# Concurrency Concerns

# In This Lab

1. Dirty Read Concurrency Problem in SQL Server
2. Last Update Concurrency Problem
3. Non-Repeatable Read Concurrency Problem
4. Phantom Read Concurrency Problem in SQL Server

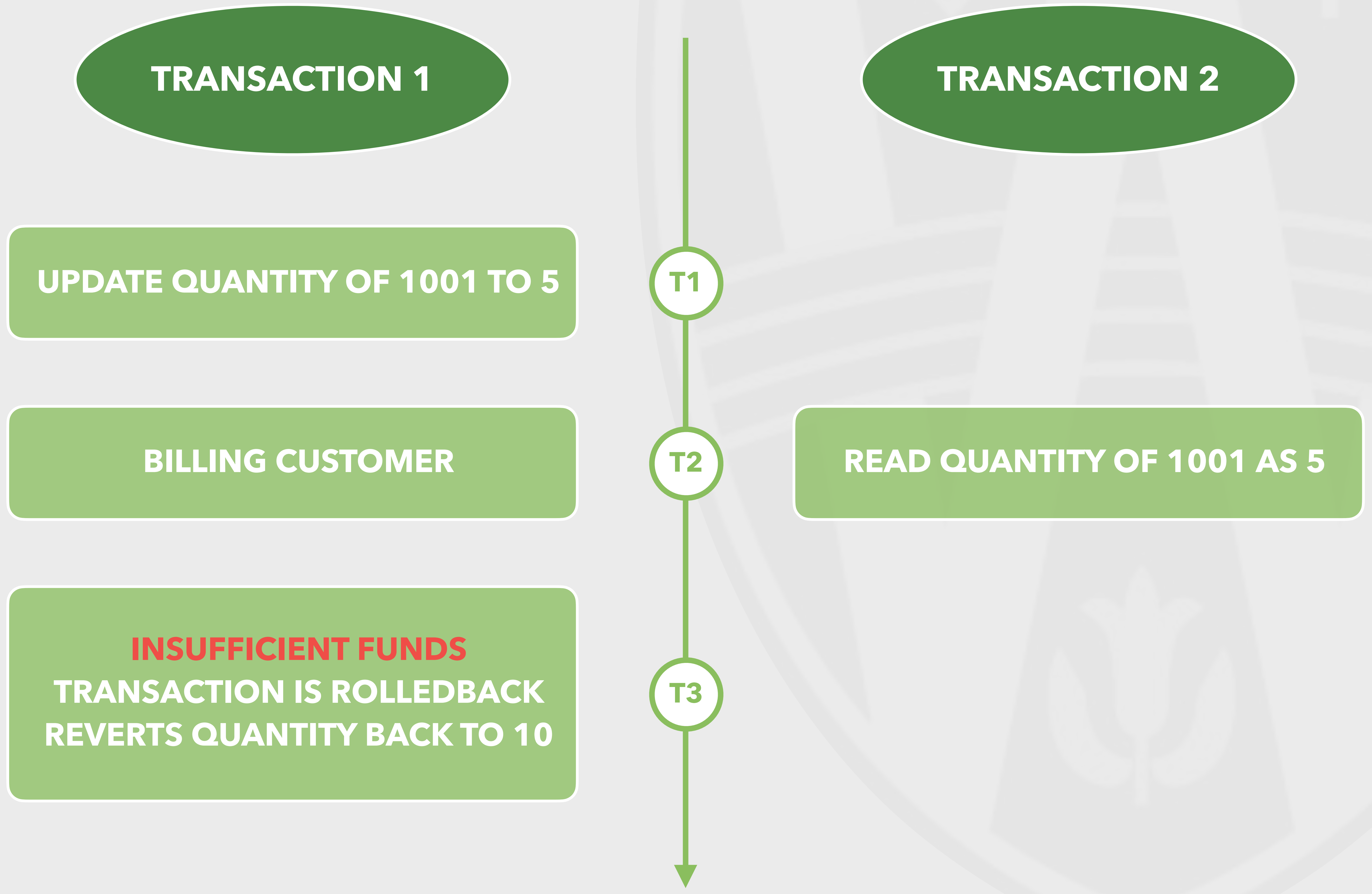
- Assignment

## 1 Dirty Read Concurrency Problem

# Dirty Read Concurrency Problem

- When one transaction is allowed to read the uncommitted data of another transaction.
- Example: Transactions 1 and 2 are going to work with the same data

| id | Name | Quantity |
|---|---|---|
| 1001 | Mobile | ~~10~~ ~~5~~ 10 |
| 1002 | Tablet | 20 |
| 1003 | Laptop | 30 |


