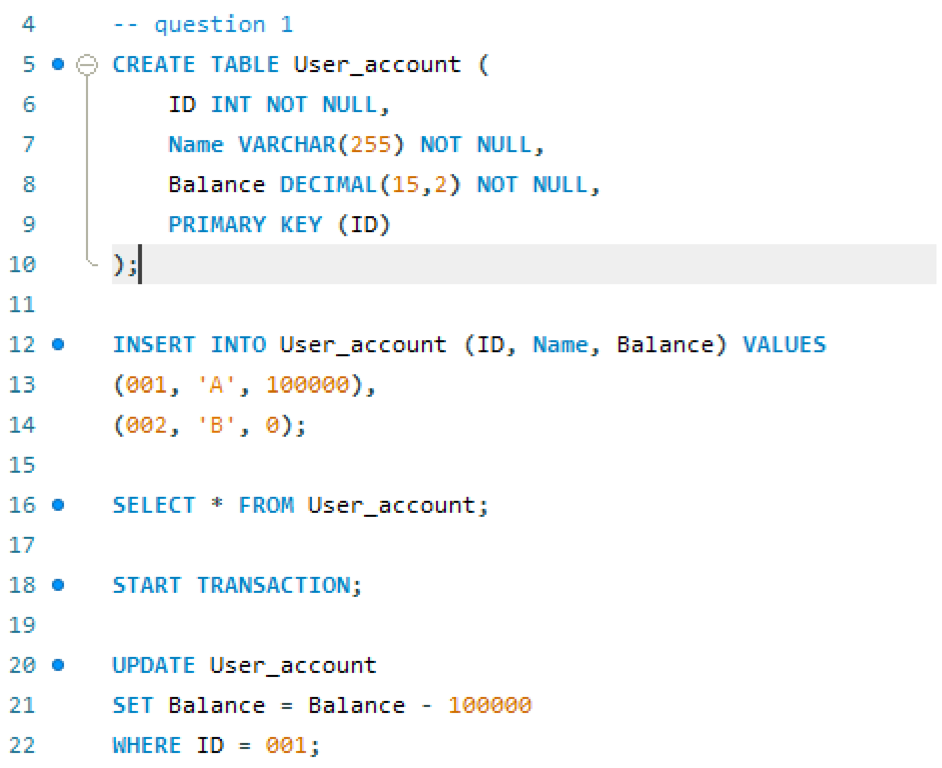
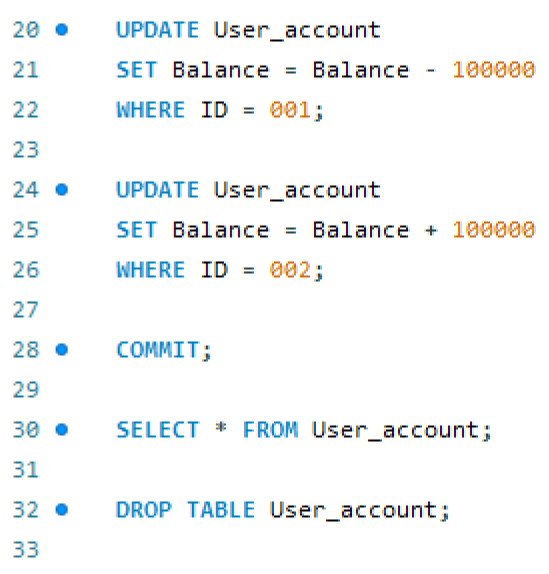
Lab 1 - 2331200153

Question 1

1. **Question**
   i. Create a Database (Assume the database name as per your choice)
   ii. Create a table for a user account with attribute ID, Name, and Balance in a Database.
   iii. Insert the data in the table.
   iv. Create a banking transaction that transfers 100,000VND from one account_id to another account_id. (such as account_id 001, 002)
      a. Deduct amount from Account such as (A).
      b. Add amount to Account such as (B).
   v. Commit if both operations succeed; rollback if any error occurs

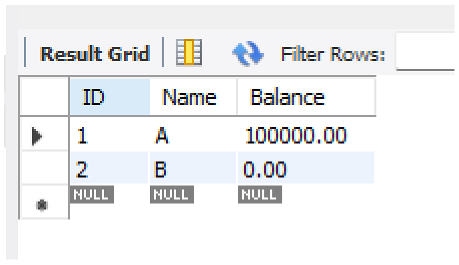


```
-- question 1
CREATE TABLE User_account (
    ID INT NOT NULL,
    Name VARCHAR(255) NOT NULL,
    Balance DECIMAL(15,2) NOT NULL,
    PRIMARY KEY (ID)
);

INSERT INTO User_account (ID, Name, Balance) VALUES
(001, 'A', 100000),
(002, 'B', 0);

SELECT * FROM User_account;

START TRANSACTION;

UPDATE User_account
SET Balance = Balance - 100000
WHERE ID = 001;
```



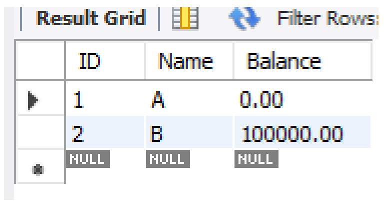
```sql
UPDATE User_account
SET Balance = Balance - 100000
WHERE ID = 001;

UPDATE User_account
SET Balance = Balance + 100000
WHERE ID = 002;

COMMIT;

SELECT * FROM User_account;

DROP TABLE User_account;
```

Before



ID	Name	Balance
1	A	100000.00
2	B	0.00
NULL	NULL	NULL

Result



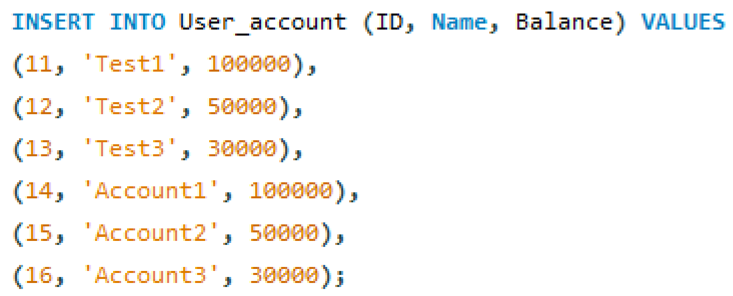
ID	Name	Balance
1	A	0.00
2	B	100000.00
NULL	NULL	NULL

basic money transfer transaction between two accounts

transaction successfully transfers 100000 from Account A to Account B

Question 2

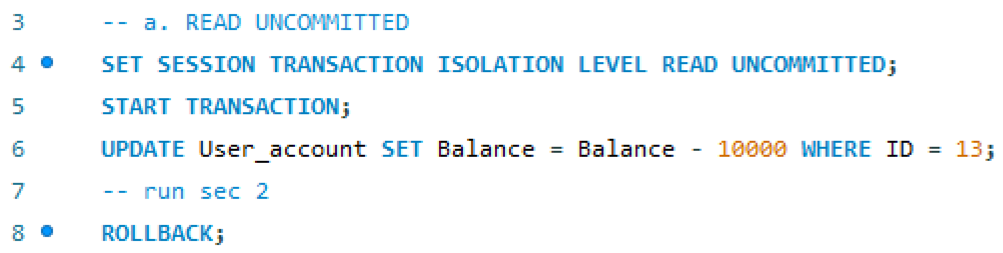
test data:



```sql
INSERT INTO User_account (ID, Name, Balance) VALUES
(11, 'Test1', 100000),
(12, 'Test2', 50000),
(13, 'Test3', 30000),
(14, 'Account1', 100000),
(15, 'Account2', 50000),
(16, 'Account3', 30000);
```

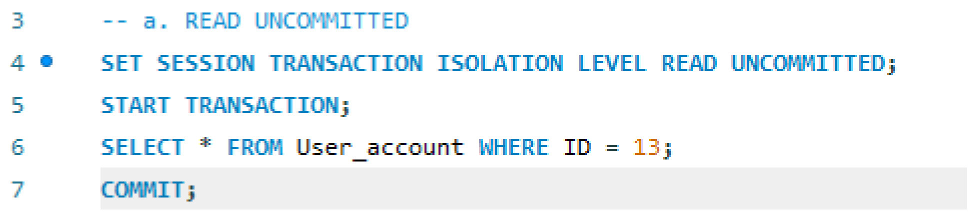
a. READ UNCOMMITTED

Section 1:



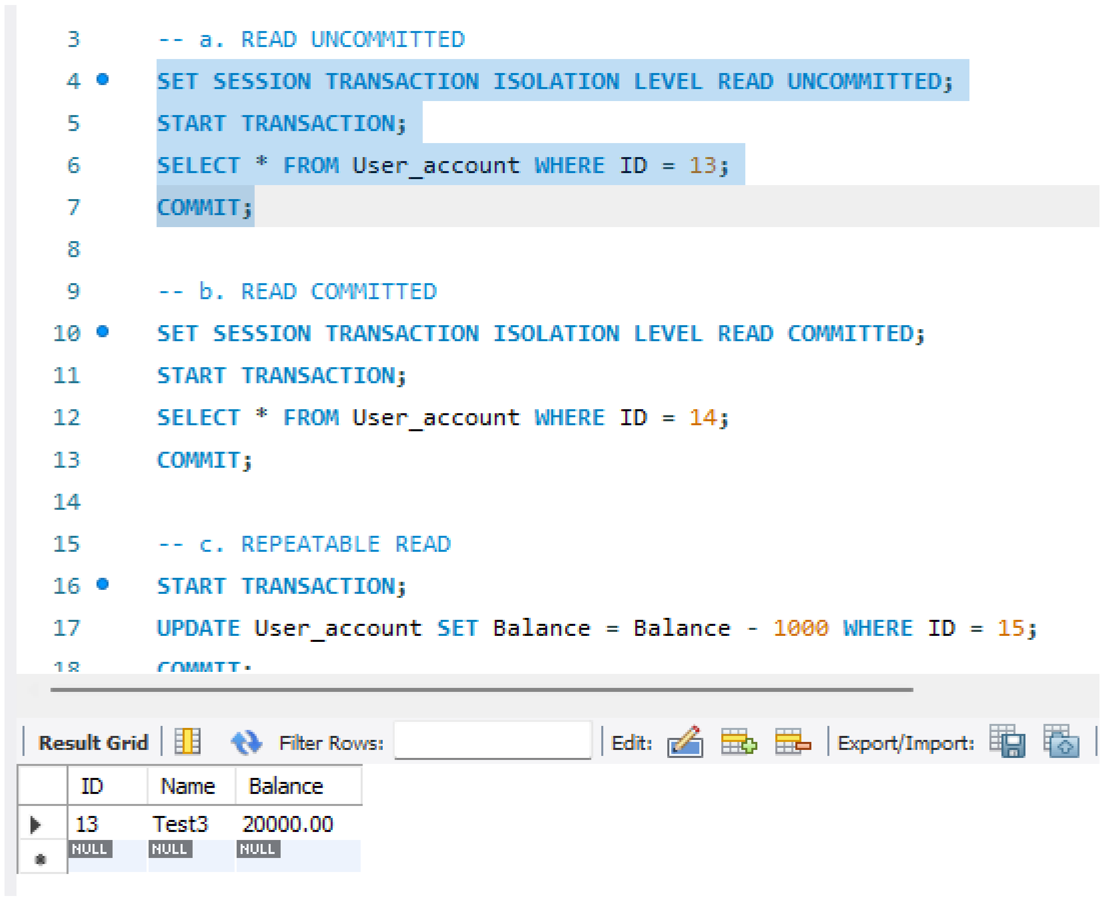
```sql
-- a. READ UNCOMMITTED
SET SESSION TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
START TRANSACTION;
UPDATE User_account SET Balance = Balance - 10000 WHERE ID = 13;
-- run sec 2
ROLLBACK;
```

Section 2:



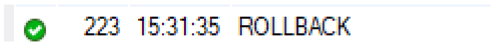
```sql
-- a. READ UNCOMMITTED
SET SESSION TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
START TRANSACTION;
SELECT * FROM User_account WHERE ID = 13;
COMMIT;
```

Result:



```
-- a. READ UNCOMMITTED
SET SESSION TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
START TRANSACTION;
SELECT * FROM User_account WHERE ID = 13;
COMMIT;

-- b. READ COMMITTED
SET SESSION TRANSACTION ISOLATION LEVEL READ COMMITTED;
START TRANSACTION;
SELECT * FROM User_account WHERE ID = 14;
COMMIT;

-- c. REPEATABLE READ
START TRANSACTION;
UPDATE User_account SET Balance = Balance - 1000 WHERE ID = 15;
COMMIT;
```

ID	Name	Balance
13	Test3	20000.00
NULL	NULL	NULL



```
223  15:31:35  ROLLBACK
```

1. Section 1 start a transaction and update Account 13 balance (reduce by 10000)

2. The change is not committed yet

3. Section 2 read Account 13 and see the uncommitted change

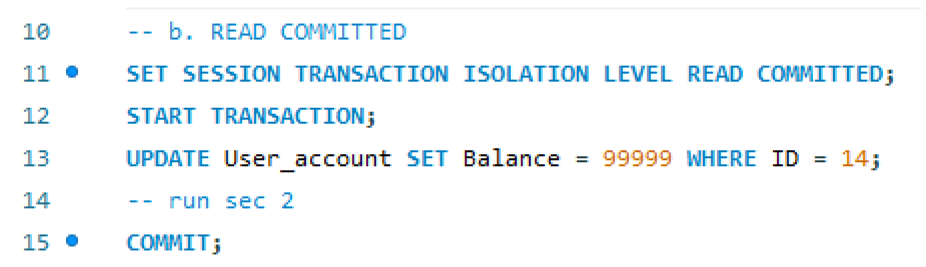
4. Section 1 roll back, meaning the change dont happened

5. Section 2 has read data that dont happened   in the official database

Section 2 sees the updated balance (20000) even though Section 1 hasnt committed. After Section 1 rolls back, the actual balance remains 30000, but Section 2 already read the incorrect value

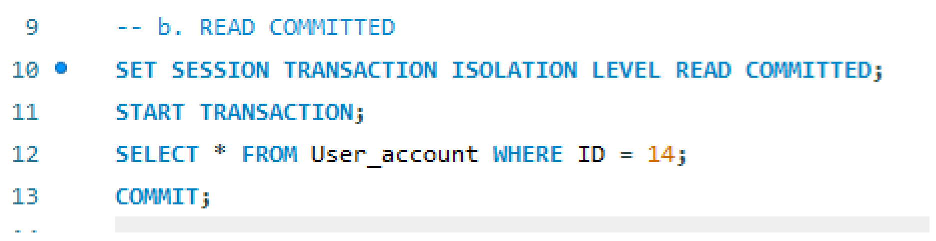
b. READ COMMITTED

Section1:



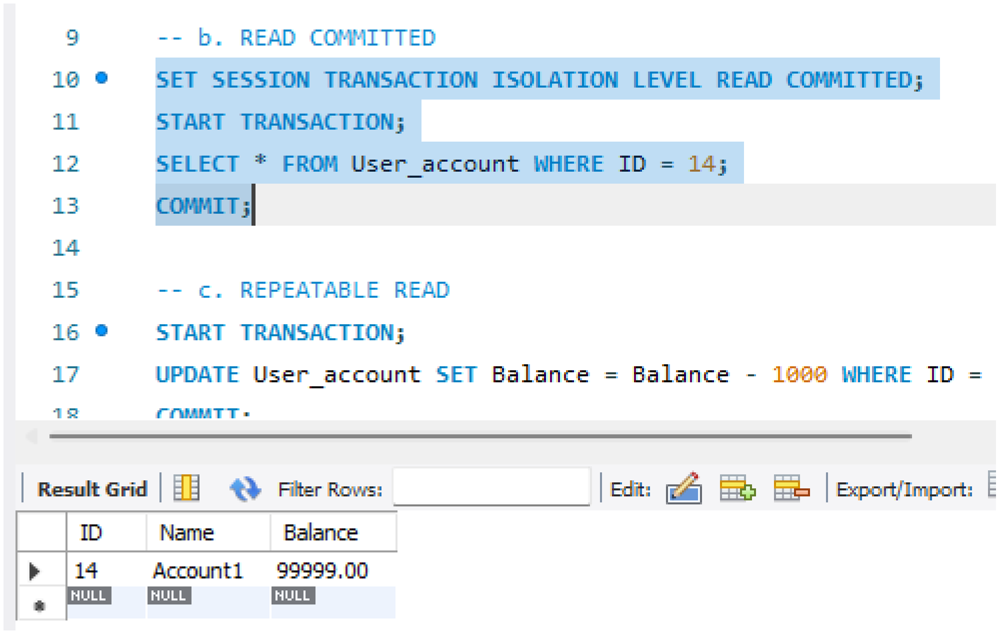
```sql
-- b. READ COMMITTED
SET SESSION TRANSACTION ISOLATION LEVEL READ COMMITTED;
START TRANSACTION;
UPDATE User_account SET Balance = 99999 WHERE ID = 14;
-- run sec 2
COMMIT;
```

Section2:



```sql
-- b. READ COMMITTED
SET SESSION TRANSACTION ISOLATION LEVEL READ COMMITTED;
START TRANSACTION;
SELECT * FROM User_account WHERE ID = 14;
COMMIT;
```

Result:



```sql
-- b. READ COMMITTED
SET SESSION TRANSACTION ISOLATION LEVEL READ COMMITTED;
START TRANSACTION;
SELECT * FROM User_account WHERE ID = 14;
COMMIT;

-- c. REPEATABLE READ
START TRANSACTION;
UPDATE User_account SET Balance = Balance - 1000 WHERE ID =
COMMIT;
```

ID	Name	Balance
14	Account1	99999.00
NULL	NULL	NULL

1. Section 1 update Account 14 to 99999 but dont commit immediately

2. Section 2 tries to read Account 14

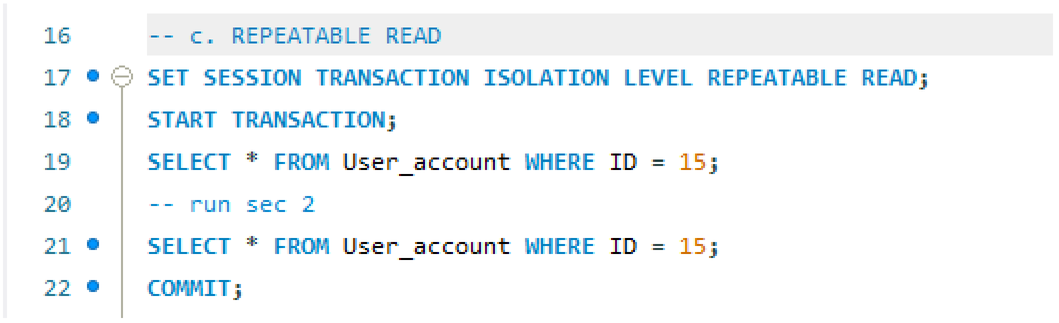
3. Section 2 must wait until Section 1 commits or reads the old committed value (100000)

