Balance + 500
WHERE AccountID = 2;

COMMIT;
```

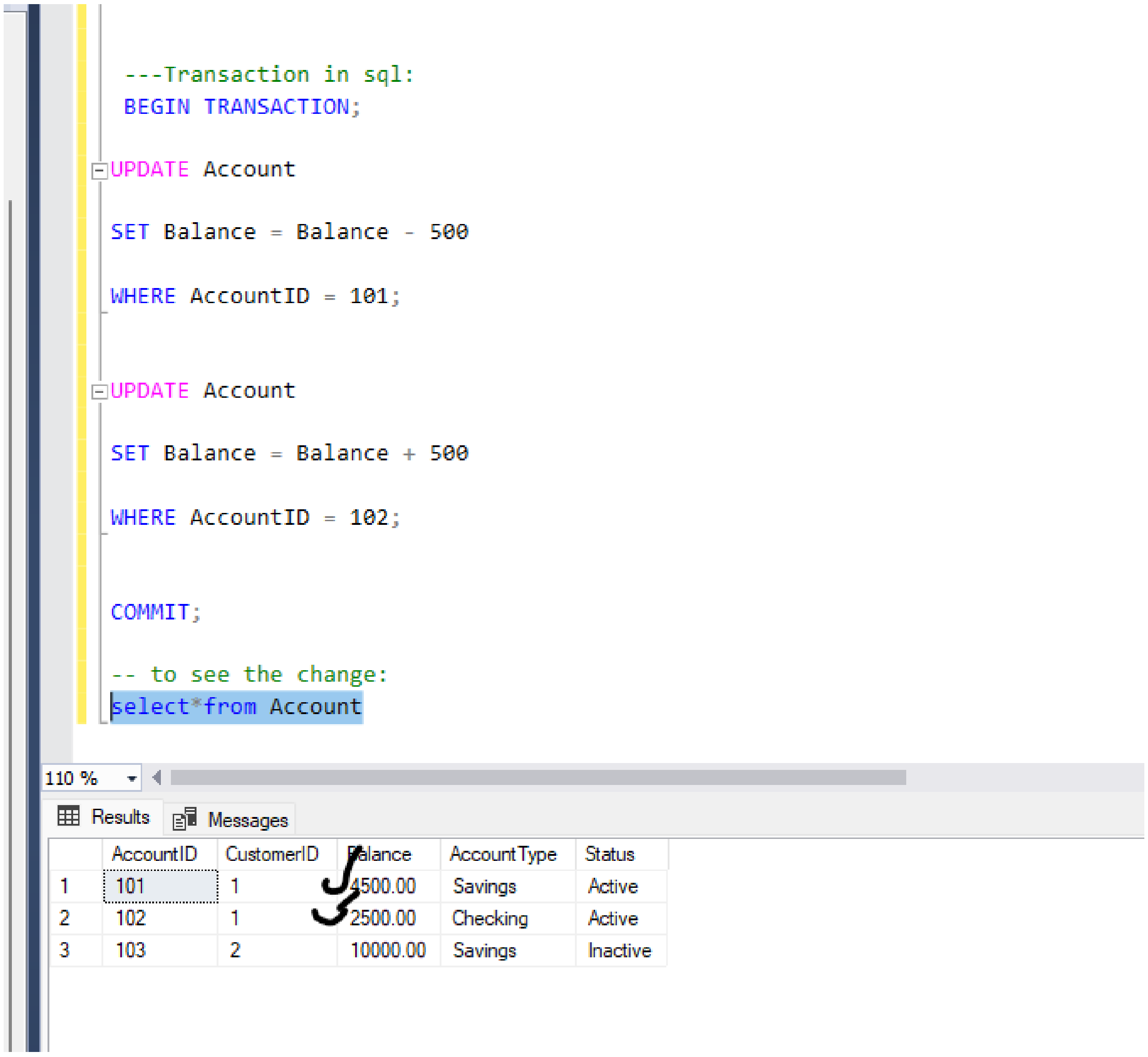
❌ If an error occurs, all changes are undone.

Using TRY...CATCH (Recommended)