TRANSACTION 1
TRANSACTION 2
UPDATE QUANTITY OF 1001 TO 5
T1
BILLING CUSTOMER
T2
READ QUANTITY OF 1001 AS 5
INSUFFICIENT FUNDS
TRANSACTION IS ROLLEDBACK
REVERTS QUANTITY BACK TO 10
T3

# Dirty Read Concurrency Problem (Example)

Create Table

```
CREATE TABLE Products
(
    Id INT PRIMARY KEY,
    Name VARCHAR(100),
    Quantity INT
)
Go

-- Insert test data into Products table
INSERT INTO Products values (1001, 'Mobile', 10)
INSERT INTO Products values (1002, 'Tablet', 20)
INSERT INTO Products values (1003, 'Laptop', 30)
```

Transaction 1

```
BEGIN TRANSACTION
  UPDATE Products SET Quantity = 5 WHERE Id=1001

  -- Billing the customer
  Waitfor Delay '00:00:15'
  -- Insufficient Funds. Rollback transaction

ROLLBACK TRANSACTION
```

Transaction 2

```
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED
SELECT * FROM Products WHERE Id=1001
```

Intentionally delay the execution to 15 seconds by using `Waitfor Delay` statement

By default SQL Server will not allow reading the uncommitted data of one transaction. So, to understand the Dirty Read Concurrency Problem here we set the transaction isolation level to **Read Uncommitted.**

**Read Uncommitted** transaction isolation level is the only Transaction Isolation Level provided by SQL Server which has the Dirty Read Concurrency Problem. The **Read Uncommitted** transaction isolation level is the least restrictive isolation level among all the isolation levels provided by SQL Server. When we use this transaction isolation level then it is possible to read the uncommitted or dirty data.

# Dirty Read Concurrency Problem (Solution)

- **Option 1:**
  - Avoid using **Read Uncommitted** transaction isolation level !
  - **Get back to Read Committed:**

```
SET TRANSACTION ISOLATION LEVEL READ COMMITTED
SELECT * FROM Products WHERE Id=1001
```

  - Now run the first transaction and then immediately run the second transaction. You will see that **until the first is not completed, you will not get the result in the second transaction.** Once the first transaction execution is completed, then you will get the data in the second transaction and this time you will not get the uncommitted data rather you will get the committed data that exist in the database.

- **Option 2:**
  - Another option provided by SQL Server to read the dirty data is by using the **NOLOCK** table hint option. The below query is equivalent to the query that we wrote in Transaction 2.

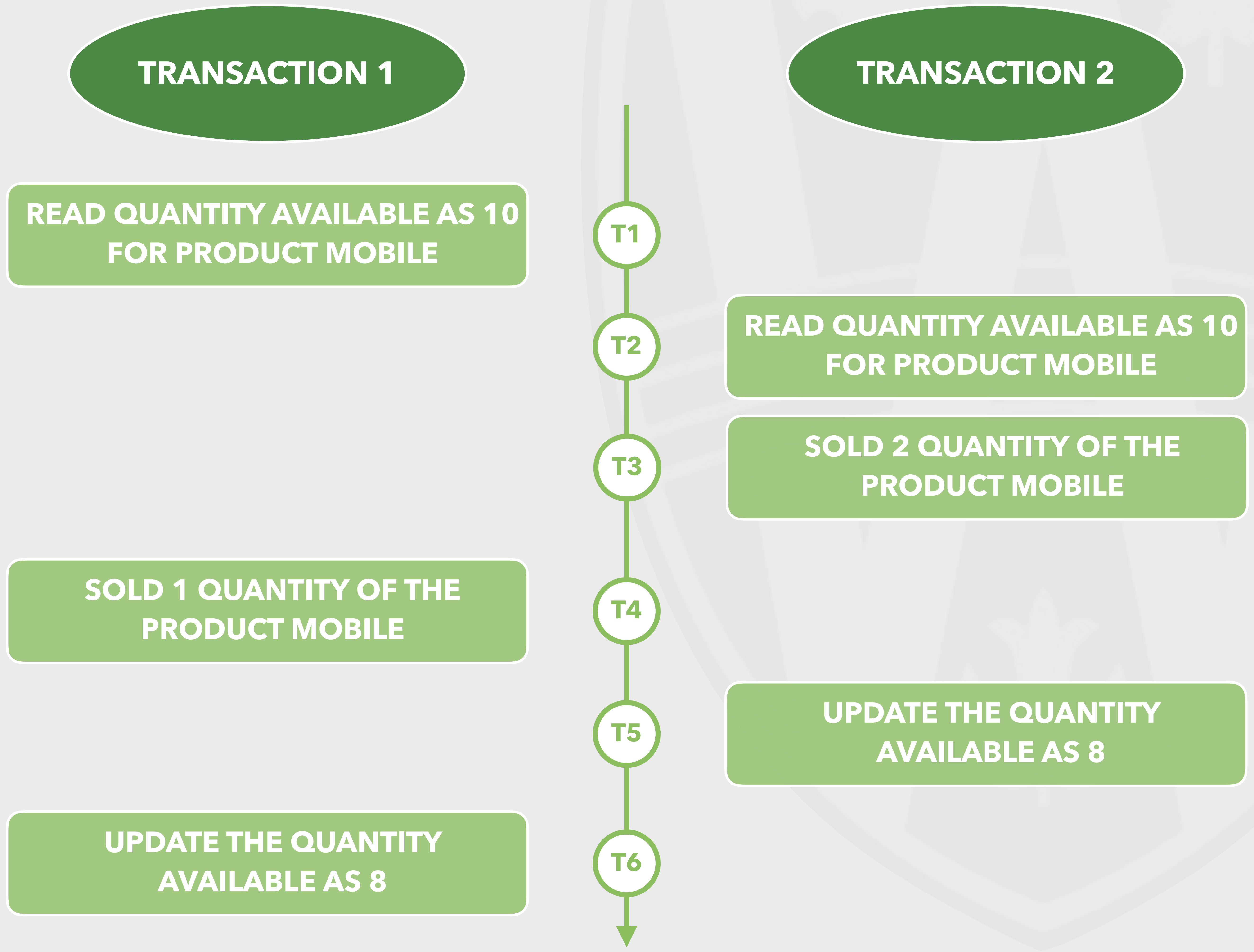
```
SELECT * FROM Products (NOLOCK) WHERE Id=1001
```