4. After Section 1 commits, Section 2 sees the new value (99999)

Section 2 either waits for Section 1 to commit or sees the old committed value. Once Section 1 commits, any new reads will see 99999. This ensures data reliability

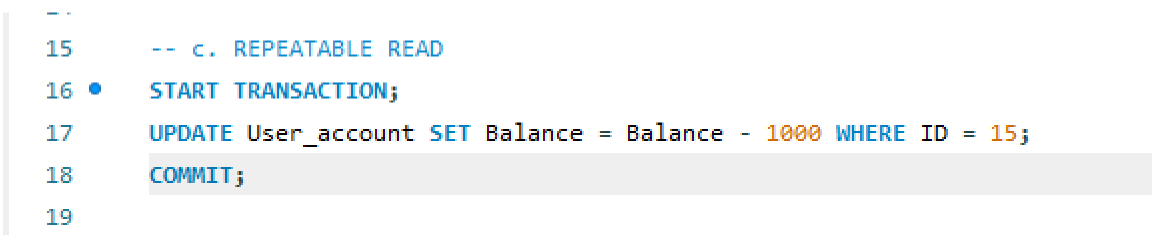
c. Repeatable Read

section 1



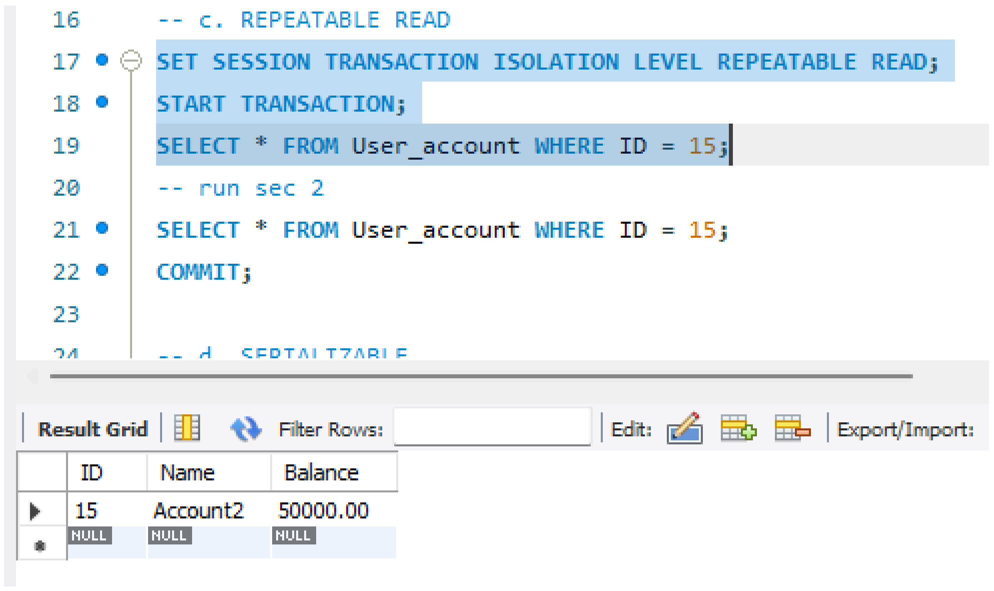
```
-- c. REPEATABLE READ
SET SESSION TRANSACTION ISOLATION LEVEL REPEATABLE READ;
START TRANSACTION;
SELECT * FROM User_account WHERE ID = 15;
-- run sec 2
SELECT * FROM User_account WHERE ID = 15;
COMMIT;
```

section 2



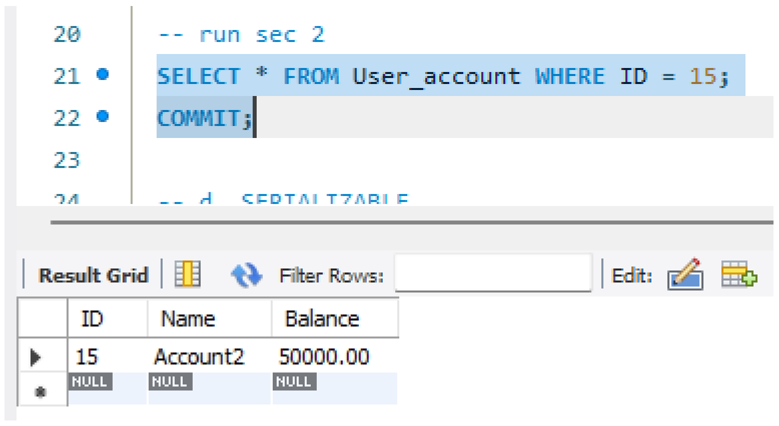
```
-- c. REPEATABLE READ
START TRANSACTION;
UPDATE User_account SET Balance = Balance - 1000 WHERE ID = 15;
COMMIT;
```

Result:



```
-- c. REPEATABLE READ
SET SESSION TRANSACTION ISOLATION LEVEL REPEATABLE READ;
START TRANSACTION;
SELECT * FROM User_account WHERE ID = 15;
-- run sec 2
SELECT * FROM User_account WHERE ID = 15;
COMMIT;

-- d. SERIALIZABLE
```

	ID	Name	Balance
▶	15	Account2	50000.00
*	NULL	NULL	NULL


```
-- run sec 2
SELECT * FROM User_account WHERE ID = 15;
COMMIT;
```

Result Grid | Filter Rows: | Edit:

	ID	Name	Balance
▶	15	Account2	50000.00
*	NULL	NULL	NULL

1. Section 1 read Account 15 (see 50000)

2. Section 2 update Account 15 and commits (change to 49000)

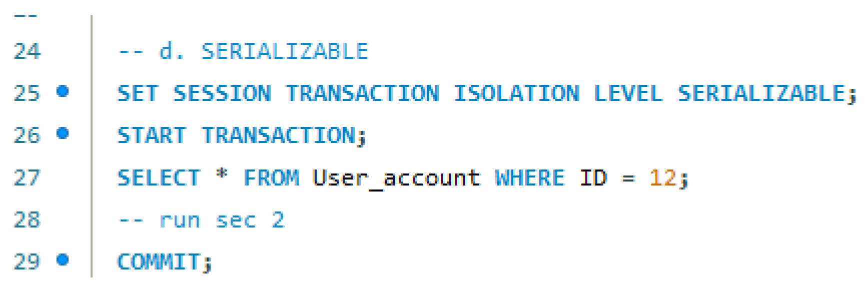
3. Section 1 read Account 15 again

4. Section 1 still see 50000 because of reapeatable read

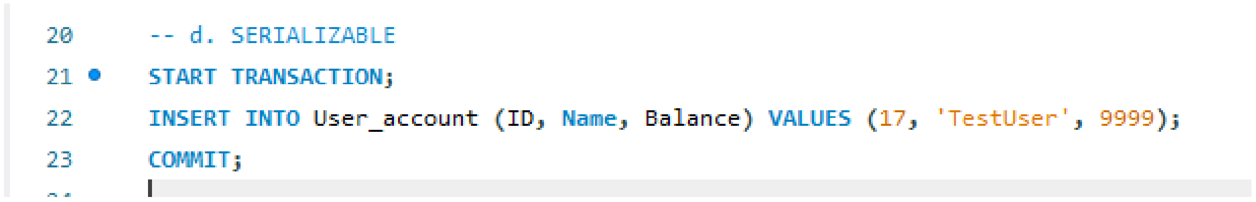
Section 1 two SELECT statements return the same balance (50000) even though Section 2 modified and commit a change.

d. SERIALIZABLE

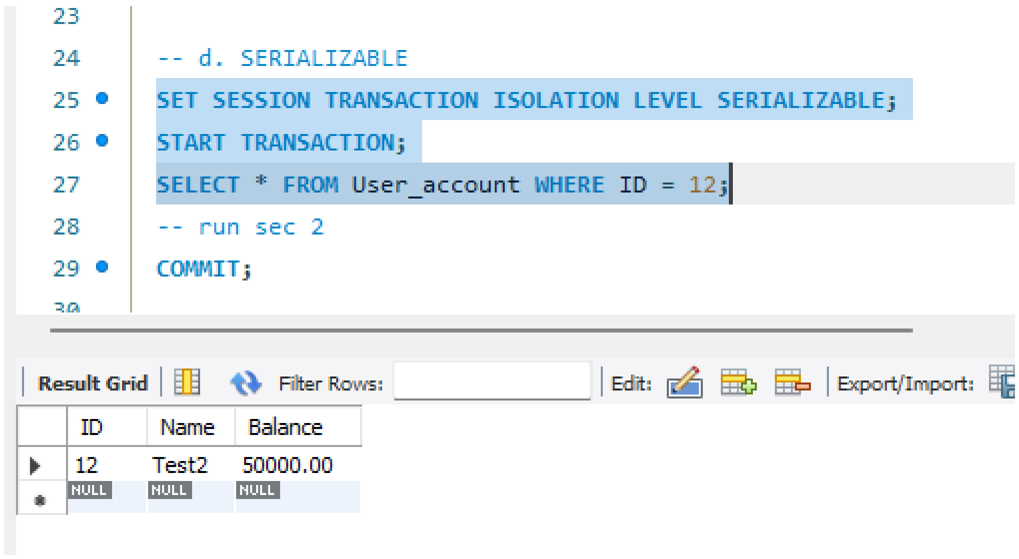
section 1:


```
-- d. SERIALIZABLE
SET SESSION TRANSACTION ISOLATION LEVEL SERIALIZABLE;
START TRANSACTION;
SELECT * FROM User_account WHERE ID = 12;
-- run sec 2
COMMIT;
```

section2:

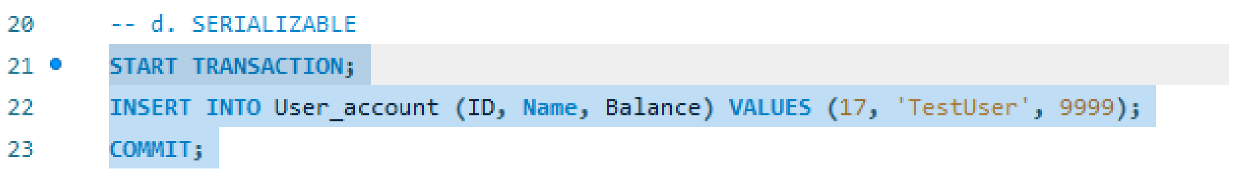

```
-- d. SERIALIZABLE
START TRANSACTION;
INSERT INTO User_account (ID, Name, Balance) VALUES (17, 'TestUser', 9999);
COMMIT;
```

Result:



```sql
-- d. SERIALIZABLE
SET SESSION TRANSACTION ISOLATION LEVEL SERIALIZABLE;
START TRANSACTION;
SELECT * FROM User_account WHERE ID = 12;
-- run sec 2
COMMIT;
```

ID	Name	Balance
12	Test2	50000.00
NULL	NULL	NULL



```sql
-- d. SERIALIZABLE
START TRANSACTION;
INSERT INTO User_account (ID, Name, Balance) VALUES (17, 'TestUser', 9999);
COMMIT;
```

1. Section 1 read Account 12 and place a lock on the related data range

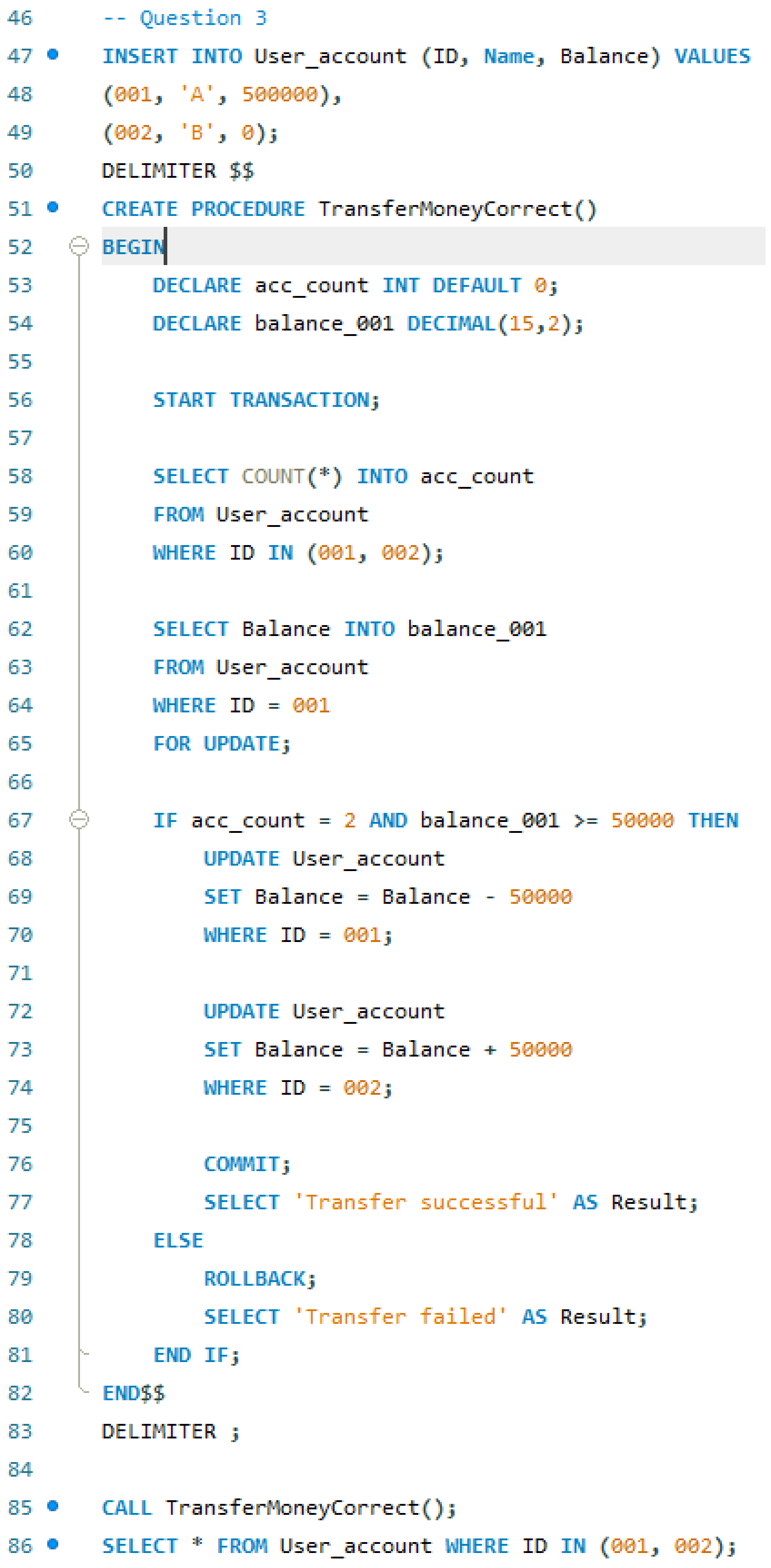
2. Section 2 tries to insert a new account or modify data in the locked range

3. Section 2 must wait until Section 1 completes

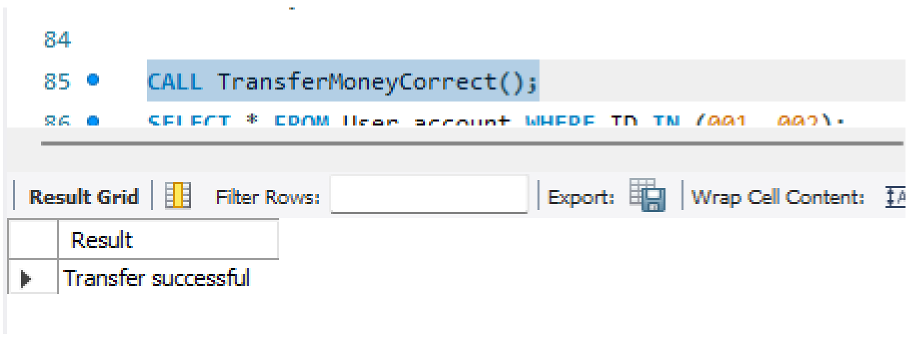
4. After Section 1 commit, Section 2 can proceed

Section 2 is blocked and must wait for Section 1 finish. This prevents any interference between transactions.