```
BEGIN TRY
    BEGIN TRANSACTION;

    UPDATE Accounts
    SET Balance = Balance - 500
    WHERE AccountID = 1;

    UPDATE Accounts
    SET Balance = Balance + 500
    WHERE AccountID = 2;

    COMMIT;
END TRY
BEGIN CATCH
    ROLLBACK;
END CATCH;
```

Important Transaction Commands

Command	Description
BEGIN TRANSACTION	Starts a transaction
COMMIT	Saves changes
ROLLBACK	Cancels changes
SAVEPOINT	Creates a rollback point

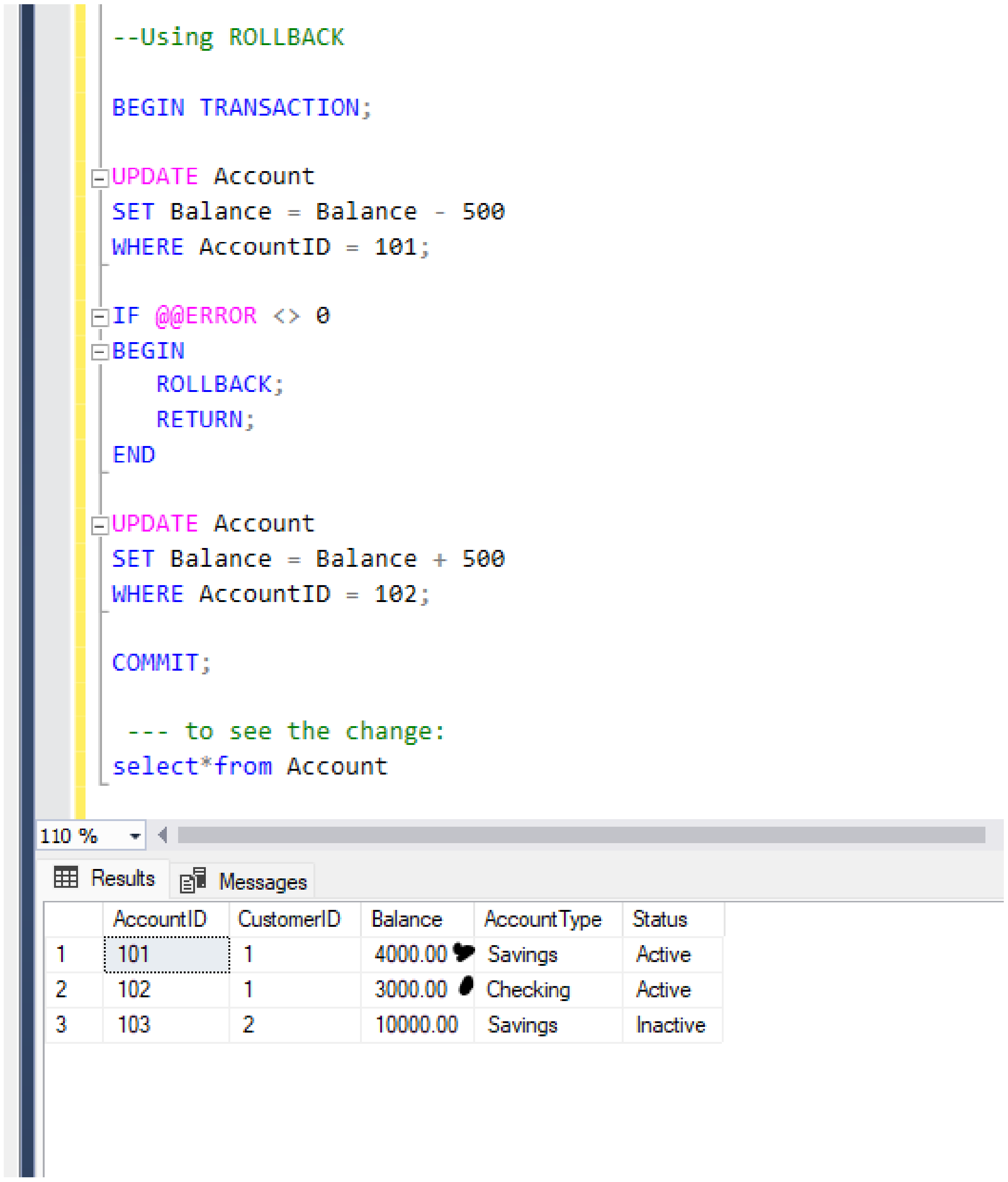