Question 3



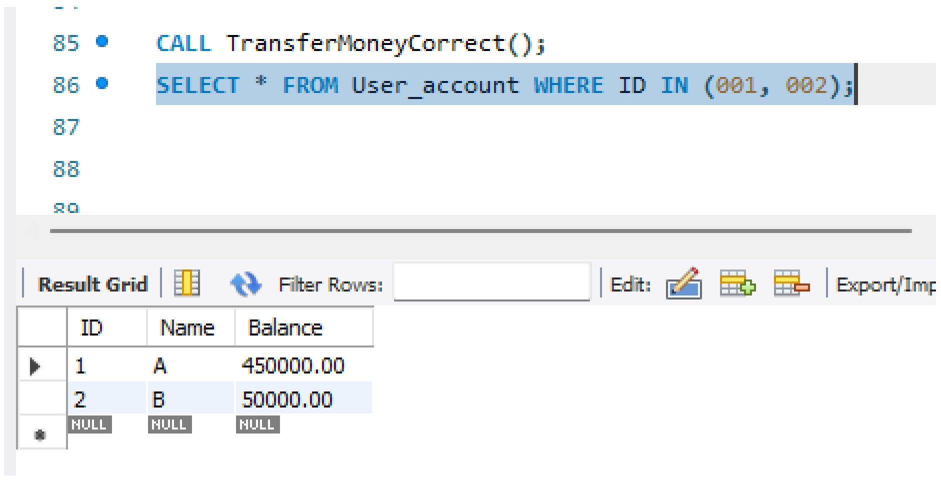
```sql
-- Question 3
INSERT INTO User_account (ID, Name, Balance) VALUES
(001, 'A', 500000),
(002, 'B', 0);
DELIMITER $$
CREATE PROCEDURE TransferMoneyCorrect()
BEGIN
    DECLARE acc_count INT DEFAULT 0;
    DECLARE balance_001 DECIMAL(15,2);

    START TRANSACTION;

    SELECT COUNT(*) INTO acc_count
    FROM User_account
    WHERE ID IN (001, 002);

    SELECT Balance INTO balance_001
    FROM User_account
    WHERE ID = 001
    FOR UPDATE;

    IF acc_count = 2 AND balance_001 >= 50000 THEN
        UPDATE User_account
        SET Balance = Balance - 50000
        WHERE ID = 001;

        UPDATE User_account
        SET Balance = Balance + 50000
        WHERE ID = 002;

        COMMIT;
        SELECT 'Transfer successful' AS Result;
    ELSE
        ROLLBACK;
        SELECT 'Transfer failed' AS Result;
    END IF;
END$$
DELIMITER ;

CALL TransferMoneyCorrect();
SELECT * FROM User_account WHERE ID IN (001, 002);
```



```
CALL TransferMoneyCorrect();
```

	Result
▶	Transfer successful

result



```
CALL TransferMoneyCorrect();
SELECT * FROM User_account WHERE ID IN (001, 002);
```

	ID	Name	Balance
▶	1	A	450000.00
	2	B	50000.00
*	NULL	NULL	NULL

1. Validation Check: checks if both accounts exist (acc_count = 2)

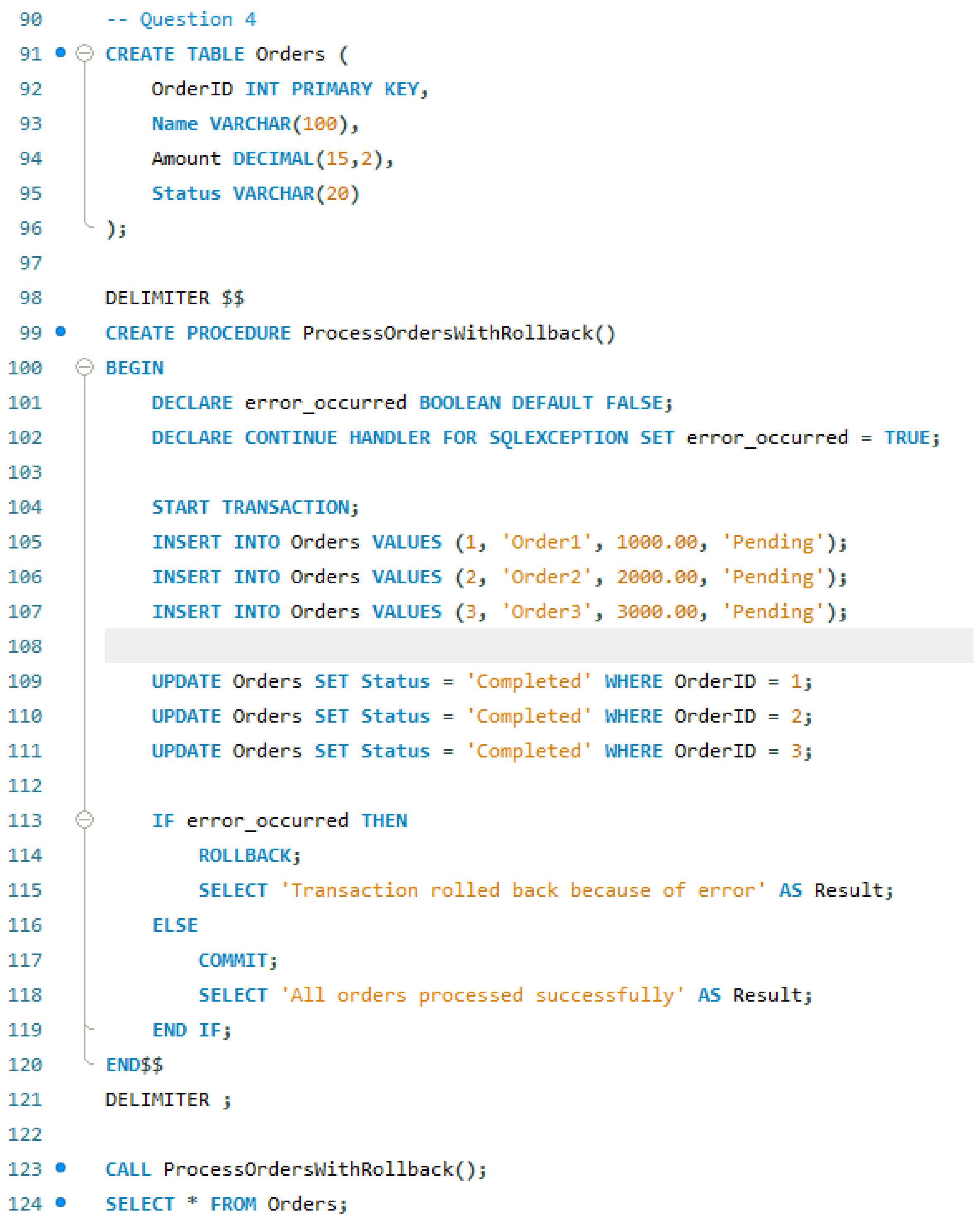
2. Balance Check: Verify Account 001 has eenough balance (>= 50000)

3. Locks the account row to prevent other transactions from changing it

4. If checks pass: Transfers 50000 and commits. If checks fail: Rolls back and returns failure message

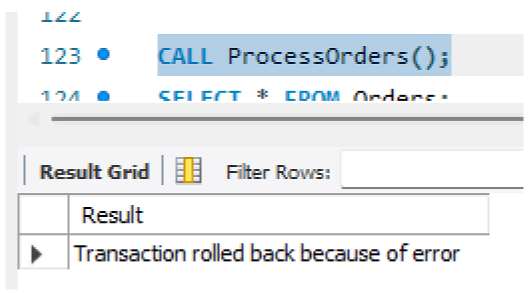
The transaction validate both accounts and the balance, then transfers 50000 from Account A to Account B.

Question 4

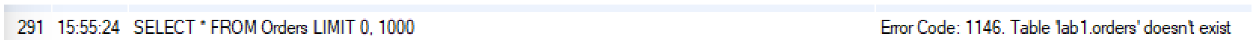


```sql
-- Question 4
CREATE TABLE Orders (
    OrderID INT PRIMARY KEY,
    Name VARCHAR(100),
    Amount DECIMAL(15,2),
    Status VARCHAR(20)
);

DELIMITER $$
CREATE PROCEDURE ProcessOrdersWithRollback()
BEGIN
    DECLARE error_occurred BOOLEAN DEFAULT FALSE;
    DECLARE CONTINUE HANDLER FOR SQLEXCEPTION SET error_occurred = TRUE;

    START TRANSACTION;
    INSERT INTO Orders VALUES (1, 'Order1', 1000.00, 'Pending');
    INSERT INTO Orders VALUES (2, 'Order2', 2000.00, 'Pending');
    INSERT INTO Orders VALUES (3, 'Order3', 3000.00, 'Pending');

    UPDATE Orders SET Status = 'Completed' WHERE OrderID = 1;
    UPDATE Orders SET Status = 'Completed' WHERE OrderID = 2;
    UPDATE Orders SET Status = 'Completed' WHERE OrderID = 3;

    IF error_occurred THEN
        ROLLBACK;
        SELECT 'Transaction rolled back because of error' AS Result;
    ELSE
        COMMIT;
        SELECT 'All orders processed successfully' AS Result;
    END IF;
END$$
DELIMITER ;

CALL ProcessOrdersWithRollback();
SELECT * FROM Orders;
```

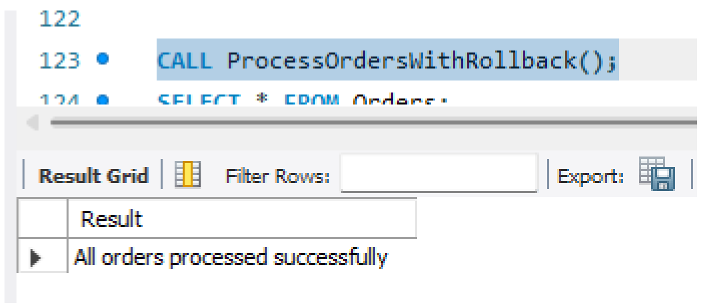
If error:


```
CALL ProcessOrders();
SELECT * FROM Orders;
```

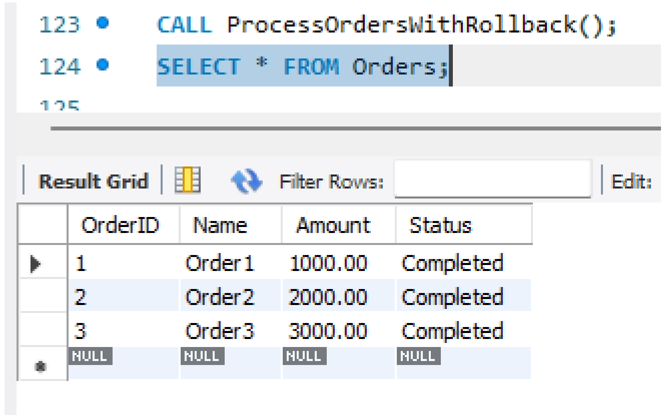
	Result
▶	Transaction rolled back because of error


```
291  15:55:24  SELECT * FROM Orders LIMIT 0, 1000          Error Code: 1146. Table 'lab1.orders' doesn't exist
```

Result:


```
CALL ProcessOrdersWithRollback();
SELECT * FROM Orders;
```

	Result
▶	All orders processed successfully


```
CALL ProcessOrdersWithRollback();
SELECT * FROM Orders;
```

	OrderID	Name	Amount	Status
▶	1	Order1	1000.00	Completed
	2	Order2	2000.00	Completed
	3	Order3	3000.00	Completed
*	NULL	NULL	NULL	NULL

1. DECLARE CONTINUE HANDLER catch SQL exceptions

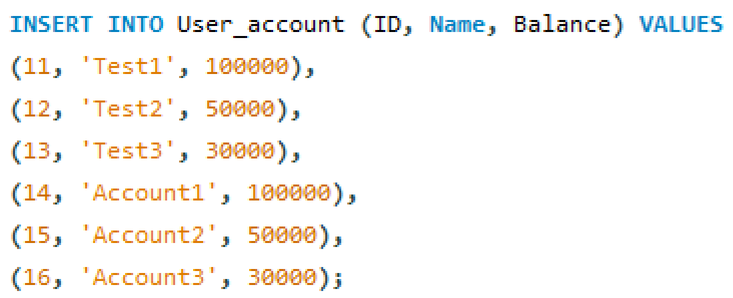
2. Inserts 3 orders and updates their status

3. If any operation fails, sets error_occurred to true

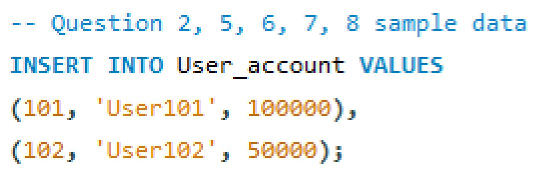
4. If error: ROLLBACK (undo all changes). If no error: COMMIT (save all changes)

All 3 orders are inserted and status updated to 'Completed'. If error occurs the error handler catch it, roll back all changes, and no orders appear in the database.

Question 5,6,7,8 test data:



```
INSERT INTO User_account (ID, Name, Balance) VALUES
(11, 'Test1', 100000),
(12, 'Test2', 50000),
(13, 'Test3', 30000),
(14, 'Account1', 100000),
(15, 'Account2', 50000),
(16, 'Account3', 30000);
```



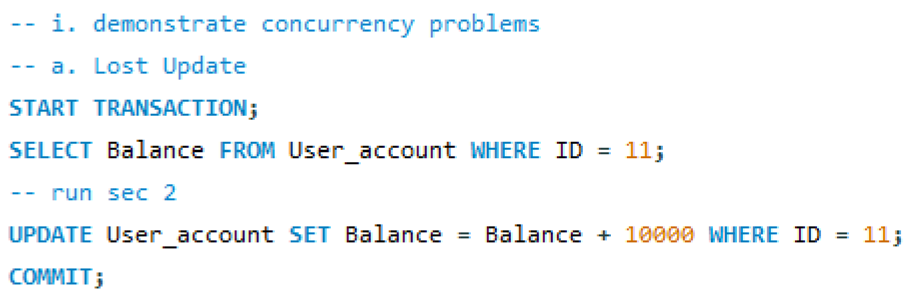
```
-- Question 2, 5, 6, 7, 8 sample data
INSERT INTO User_account VALUES
(101, 'User101', 100000),
(102, 'User102', 50000);
```

Question 5:

DEMONSTRATE CONCURRENCY PROBLEMS

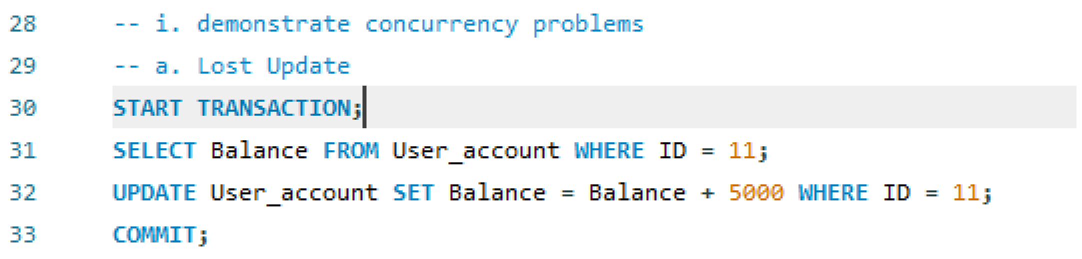
- a/ Lost Update

Section1:



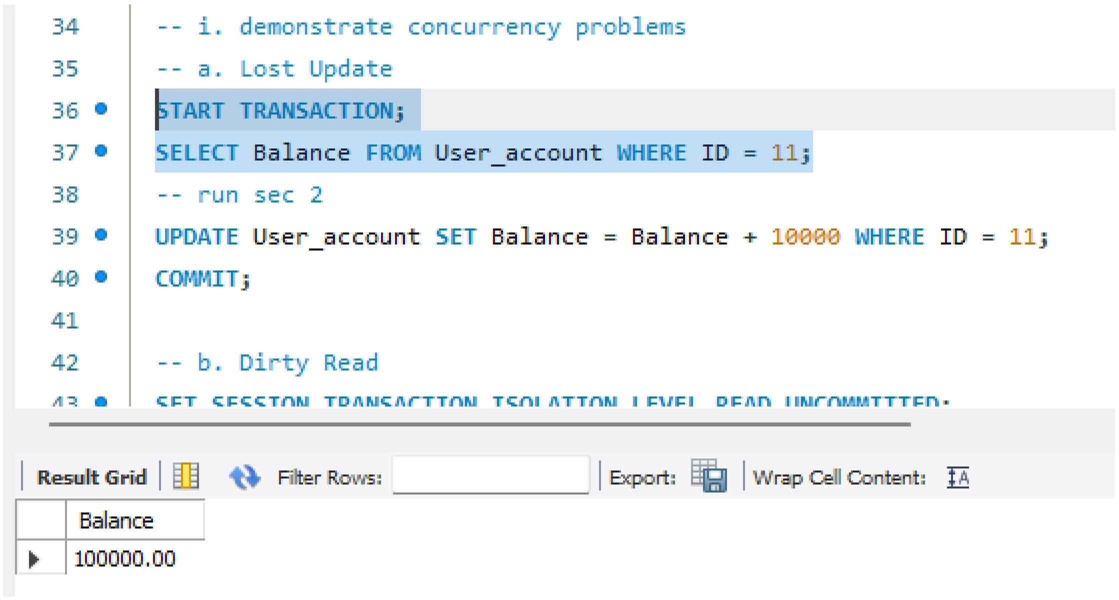
```
-- i. demonstrate concurrency problems
-- a. Lost Update
START TRANSACTION;
SELECT Balance FROM User_account WHERE ID = 11;
-- run sec 2
UPDATE User_account SET Balance = Balance + 10000 WHERE ID = 11;
COMMIT;
```

Section 2



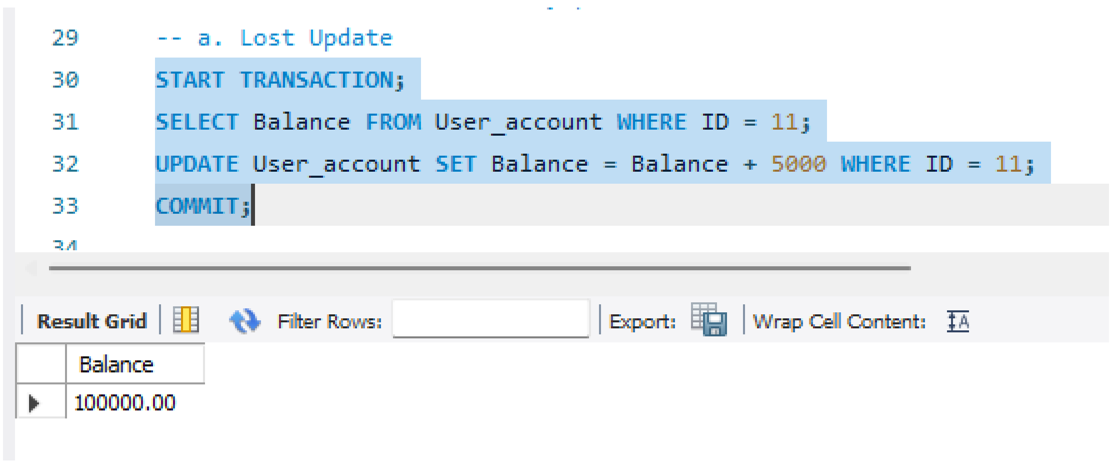
```
-- i. demonstrate concurrency problems
-- a. Lost Update
START TRANSACTION;
SELECT Balance FROM User_account WHERE ID = 11;
UPDATE User_account SET Balance = Balance + 5000 WHERE ID = 11;
COMMIT;
```

Result



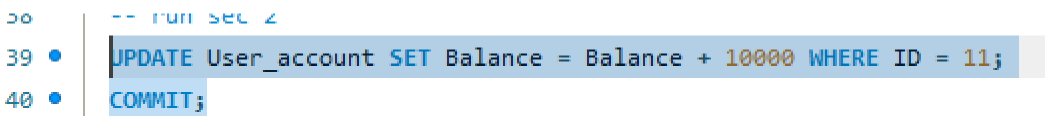
```
-- i. demonstrate concurrency problems
-- a. Lost Update
START TRANSACTION;
SELECT Balance FROM User_account WHERE ID = 11;
-- run sec 2
UPDATE User_account SET Balance = Balance + 10000 WHERE ID = 11;
COMMIT;

-- b. Dirty Read
SET SESSION TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
```

Balance
100000.00



```
-- a. Lost Update
START TRANSACTION;
SELECT Balance FROM User_account WHERE ID = 11;
UPDATE User_account SET Balance = Balance + 5000 WHERE ID = 11;
COMMIT;
```

Balance
100000.00



```
-- run sec 2
UPDATE User_account SET Balance = Balance + 10000 WHERE ID = 11;
COMMIT;
```

1. Time 1: Section 1 reads balance

2. Time 2: Section 2 reads balance

3. Time 3: Section 2 update

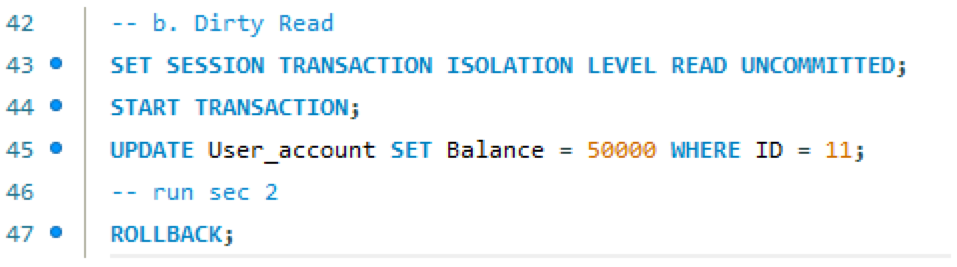
4. Time 4: Section 1 update

5. Final Result: Balance is 100000, but should be 115000

The final balance is 100000 instead of the expected 115000. Section 2 update was overwritten by Section 1 update because both read the initial balance before either committed

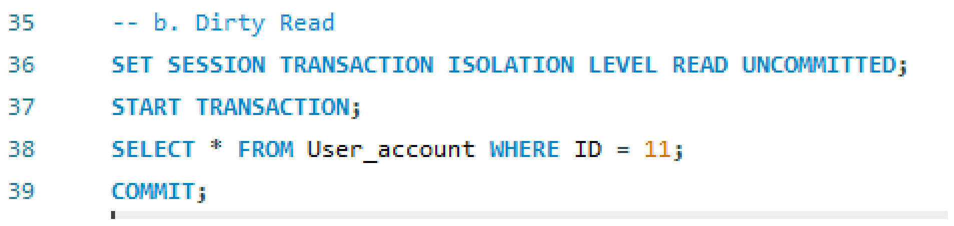
- b/ Dirty Read

Section1:



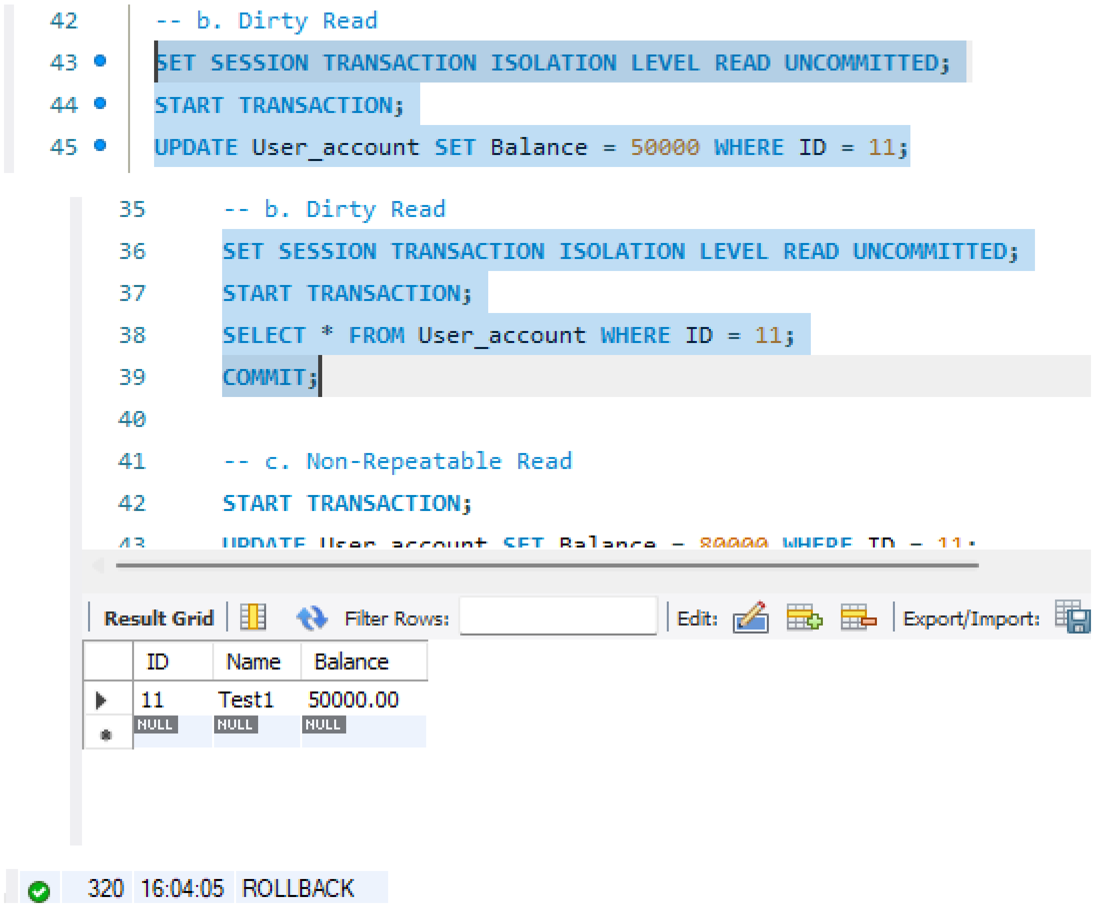
```
-- b. Dirty Read
SET SESSION TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
START TRANSACTION;
UPDATE User_account SET Balance = 50000 WHERE ID = 11;
-- run sec 2
ROLLBACK;
```

Section 2



```
-- b. Dirty Read
SET SESSION TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
START TRANSACTION;
SELECT * FROM User_account WHERE ID = 11;
COMMIT;
```

Result



```
-- b. Dirty Read
SET SESSION TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
START TRANSACTION;
UPDATE User_account SET Balance = 50000 WHERE ID = 11;
```

```
-- b. Dirty Read
SET SESSION TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
START TRANSACTION;
SELECT * FROM User_account WHERE ID = 11;
COMMIT;

-- c. Non-Repeatable Read
START TRANSACTION;
UPDATE User_account SET Balance = 80000 WHERE ID = 11;
```

ID	Name	Balance
11	Test1	50000.00
NULL	NULL	NULL

```
320 16:04:05 ROLLBACK
```

1. Section 1 change balance to 50000

2. Section 1 hasnt committed

3. Section 2 reads the account and see 50000

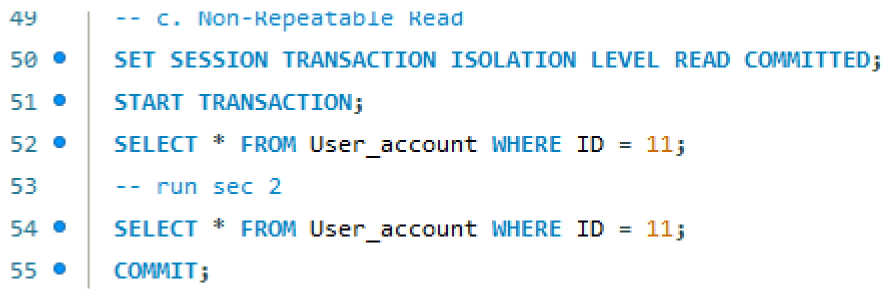
4. Section 1 rolls back

5. The actual balance is back to original, but Section 2 has saw 50000

Section 2 read 50000 even though this value is never committed. After Section 1 roll back, the database still has the original balance.

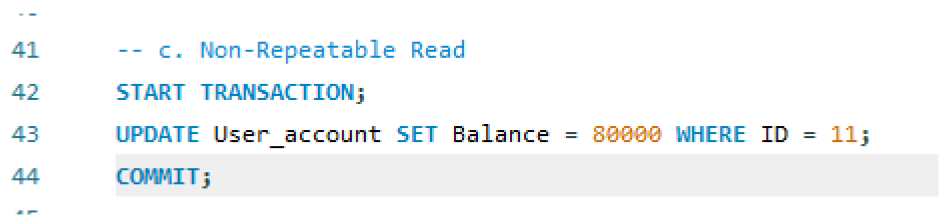
- **c/ Non-Repeatable Read**

Section1:



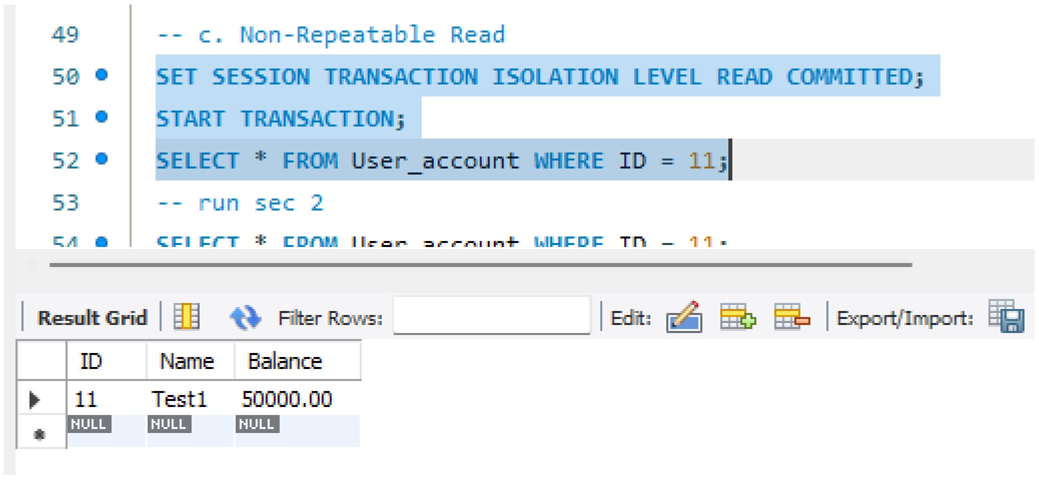
```sql
-- c. Non-Repeatable Read
SET SESSION TRANSACTION ISOLATION LEVEL READ COMMITTED;
START TRANSACTION;
SELECT * FROM User_account WHERE ID = 11;
-- run sec 2
SELECT * FROM User_account WHERE ID = 11;
COMMIT;
```

Section 2



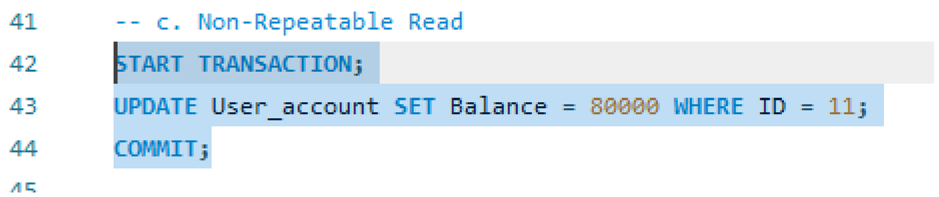
```sql
-- c. Non-Repeatable Read
START TRANSACTION;
UPDATE User_account SET Balance = 80000 WHERE ID = 11;
COMMIT;
```

Result

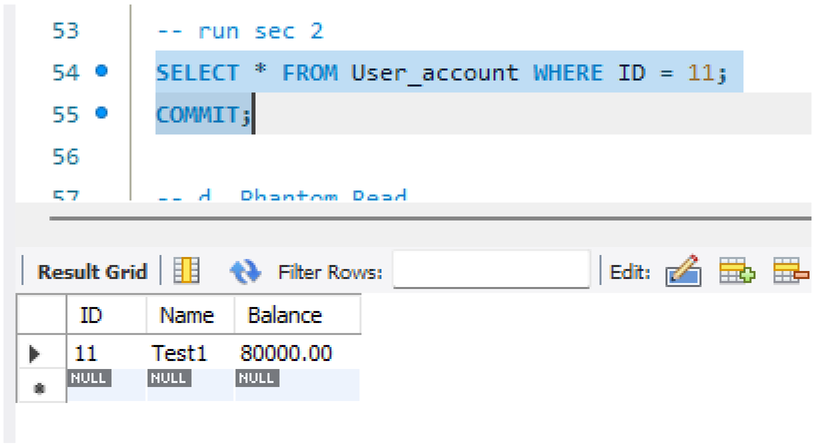


```sql
-- c. Non-Repeatable Read
SET SESSION TRANSACTION ISOLATION LEVEL READ COMMITTED;
START TRANSACTION;
SELECT * FROM User_account WHERE ID = 11;
-- run sec 2
SELECT * FROM User_account WHERE ID = 11;
```

ID	Name	Balance
11	Test1	50000.00
NULL	NULL	NULL



```sql
-- c. Non-Repeatable Read
START TRANSACTION;
UPDATE User_account SET Balance = 80000 WHERE ID = 11;
COMMIT;
```



```sql
-- run sec 2
SELECT * FROM User_account WHERE ID = 11;
COMMIT;

-- d. Phantom Read
```

ID	Name	Balance
11	Test1	80000.00
NULL	NULL	NULL

1. Section 1 reads Account 11

2. Section 2 updates Account 11 to 80000 and commit

3. Section 1 reads Account 11 again

4. Return different result

Section 1 first SELECT returns the original balance, but the second SELECT returns 80000

- d/ Phantom Read

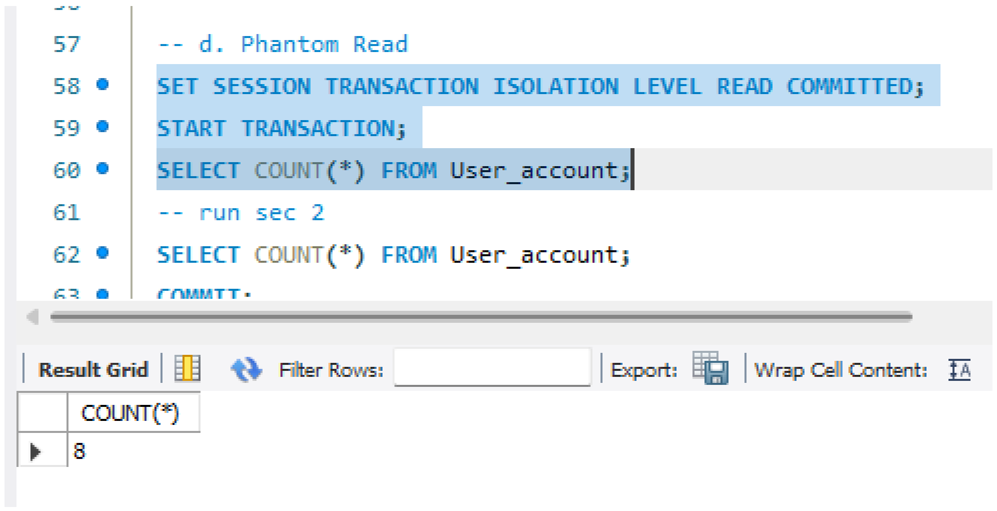
Section1:

```sql
-- d. Phantom Read
SET SESSION TRANSACTION ISOLATION LEVEL READ COMMITTED;
START TRANSACTION;
SELECT COUNT(*) FROM User_account;
-- run sec 2
SELECT COUNT(*) FROM User_account;
COMMIT;
```

Section 2

```sql
-- d. Phantom Read
START TRANSACTION;
INSERT INTO User_account VALUES (18, 'NewUser', 25000);
COMMIT;
```

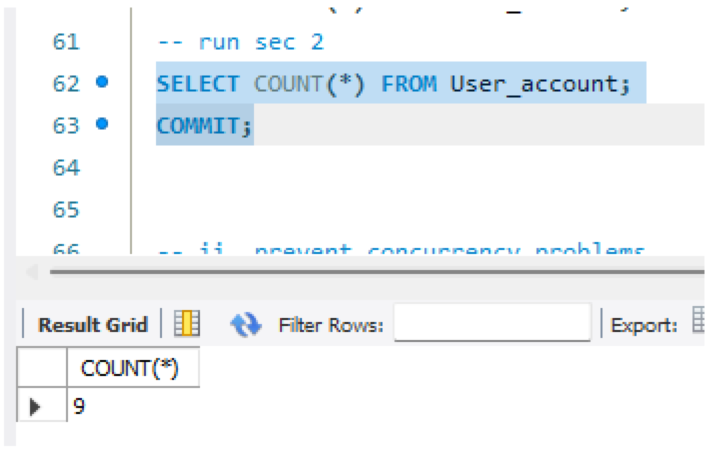
Result



```sql
-- d. Phantom Read
SET SESSION TRANSACTION ISOLATION LEVEL READ COMMITTED;
START TRANSACTION;
SELECT COUNT(*) FROM User_account;
-- run sec 2
SELECT COUNT(*) FROM User_account;
```

```
COUNT(*)
8
```

```sql
-- d. Phantom Read
START TRANSACTION;
INSERT INTO User_account VALUES (18, 'NewUser', 25000);
COMMIT;
```


```
-- run sec 2
SELECT COUNT(*) FROM User_account;
COMMIT;


-- ii. prevent concurrency problems
```
Result Grid | Filter Rows: | Export:

```
COUNT(*)
9
```

1. Section 1 counts accounts

2. Section 2 inserts a new account and commits

3. Section 1 count again

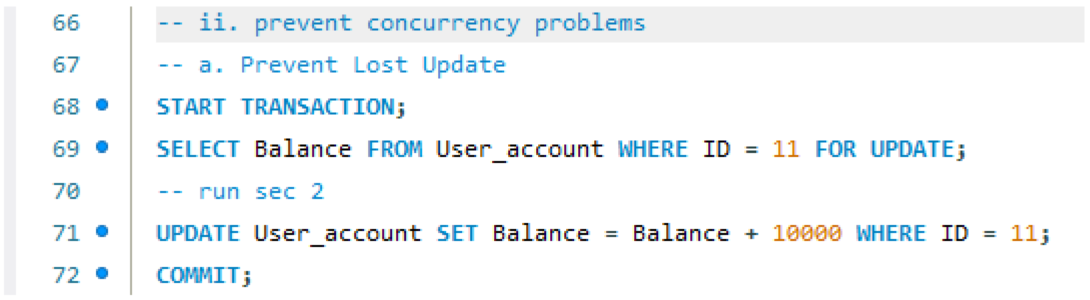
4. New phantom row appeared during the transaction

Section 1 first COUNT returns the original number of accounts. After Section 2 insert and commits, Section 1 second COUNT is higher.