```sql
---Transaction in sql:
BEGIN TRANSACTION;

UPDATE Account

SET Balance = Balance - 500

WHERE AccountID = 101;


UPDATE Account

SET Balance = Balance + 500

WHERE AccountID = 102;


COMMIT;

-- to see the change:
select*from Account
```

110 %

Results | Messages

	AccountID	CustomerID	Balance	AccountType	Status
1	101	1	4500.00	Savings	Active
2	102	1	2500.00	Checking	Active
3	103	2	10000.00	Savings	Inactive

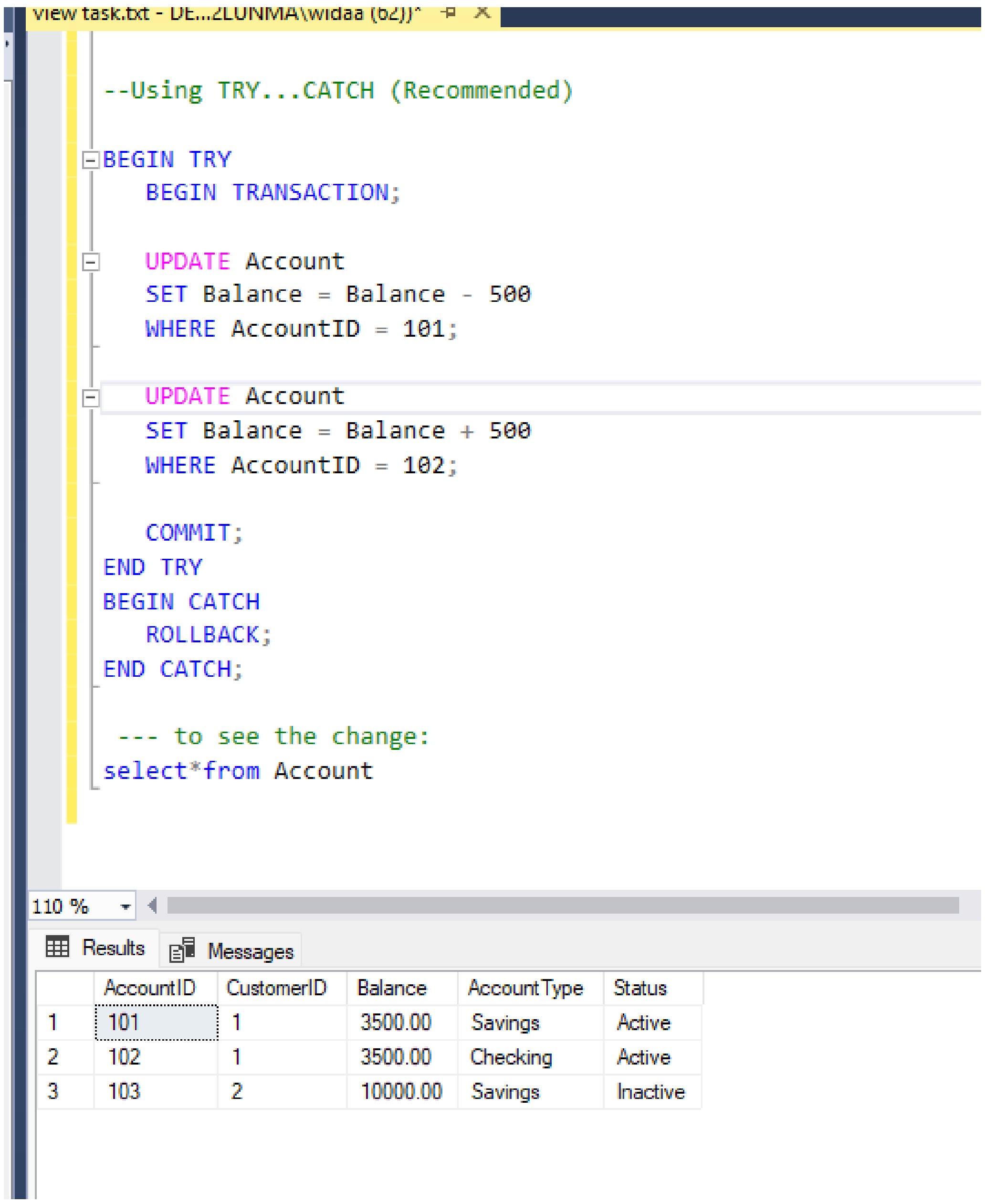