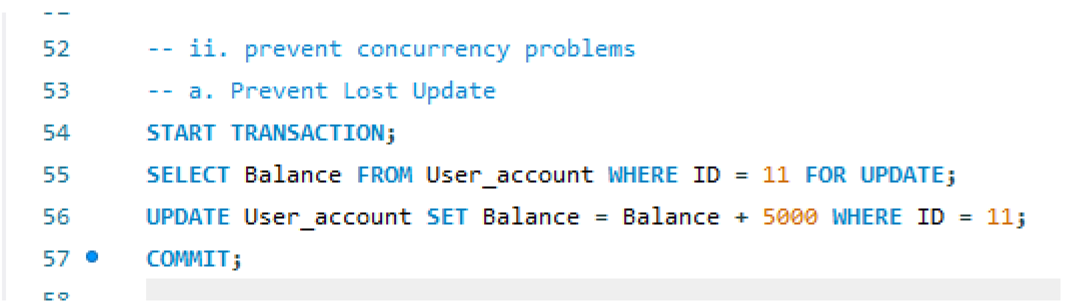
PREVENT CONCURRENCY PROBLEMS

- a/ Prevent Lost Update

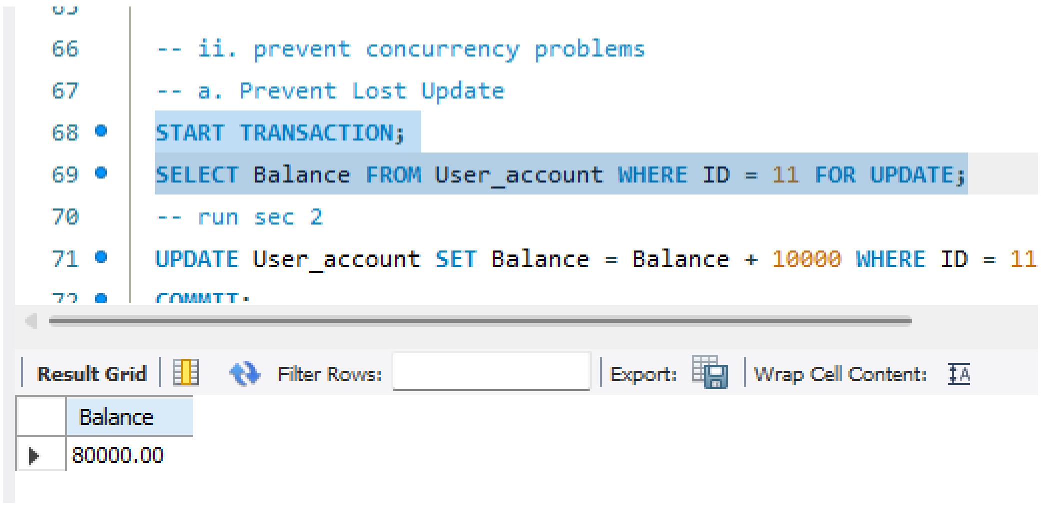
Section1:


```
-- ii. prevent concurrency problems
-- a. Prevent Lost Update
START TRANSACTION;
SELECT Balance FROM User_account WHERE ID = 11 FOR UPDATE;
-- run sec 2
UPDATE User_account SET Balance = Balance + 10000 WHERE ID = 11;
COMMIT;
```

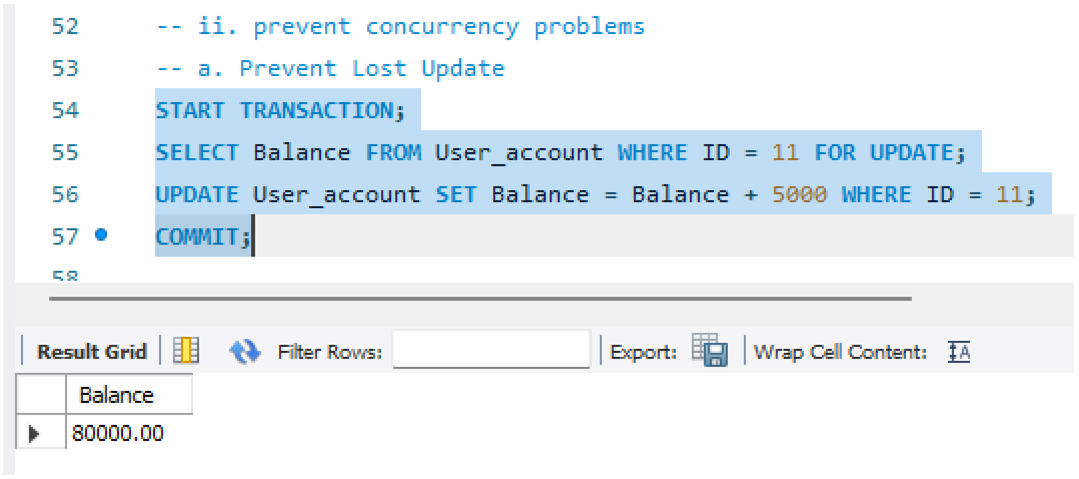
Section 2


```sql
-- ii. prevent concurrency problems
-- a. Prevent Lost Update
START TRANSACTION;
SELECT Balance FROM User_account WHERE ID = 11 FOR UPDATE;
UPDATE User_account SET Balance = Balance + 5000 WHERE ID = 11;
COMMIT;
```

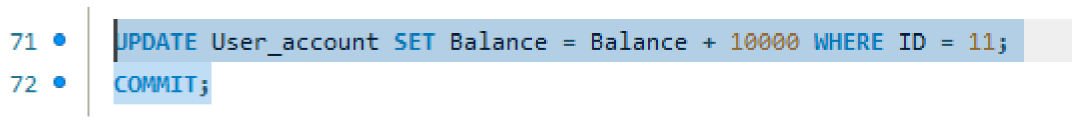
Result


```sql
-- ii. prevent concurrency problems
-- a. Prevent Lost Update
START TRANSACTION;
SELECT Balance FROM User_account WHERE ID = 11 FOR UPDATE;
-- run sec 2
UPDATE User_account SET Balance = Balance + 10000 WHERE ID = 11
COMMIT;
```

	Balance
▶	80000.00


```sql
-- ii. prevent concurrency problems
-- a. Prevent Lost Update
START TRANSACTION;
SELECT Balance FROM User_account WHERE ID = 11 FOR UPDATE;
UPDATE User_account SET Balance = Balance + 5000 WHERE ID = 11;
COMMIT;
```

	Balance
▶	80000.00


```sql
UPDATE User_account SET Balance = Balance + 10000 WHERE ID = 11;
COMMIT;
```

1. Section 1 reads and LOCKS Account 11

2. Section 2 tries to read Account 11

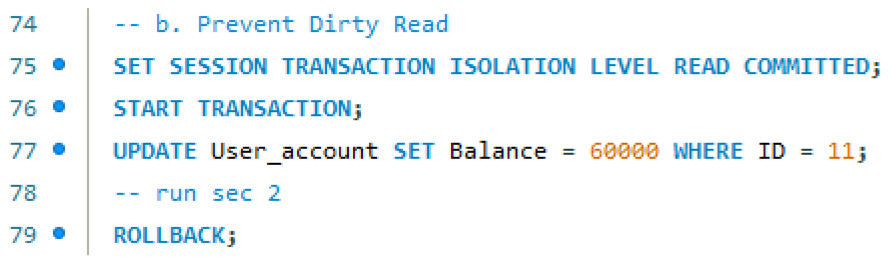
3. Section 1 updates and commit

4. Section 2 can proceed

Section 2 waits for Section 1 to complete before proceeding. Both updates are preserved, preventing data loss

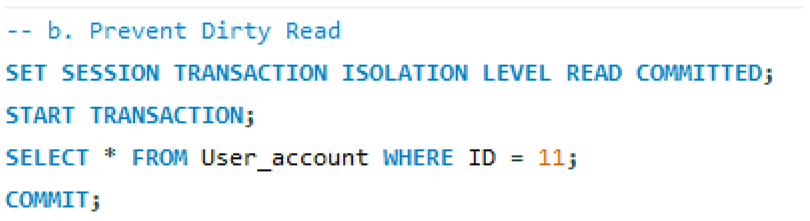
- b/ Prevent Dirty Read

Section1:



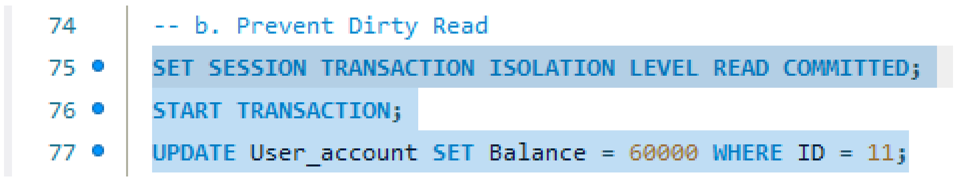
```sql
-- b. Prevent Dirty Read
SET SESSION TRANSACTION ISOLATION LEVEL READ COMMITTED;
START TRANSACTION;
UPDATE User_account SET Balance = 60000 WHERE ID = 11;
-- run sec 2
ROLLBACK;
```

Section 2

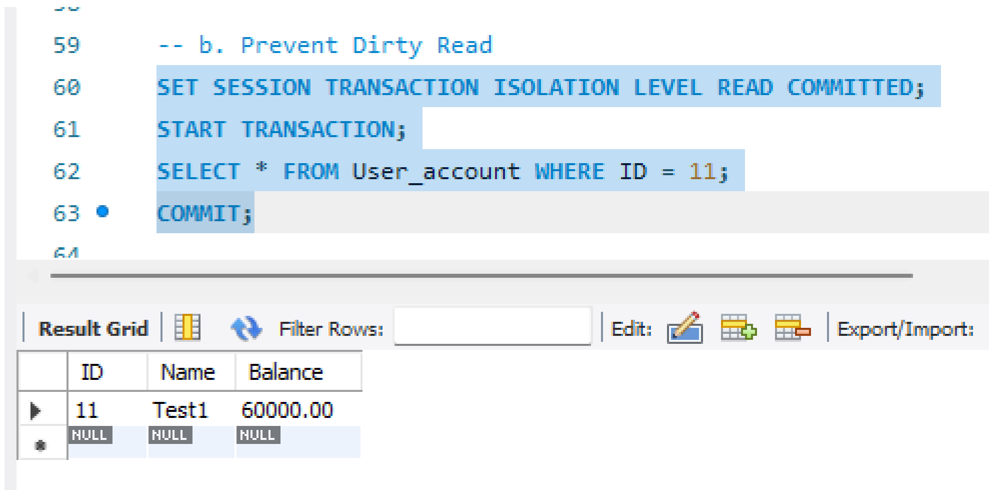


```sql
-- b. Prevent Dirty Read
SET SESSION TRANSACTION ISOLATION LEVEL READ COMMITTED;
START TRANSACTION;
SELECT * FROM User_account WHERE ID = 11;
COMMIT;
```

Result



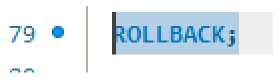
```sql
-- b. Prevent Dirty Read
SET SESSION TRANSACTION ISOLATION LEVEL READ COMMITTED;
START TRANSACTION;
UPDATE User_account SET Balance = 60000 WHERE ID = 11;
```



```
-- b. Prevent Dirty Read
SET SESSION TRANSACTION ISOLATION LEVEL READ COMMITTED;
START TRANSACTION;
SELECT * FROM User_account WHERE ID = 11;
COMMIT;
```

Result Grid | Filter Rows: | Edit: | Export/Import:

ID	Name	Balance
11	Test1	60000.00
NULL	NULL	NULL



```
ROLLBACK;
```

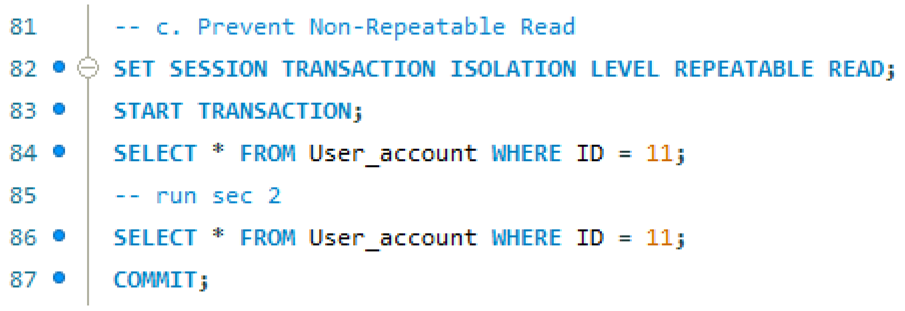
1. Section 1 updates balance to 60,000

2. Section 2 reads Account 11

4. Section 1 rolls back

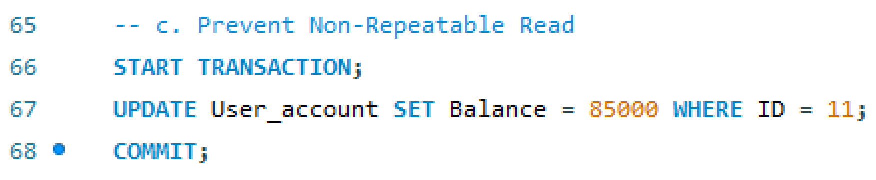
- c/ Prevent Non-Repeatable Read

Section1:



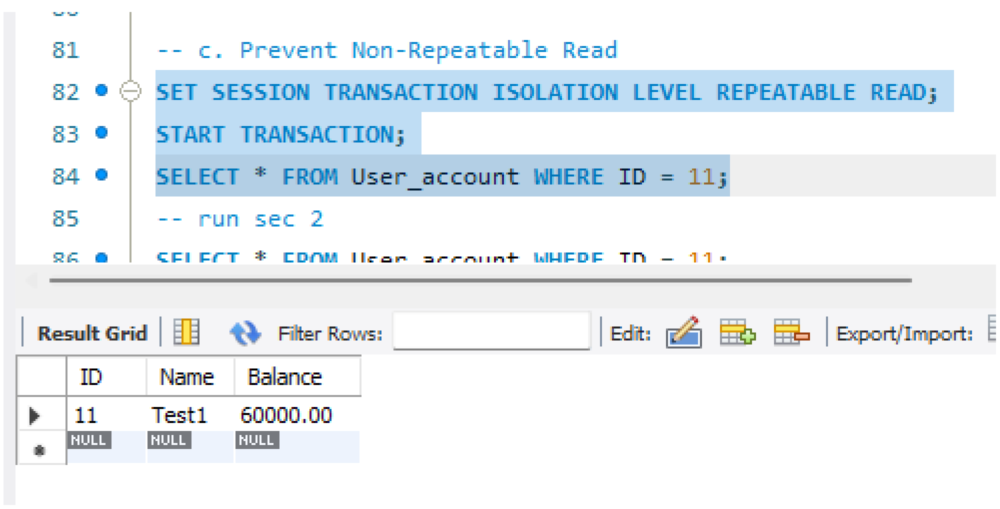
```
-- c. Prevent Non-Repeatable Read
SET SESSION TRANSACTION ISOLATION LEVEL REPEATABLE READ;
START TRANSACTION;
SELECT * FROM User_account WHERE ID = 11;
-- run sec 2
SELECT * FROM User_account WHERE ID = 11;
COMMIT;
```

Section 2

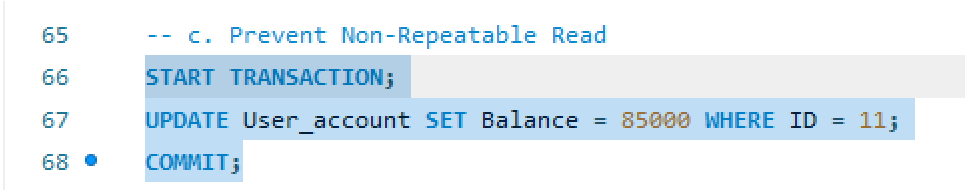


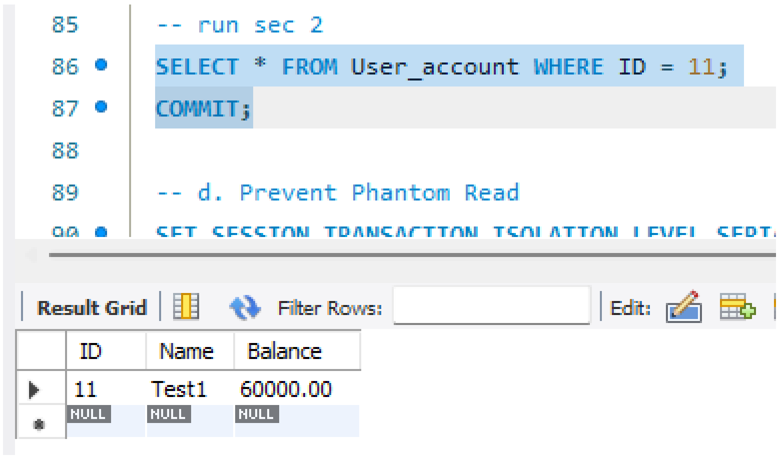
```
-- c. Prevent Non-Repeatable Read
START TRANSACTION;
UPDATE User_account SET Balance = 85000 WHERE ID = 11;
COMMIT;
```

Result


```
-- c. Prevent Non-Repeatable Read
SET SESSION TRANSACTION ISOLATION LEVEL REPEATABLE READ;
START TRANSACTION;
SELECT * FROM User_account WHERE ID = 11;
-- run sec 2
SELECT * FROM User_account WHERE ID = 11;
```

ID	Name	Balance
11	Test1	60000.00
NULL	NULL	NULL


```
-- c. Prevent Non-Repeatable Read
START TRANSACTION;
UPDATE User_account SET Balance = 85000 WHERE ID = 11;
COMMIT;
```


```
-- run sec 2
SELECT * FROM User_account WHERE ID = 11;
COMMIT;

-- d. Prevent Phantom Read
SET SESSION TRANSACTION ISOLATION LEVEL SERI
```

ID	Name	Balance
11	Test1	60000.00
NULL	NULL	NULL

1. Section 1 reads Account 11

2. Section 2 update and commit

3. Section 1 reads Account 11 again

4. Section 1 still sees 60000

5. Consistent view in the transaction

Both of Section 1 SELECT statements return the same balance.

- d/ Prevent Phantom Read

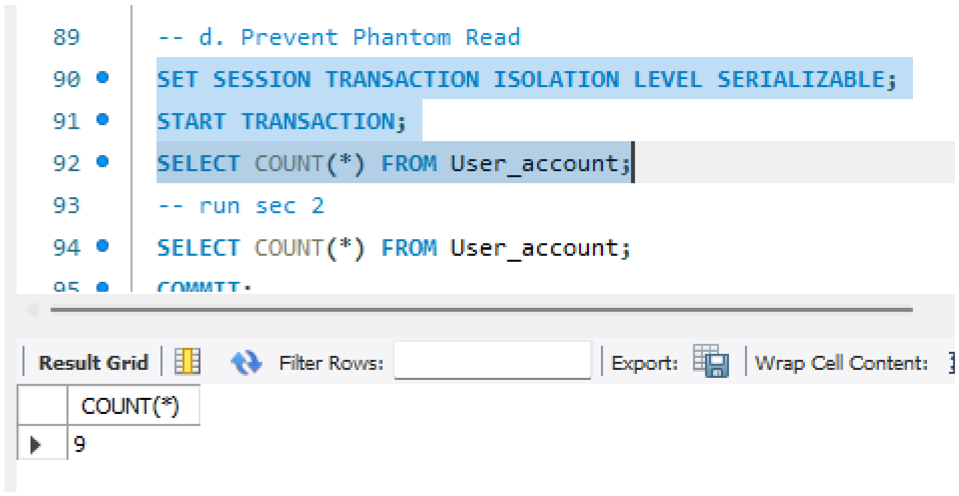
Section1:

```sql
-- d. Prevent Phantom Read
SET SESSION TRANSACTION ISOLATION LEVEL SERIALIZABLE;
START TRANSACTION;
SELECT COUNT(*) FROM User_account;
-- run sec 2
SELECT COUNT(*) FROM User_account;
COMMIT;
```

Section 2

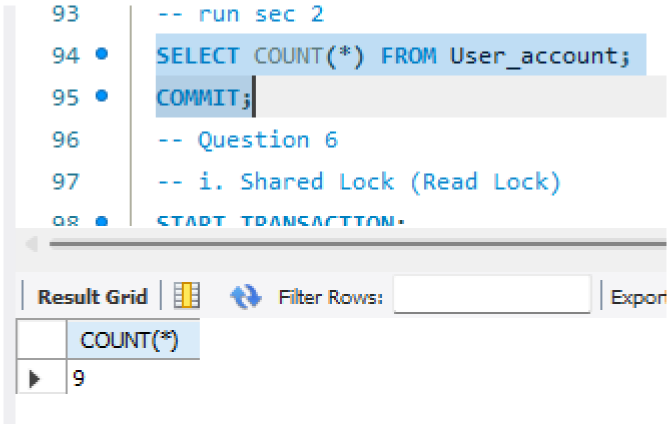
```sql
-- d. Prevent Phantom Read
START TRANSACTION;
INSERT INTO User_account VALUES (19, 'AnotherUser', 30000);
COMMIT;
```

Result



```sql
-- d. Prevent Phantom Read
SET SESSION TRANSACTION ISOLATION LEVEL SERIALIZABLE;
START TRANSACTION;
SELECT COUNT(*) FROM User_account;
-- run sec 2
SELECT COUNT(*) FROM User_account;
COMMIT;
```

```
COUNT(*)
9
```



```sql
-- run sec 2
SELECT COUNT(*) FROM User_account;
COMMIT;
-- Question 6
-- i. Shared Lock (Read Lock)
START TRANSACTION;
```

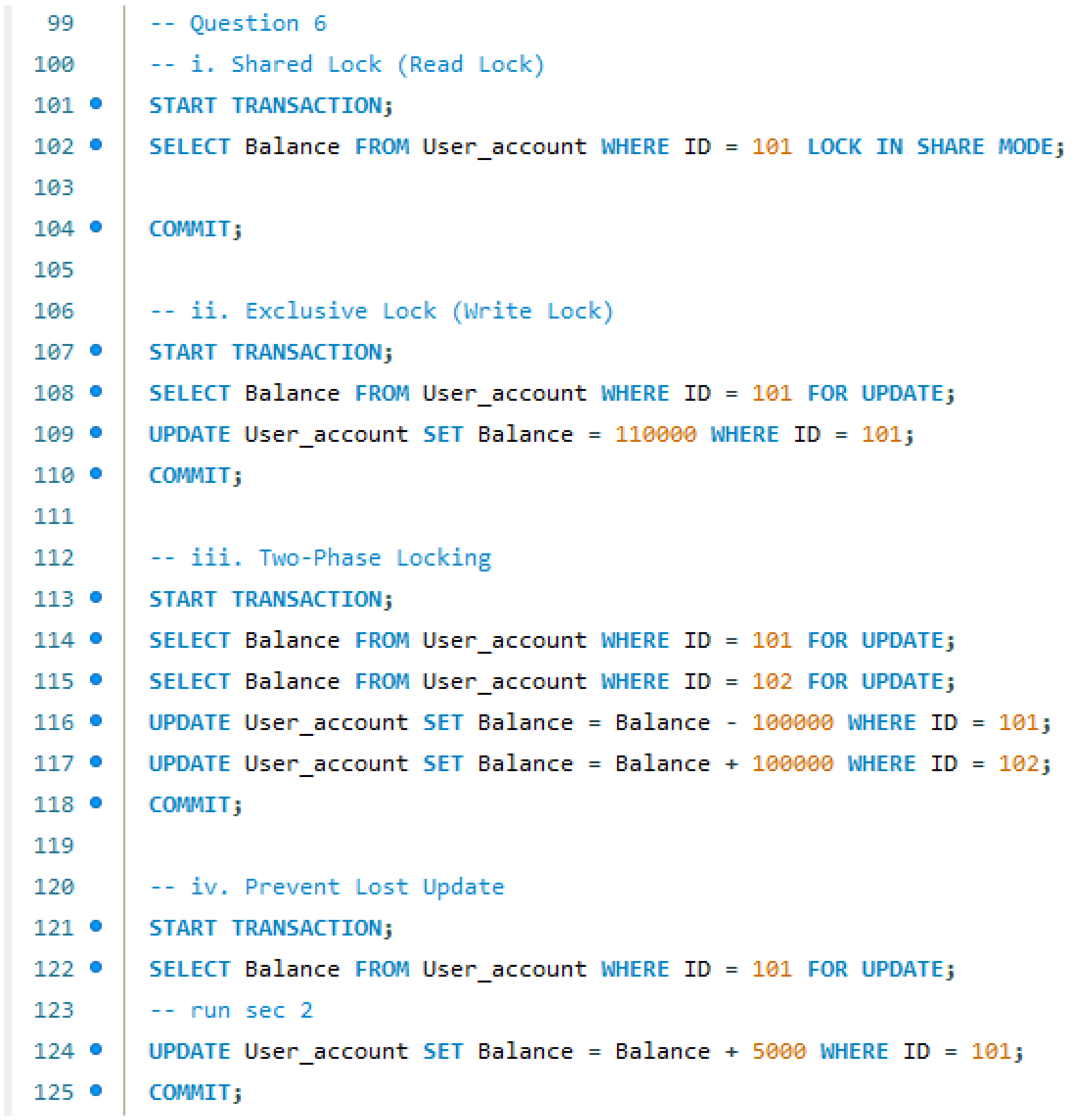
```
COUNT(*)
9
```

1. Section 1 counts accounts and locks the table range
2. Section 2 tries to insert a new account
3. Section 2 must wait until Section 1 finishes
4. Section 1 count again (same result)
5. Section 1 commits and releases lock
6. Section 2 can now insert

Both COUNT query in Section 1 return the same number.

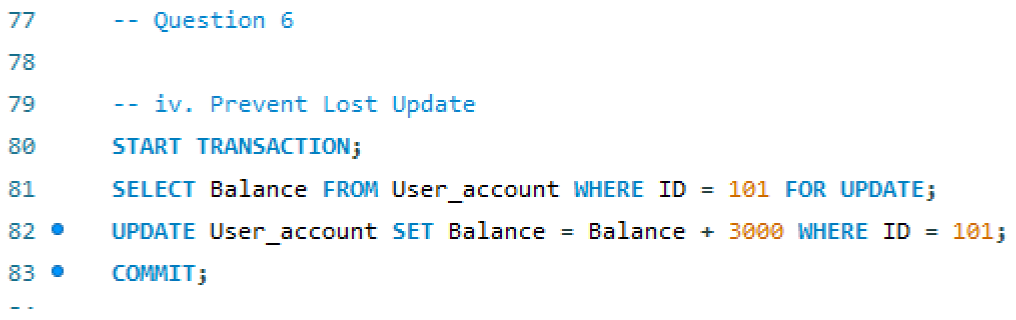
Question 6:

Section 1:



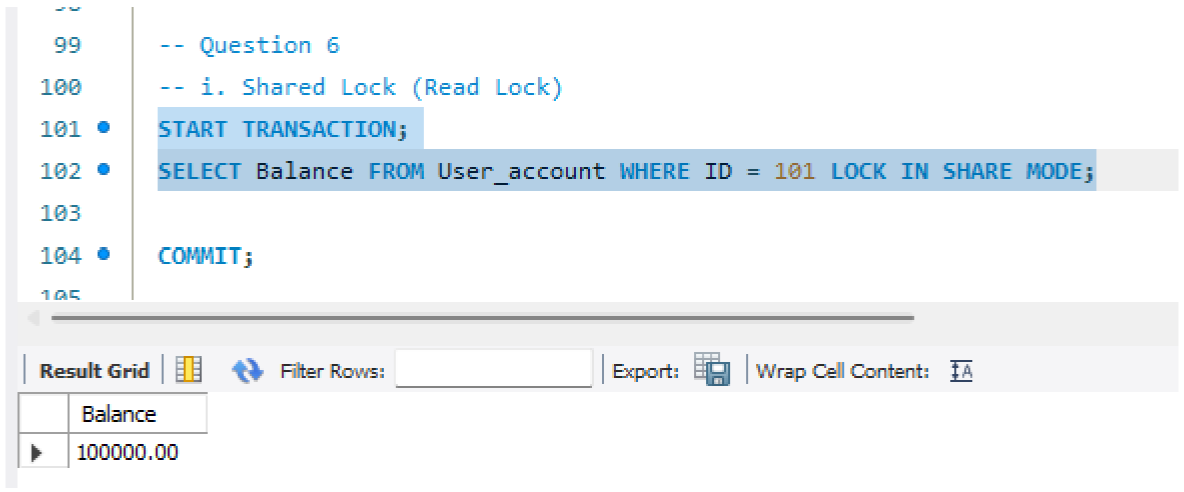
```sql
-- Question 6
-- i. Shared Lock (Read Lock)
START TRANSACTION;
SELECT Balance FROM User_account WHERE ID = 101 LOCK IN SHARE MODE;

COMMIT;

-- ii. Exclusive Lock (Write Lock)
START TRANSACTION;
SELECT Balance FROM User_account WHERE ID = 101 FOR UPDATE;
UPDATE User_account SET Balance = 110000 WHERE ID = 101;
COMMIT;

-- iii. Two-Phase Locking
START TRANSACTION;
SELECT Balance FROM User_account WHERE ID = 101 FOR UPDATE;
SELECT Balance FROM User_account WHERE ID = 102 FOR UPDATE;
UPDATE User_account SET Balance = Balance - 100000 WHERE ID = 101;
UPDATE User_account SET Balance = Balance + 100000 WHERE ID = 102;
COMMIT;

-- iv. Prevent Lost Update
START TRANSACTION;
SELECT Balance FROM User_account WHERE ID = 101 FOR UPDATE;
-- run sec 2
UPDATE User_account SET Balance = Balance + 5000 WHERE ID = 101;
COMMIT;
```

Section 2:



```
-- Question 6

-- iv. Prevent Lost Update
START TRANSACTION;
SELECT Balance FROM User_account WHERE ID = 101 FOR UPDATE;
UPDATE User_account SET Balance = Balance + 3000 WHERE ID = 101;
COMMIT;
```

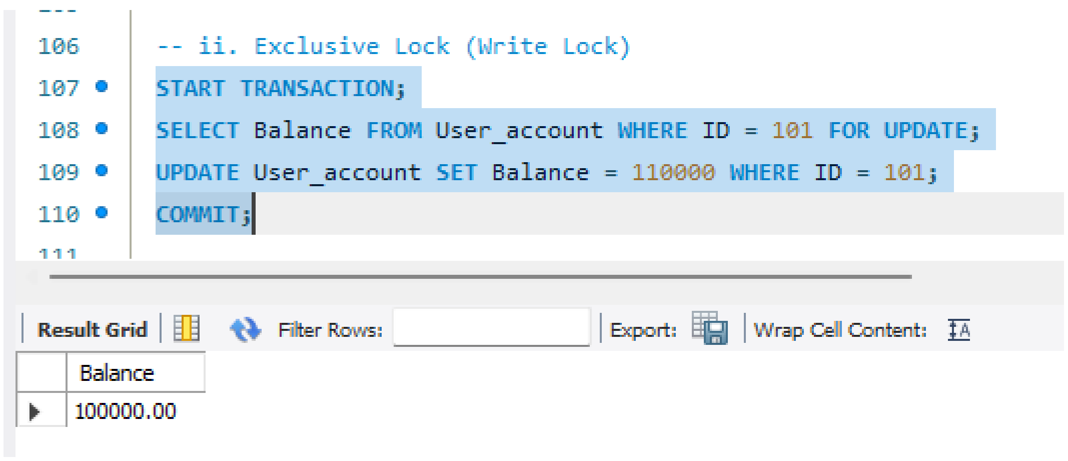
Result:



```
-- Question 6
-- i. Shared Lock (Read Lock)
START TRANSACTION;
SELECT Balance FROM User_account WHERE ID = 101 LOCK IN SHARE MODE;

COMMIT;
```

```
Result Grid | Filter Rows: | Export: | Wrap Cell Content:

Balance
100000.00
```

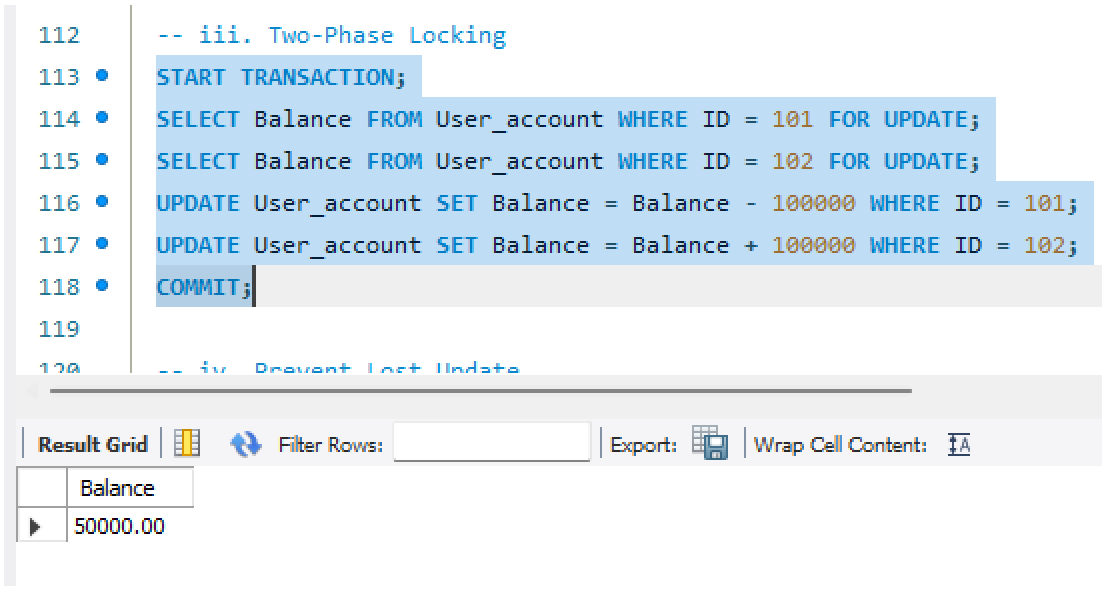
Multiple transactions can read the balance simultaneously. If any transaction tries to UPDATE, it waits until all shared locks are released



```
-- ii. Exclusive Lock (Write Lock)
START TRANSACTION;
SELECT Balance FROM User_account WHERE ID = 101 FOR UPDATE;
UPDATE User_account SET Balance = 110000 WHERE ID = 101;
COMMIT;
```

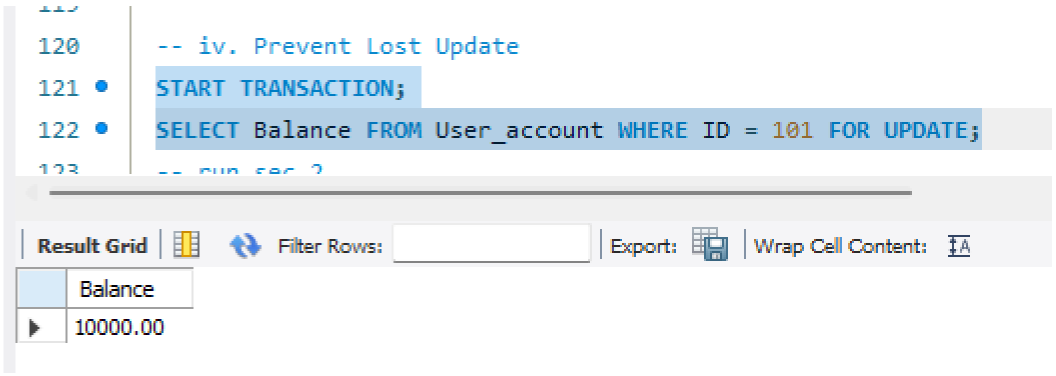
```
Result Grid | Filter Rows: | Export: | Wrap Cell Content:

Balance
100000.00
```

Section 1 has exclusive access to the row. Any other transaction trying to read or modify wait until Section 1 commits