```sql
--Using ROLLBACK

BEGIN TRANSACTION;

UPDATE Account
SET Balance = Balance - 500
WHERE AccountID = 101;

IF @@ERROR <> 0
BEGIN
    ROLLBACK;
    RETURN;
END

UPDATE Account
SET Balance = Balance + 500
WHERE AccountID = 102;

COMMIT;

 --- to see the change:
select*from Account
```

110 %

Results | Messages

	AccountID	CustomerID	Balance	AccountType	Status
1	101	1	4000.00	Savings	Active
2	102	1	3000.00	Checking	Active
3	103	2	10000.00	Savings	Inactive



view task.txt - DE...2LUNMA\widaa (62))*

```
--Using TRY...CATCH (Recommended)

BEGIN TRY
   BEGIN TRANSACTION;

   UPDATE Account
   SET Balance = Balance - 500
   WHERE AccountID = 101;

   UPDATE Account
   SET Balance = Balance + 500
   WHERE AccountID = 102;

   COMMIT;
END TRY
BEGIN CATCH
   ROLLBACK;
END CATCH;

 --- to see the change:
select*from Account
```

110 %

Results | Messages

	AccountID	CustomerID	Balance	AccountType	Status
1	101	1	3500.00	Savings	Active
2	102	1	3500.00	Checking	Active
3	103	2	10000.00	Savings	Inactive

1️⃣ Explicit Transaction

Meaning:
An **Explicit Transaction** is started and controlled **manually by the user**.
You decide **when it starts** and **when it ends**.

Commands used

- `BEGIN TRANSACTION`
- `COMMIT`
- `ROLLBACK`

Example

```
BEGIN TRANSACTION;

INSERT INTO Orders (OrderID, Amount)
VALUES (1, 500);

UPDATE Accounts
SET Balance = Balance - 500
WHERE AccountID = 10;

COMMIT;
```

What happens?

- If **COMMIT** → changes are saved permanently
- If **ROLLBACK** or server crash before COMMIT → all changes are undone

When to use it?

✓ Important operations
✓ Multiple related SQL statements
✓ Financial systems, banking, inventory

2️⃣ Implicit Transaction

Meaning:
An **Implicit Transaction** is started **automatically by SQL Server** after certain SQL statements, **without writing `BEGIN TRANSACTION`**.

Enable implicit transactions

```
SET IMPLICIT_TRANSACTIONS ON;
```

Example

```
SET IMPLICIT_TRANSACTIONS ON;

INSERT INTO Orders (OrderID, Amount)
VALUES (2, 300);

COMMIT;
```

What happens?

- SQL Server **automatically starts** the transaction
- You **must still use COMMIT or ROLLBACK**
- If you forget COMMIT → transaction stays open

Disable it (default behavior)

```
SET IMPLICIT_TRANSACTIONS OFF;
```

🔍 Key Differences

Feature	Explicit	Implicit
Transaction start	Manual	Automatic
BEGIN TRANSACTION needed	Yes	No
COMMIT / ROLLBACK	Required	Required
Control level	Full control	Less control
Common usage	Most systems	Rarely used