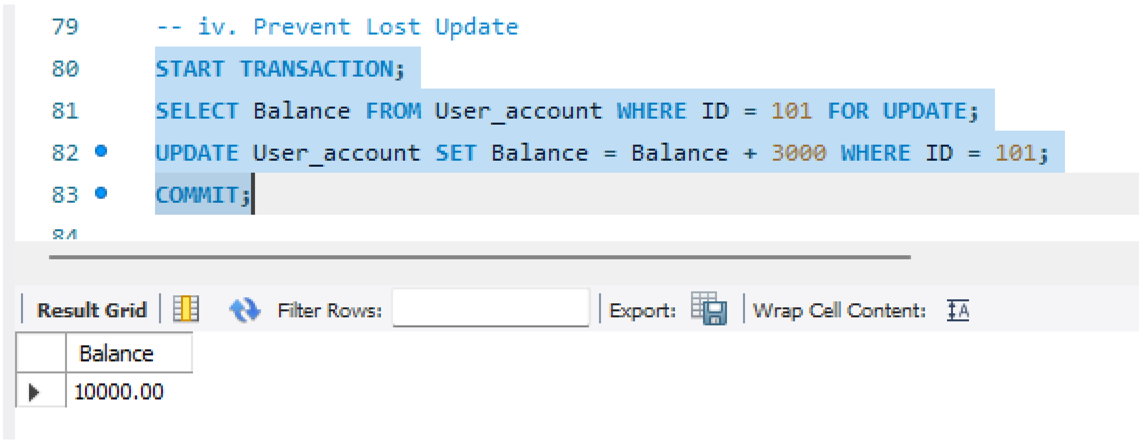

```
-- iii. Two-Phase Locking
START TRANSACTION;
SELECT Balance FROM User_account WHERE ID = 101 FOR UPDATE;
SELECT Balance FROM User_account WHERE ID = 102 FOR UPDATE;
UPDATE User_account SET Balance = Balance - 100000 WHERE ID = 101;
UPDATE User_account SET Balance = Balance + 100000 WHERE ID = 102;
COMMIT;
```

```
Balance
50000.00
```

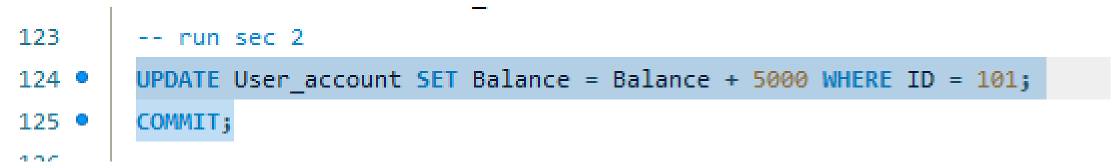
Both accounts are locked before any updates. The transaction completes and both locks are released together at commit.


```
-- iv. Prevent Lost Update
START TRANSACTION;
SELECT Balance FROM User_account WHERE ID = 101 FOR UPDATE;
```

```
Balance
10000.00
```


```
-- iv. Prevent Lost Update
START TRANSACTION;
SELECT Balance FROM User_account WHERE ID = 101 FOR UPDATE;
UPDATE User_account SET Balance = Balance + 3000 WHERE ID = 101;
COMMIT;
```

```
Balance
10000.00
```

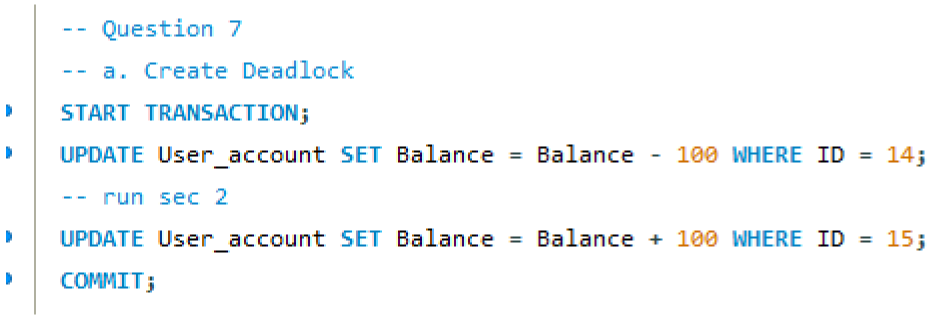

```
-- run sec 2
UPDATE User_account SET Balance = Balance + 5000 WHERE ID = 101;
COMMIT;
```

1. Section 1 locks and reads balance
2. Section 2 tries to lock but must wait
3. Section 1 updates to and commit (releases lock)
4. Section 2 now lock and read
5. Section 2 updates and commits

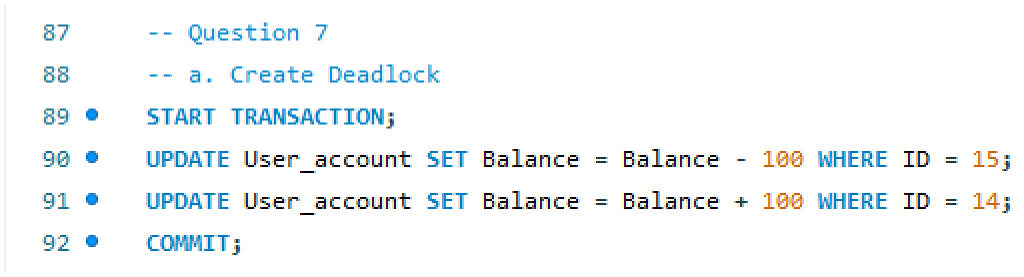
Question 7:

a. Create Deadlock

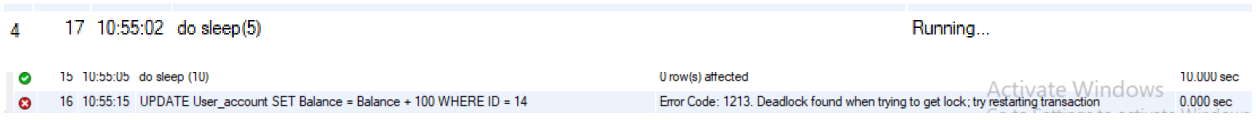
Section 1:


```
-- Question 7
-- a. Create Deadlock
START TRANSACTION;
UPDATE User_account SET Balance = Balance - 100 WHERE ID = 14;
-- run sec 2
UPDATE User_account SET Balance = Balance + 100 WHERE ID = 15;
COMMIT;
```

Section 2:


```
-- Question 7
-- a. Create Deadlock
START TRANSACTION;
UPDATE User_account SET Balance = Balance - 100 WHERE ID = 15;
UPDATE User_account SET Balance = Balance + 100 WHERE ID = 14;
COMMIT;
```

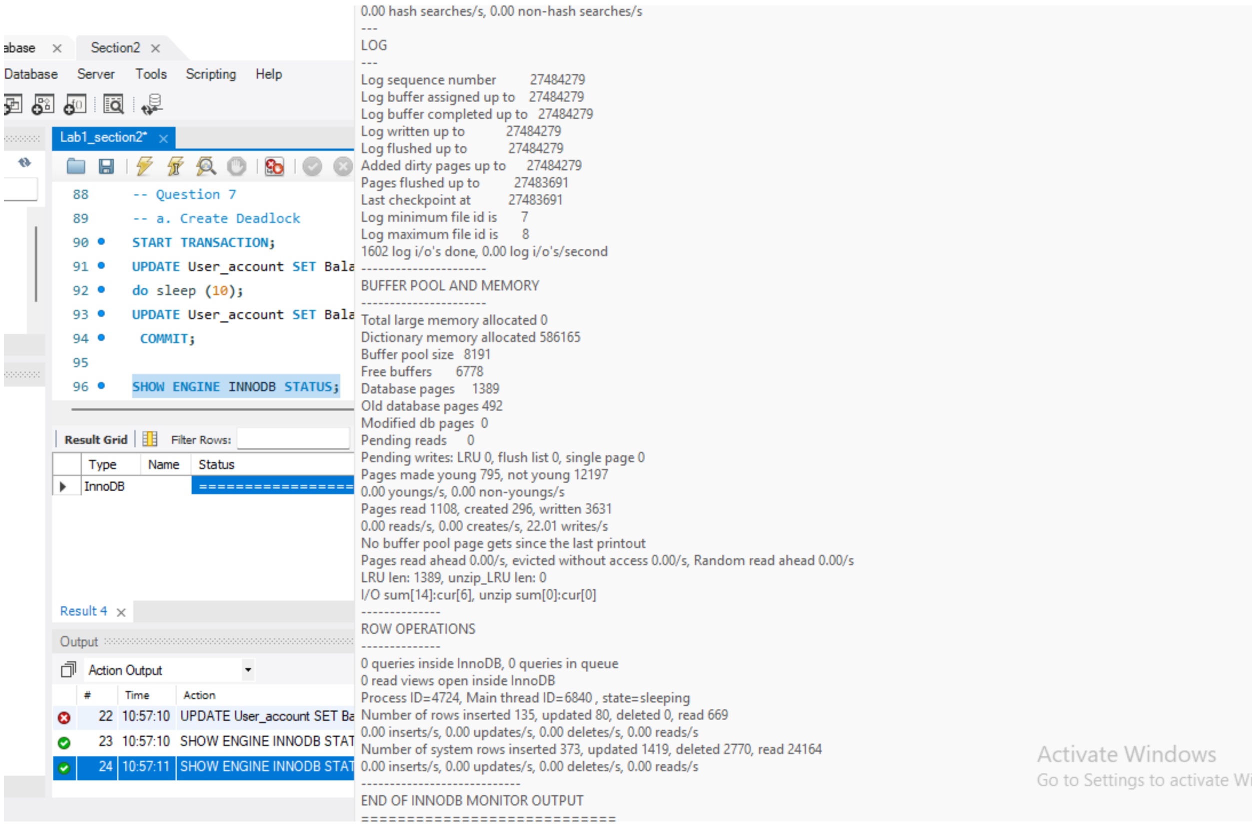
Result:


```
4    17  10:55:02  do sleep(5)                                                    Running...

✓    15  10:55:05  do sleep (10)                                                  0 row(s) affected                                                          10.000 sec
✗    16  10:55:15  UPDATE User_account SET Balance = Balance + 100 WHERE ID = 14  Error Code: 1213. Deadlock found when trying to get lock; try restarting transaction  0.000 sec
```

Database detects the deadlock after a few seconds. The other transaction completes successfully

b. Identify Deadlock

```sql
-- b. Identify Deadlock
SHOW ENGINE INNODB STATUS;
```



```sql
-- Question 7
-- a. Create Deadlock
START TRANSACTION;
UPDATE User_account SET Bala
do sleep (10);
UPDATE User_account SET Bala
 COMMIT;

SHOW ENGINE INNODB STATUS;
```

```
Type    Name   Status
InnoDB         ================
```

```
#   Time      Action
22  10:57:10  UPDATE User_account SET Ba
23  10:57:10  SHOW ENGINE INNODB STAT
24  10:57:11  SHOW ENGINE INNODB STAT
```

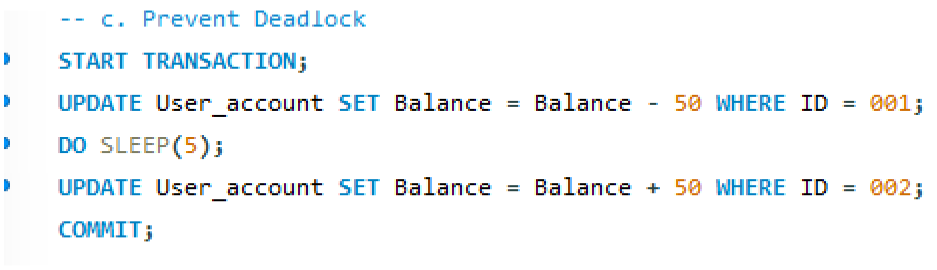
```
0.00 hash searches/s, 0.00 non-hash searches/s
---
LOG
---
Log sequence number        27484279
Log buffer assigned up to   27484279
Log buffer completed up to  27484279
Log written up to          27484279
Log flushed up to          27484279
Added dirty pages up to     27484279
Pages flushed up to        27483691
Last checkpoint at         27483691
Log minimum file id is      7
Log maximum file id is      8
1602 log i/o's done, 0.00 log i/o's/second
----------------------
BUFFER POOL AND MEMORY
----------------------
Total large memory allocated 0
Dictionary memory allocated 586165
Buffer pool size   8191
Free buffers       6778
Database pages     1389
Old database pages 492
Modified db pages  0
Pending reads      0
Pending writes: LRU 0, flush list 0, single page 0
Pages made young 795, not young 12197
0.00 youngs/s, 0.00 non-youngs/s
Pages read 1108, created 296, written 3631
0.00 reads/s, 0.00 creates/s, 22.01 writes/s
No buffer pool page gets since the last printout
Pages read ahead 0.00/s, evicted without access 0.00/s, Random read ahead 0.00/s
LRU len: 1389, unzip_LRU len: 0
I/O sum[14]:cur[6], unzip sum[0]:cur[0]
--------------
ROW OPERATIONS
--------------
0 queries inside InnoDB, 0 queries in queue
0 read views open inside InnoDB
Process ID=4724, Main thread ID=6840 , state=sleeping
Number of rows inserted 135, updated 80, deleted 0, read 669
0.00 inserts/s, 0.00 updates/s, 0.00 deletes/s, 0.00 reads/s
Number of system rows inserted 373, updated 1419, deleted 2770, read 24164
0.00 inserts/s, 0.00 updates/s, 0.00 deletes/s, 0.00 reads/s
----------------------------
END OF INNODB MONITOR OUTPUT
============================
```

c. Prevent Deadlock

Section 1:

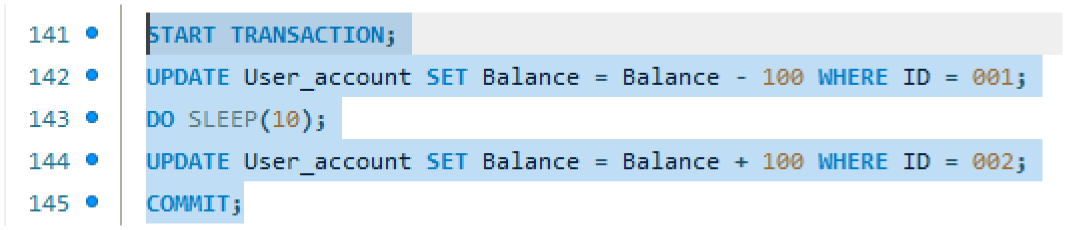
```sql
-- c. Prevent Deadlock
START TRANSACTION;
UPDATE User_account SET Balance = Balance - 100 WHERE ID = 001;
DO SLEEP(10);
UPDATE User_account SET Balance = Balance + 100 WHERE ID = 002;
COMMIT;
```

Section 2:

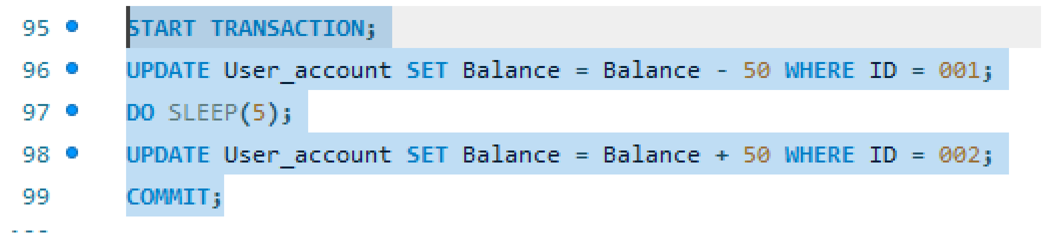


```sql
-- c. Prevent Deadlock
START TRANSACTION;
UPDATE User_account SET Balance = Balance - 50 WHERE ID = 001;
DO SLEEP(5);
UPDATE User_account SET Balance = Balance + 50 WHERE ID = 002;
COMMIT;
```

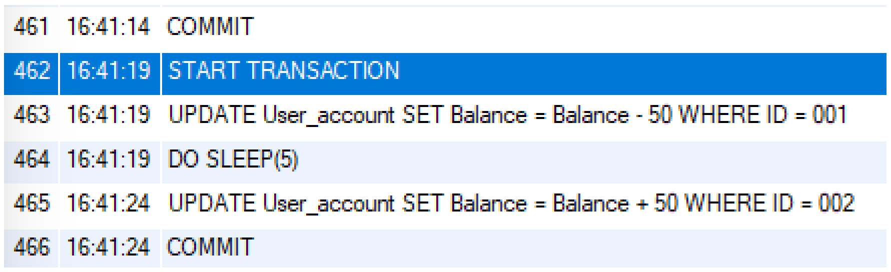
Result:



```sql
START TRANSACTION;
UPDATE User_account SET Balance = Balance - 100 WHERE ID = 001;
DO SLEEP(10);
UPDATE User_account SET Balance = Balance + 100 WHERE ID = 002;
COMMIT;
```



```sql
START TRANSACTION;
UPDATE User_account SET Balance = Balance - 50 WHERE ID = 001;
DO SLEEP(5);
UPDATE User_account SET Balance = Balance + 50 WHERE ID = 002;
COMMIT;
```

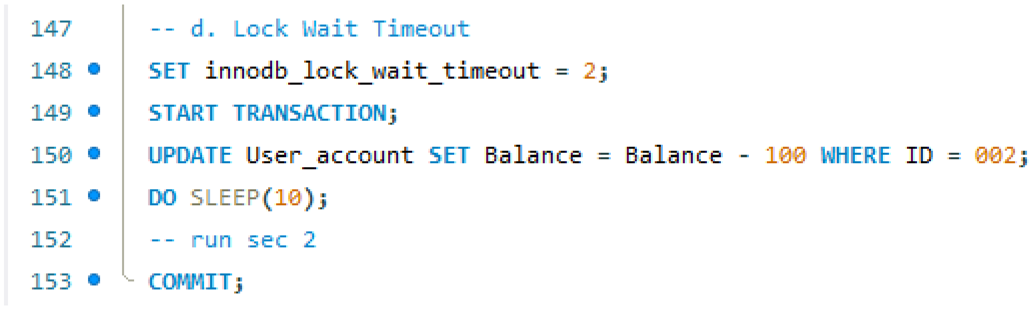


```
461  16:41:14  COMMIT
462  16:41:19  START TRANSACTION
463  16:41:19  UPDATE User_account SET Balance = Balance - 50 WHERE ID = 001
464  16:41:19  DO SLEEP(5)
465  16:41:24  UPDATE User_account SET Balance = Balance + 50 WHERE ID = 002
466  16:41:24  COMMIT
```

Both transactions complete successfully without deadlock. Section 2 waits for Section 1 to complete, then proceeds. The consistent lock ordering prevents circular wait

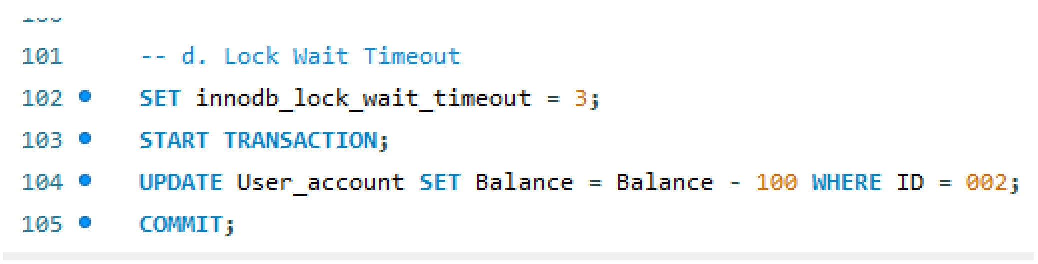
d. Lock Wait Timeout

Section 1:



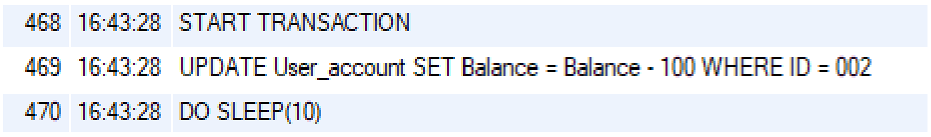
```sql
-- d. Lock Wait Timeout
SET innodb_lock_wait_timeout = 2;
START TRANSACTION;
UPDATE User_account SET Balance = Balance - 100 WHERE ID = 002;
DO SLEEP(10);
-- run sec 2
COMMIT;
```

Section 2:



```sql
-- d. Lock Wait Timeout
SET innodb_lock_wait_timeout = 3;
START TRANSACTION;
UPDATE User_account SET Balance = Balance - 100 WHERE ID = 002;
COMMIT;
```

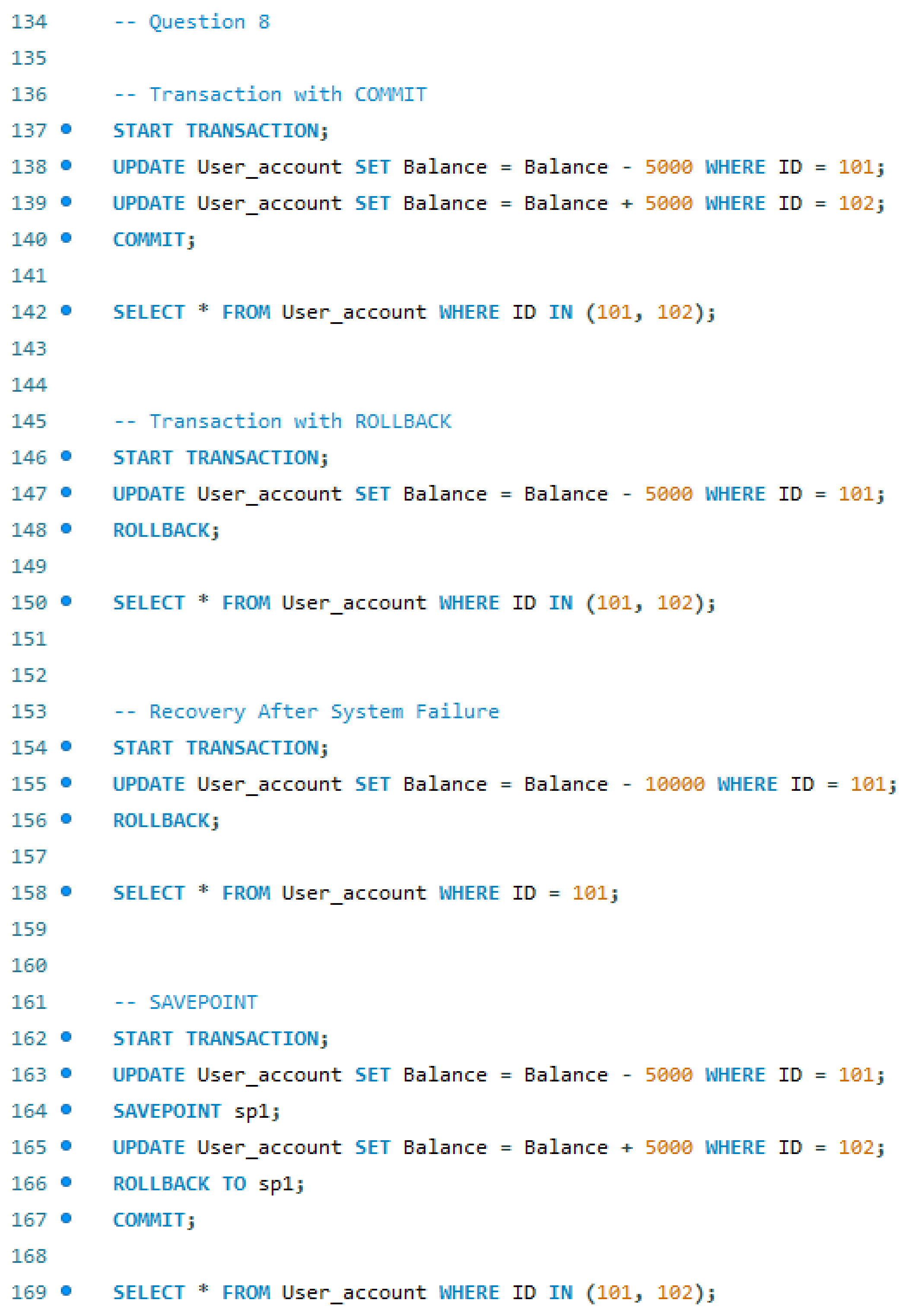
Result:



```
468  16:43:28  START TRANSACTION
469  16:43:28  UPDATE User_account SET Balance = Balance - 100 WHERE ID = 002
470  16:43:28  DO SLEEP(10)
```

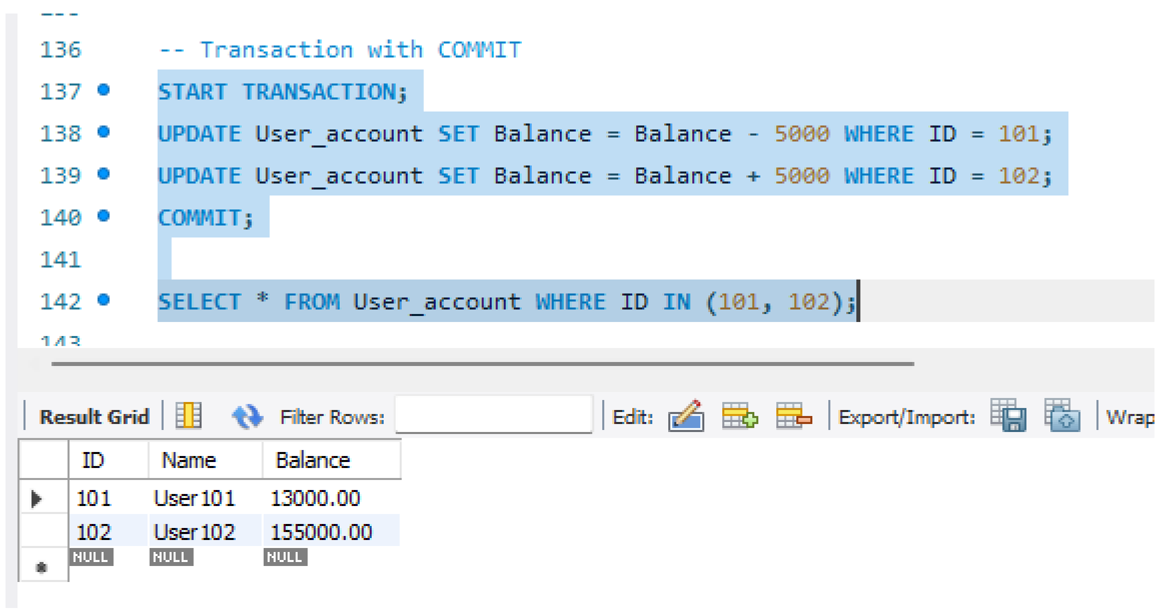
Section 2 waits for 3 seconds, then receives “Lock wait timeout exceeded” error. The transaction is not commit. Section 1 complete after its 10 second sleep

Question 8:



```sql
-- Question 8

-- Transaction with COMMIT
START TRANSACTION;
UPDATE User_account SET Balance = Balance - 5000 WHERE ID = 101;
UPDATE User_account SET Balance = Balance + 5000 WHERE ID = 102;
COMMIT;

SELECT * FROM User_account WHERE ID IN (101, 102);


-- Transaction with ROLLBACK
START TRANSACTION;
UPDATE User_account SET Balance = Balance - 5000 WHERE ID = 101;
ROLLBACK;

SELECT * FROM User_account WHERE ID IN (101, 102);


-- Recovery After System Failure
START TRANSACTION;
UPDATE User_account SET Balance = Balance - 10000 WHERE ID = 101;
ROLLBACK;

SELECT * FROM User_account WHERE ID = 101;


-- SAVEPOINT
START TRANSACTION;
UPDATE User_account SET Balance = Balance - 5000 WHERE ID = 101;
SAVEPOINT sp1;
UPDATE User_account SET Balance = Balance + 5000 WHERE ID = 102;
ROLLBACK TO sp1;
COMMIT;

SELECT * FROM User_account WHERE ID IN (101, 102);
```

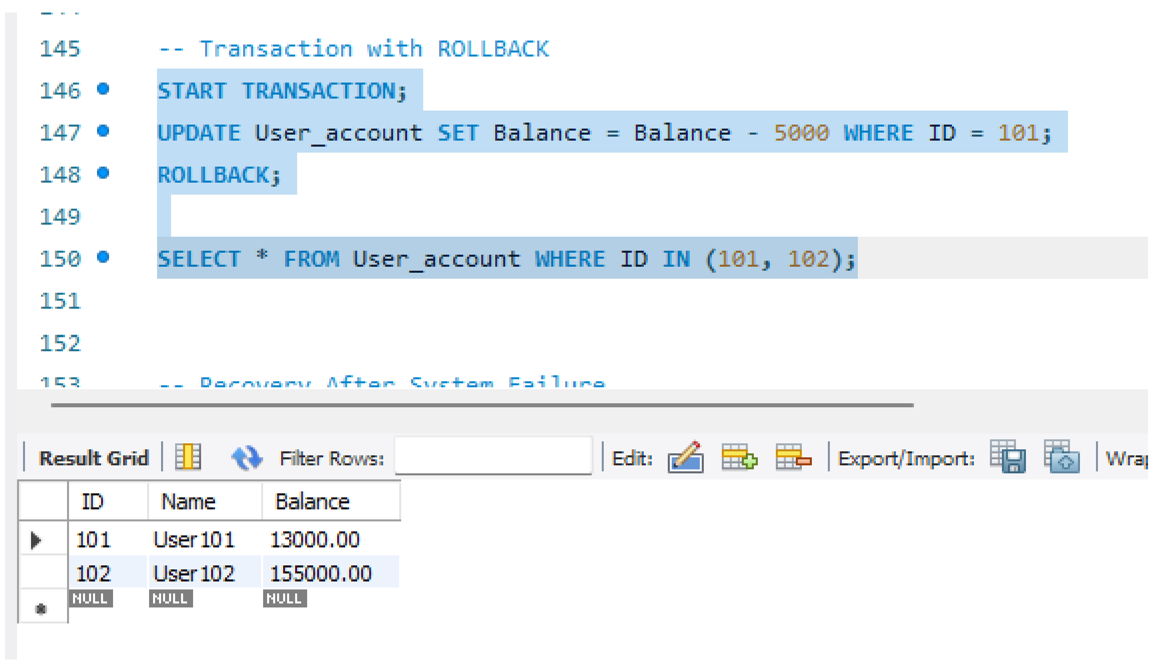
Result:



```
-- Transaction with COMMIT
START TRANSACTION;
UPDATE User_account SET Balance = Balance - 5000 WHERE ID = 101;
UPDATE User_account SET Balance = Balance + 5000 WHERE ID = 102;
COMMIT;

SELECT * FROM User_account WHERE ID IN (101, 102);
```

ID	Name	Balance
101	User101	13000.00
102	User102	155000.00
NULL	NULL	NULL

Account 101 balance decreases by 5000 and Account 102 balance increases by 5000.



```
-- Transaction with ROLLBACK
START TRANSACTION;
UPDATE User_account SET Balance = Balance - 5000 WHERE ID = 101;
ROLLBACK;

SELECT * FROM User_account WHERE ID IN (101, 102);


-- Recovery After System Failure
```

ID	Name	Balance
101	User101	13000.00
102	User102	155000.00
NULL	NULL	NULL

1. start the transaction

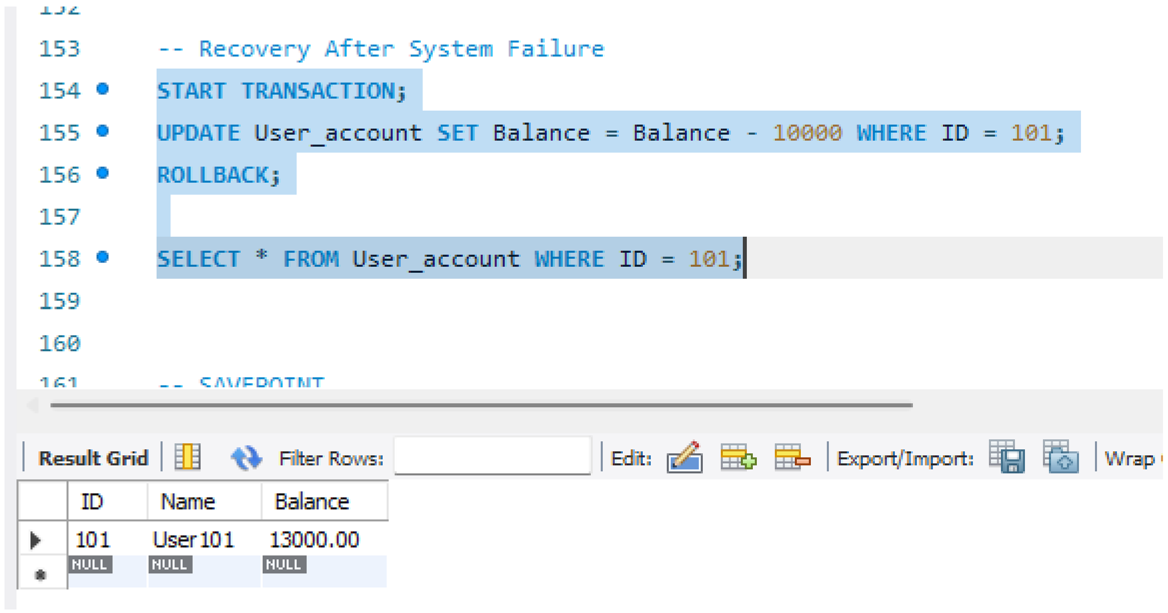
2. Update is performed

3. rollback cancel the update

4. Database return to original state

5. Transaction end

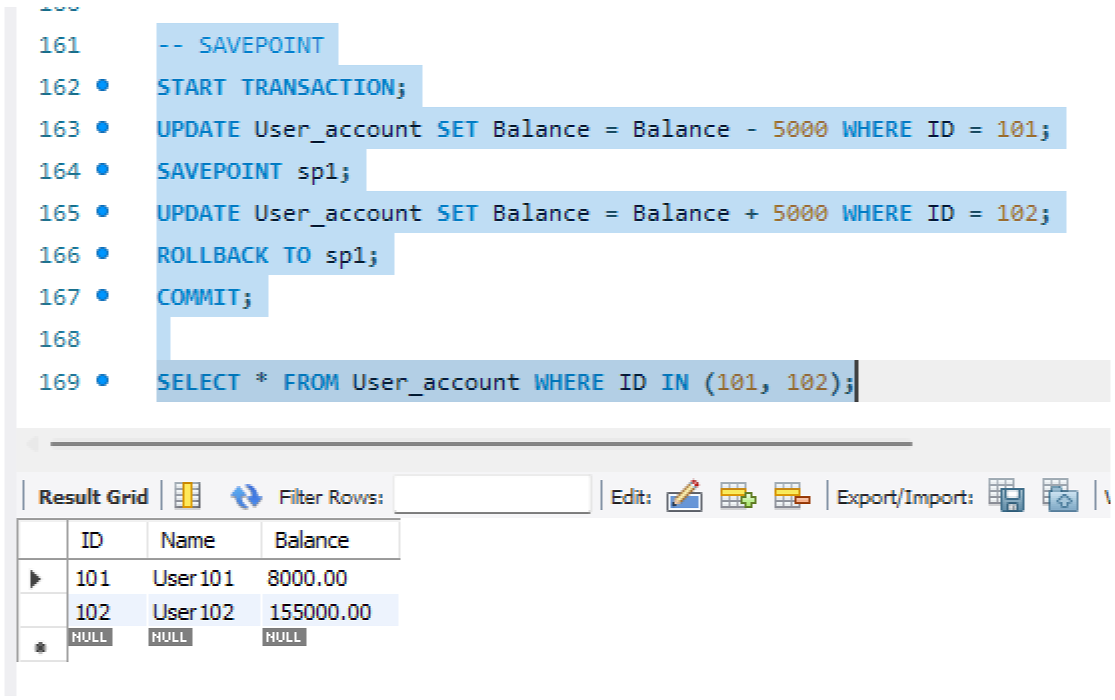
Even though the UPDATE statement ran, the ROLLBACK cancel it. The final SELECT shows the balance unchanged.



```
-- Recovery After System Failure
START TRANSACTION;
UPDATE User_account SET Balance = Balance - 10000 WHERE ID = 101;
ROLLBACK;

SELECT * FROM User_account WHERE ID = 101;


-- SAVEPOINT
```

ID	Name	Balance
101	User101	13000.00
NULL	NULL	NULL

Account 101 retain original balance. The uncommitted update is undone



```
-- SAVEPOINT
START TRANSACTION;
UPDATE User_account SET Balance = Balance - 5000 WHERE ID = 101;
SAVEPOINT sp1;
UPDATE User_account SET Balance = Balance + 5000 WHERE ID = 102;
ROLLBACK TO sp1;
COMMIT;

SELECT * FROM User_account WHERE ID IN (101, 102);
```

ID	Name	Balance
101	User101	8000.00
102	User102	155000.00
NULL	NULL	NULL

1. Update Account 101

2. Create savepoint “sp1”

3. Update Account 102

4. rollback to sp1

5. COMMIT saves the changes before sp1

Account 101 balance is reduced by 5000. Account 102 balance is unchanged.