## 2 Lost Update Concurrency Problem

# Lost Update Concurrency Problem

- When two or more transactions are allowed to read and update the same data.
- Example: Transactions 1 and 2 are going to work with the same data

| id | Name | Quantity |
|---|---|---|
| 1001 | Mobile | 10 |
| 1002 | Tablet | 20 |
| 1003 | Laptop | 30 |


TRANSACTION 1
TRANSACTION 2
T1
READ QUANTITY AVAILABLE AS 10 FOR PRODUCT MOBILE
T2
READ QUANTITY AVAILABLE AS 10 FOR PRODUCT MOBILE
T3
SOLD 2 QUANTITY OF THE PRODUCT MOBILE
T4
SOLD 1 QUANTITY OF THE PRODUCT MOBILE
T5
UPDATE THE QUANTITY AVAILABLE AS 8
T6
UPDATE THE QUANTITY AVAILABLE AS 8

# Lost Update Concurrency Problem (Example)

```
-- Re-use the code for creating Products table

BEGIN TRANSACTION
  DECLARE @QunatityAvailable int
  SELECT @QunatityAvailable = Quantity FROM Products WHERE Id=1001

  -- Transaction takes 10 seconds
  WAITFOR DELAY '00:00:10'

  SET @QunatityAvailable = @QunatityAvailable - 1
  UPDATE Products SET Quantity = @QunatityAvailable  WHERE Id=1001
  Print @QunatityAvailable

COMMIT TRANSACTION
```

Transaction 1

```
BEGIN TRANSACTION
  DECLARE @QunatityAvailable int
  SELECT @QunatityAvailable = Quantity FROM Products WHERE Id=1001

  SET @QunatityAvailable = @QunatityAvailable - 2
  UPDATE Products SET Quantity = @QunatityAvailable WHERE Id=1001
  Print @QunatityAvailable

COMMIT TRANSACTION
```

Transaction 2

```
SELECT * FROM Products WHERE Id=1001
```

At the end of both the transactions, the Quantity of the Mobile should be 7 in the database, but we have a value of 9. This is because **Transaction 1 silently overwrites the update which is made by Transaction 2.**

You can see the final table status with this query.

# Lost Update Concurrency Problem (Solution)

- Use either **Repeatable Read** transaction isolation level or any other higher isolation level such as **Snapshot** or **Serializable**.

Transaction 1

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ
BEGIN TRANSACTION
SELECT Quantity FROM Products WHERE Id = 1001

-- Do Some work
WAITFOR DELAY '00:00:15'
SELECT Quantity FROM Products WHERE Id = 1001
COMMIT TRANSACTION
```

Transaction 2

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ
UPDATE Products SET Quantity = 5 WHERE Id = 1001
```

Transaction 2 is **blocked** until Transaction 1 completes, and at the end of Transaction 1, both the reads get the same value for the quantity of the same product mobile.

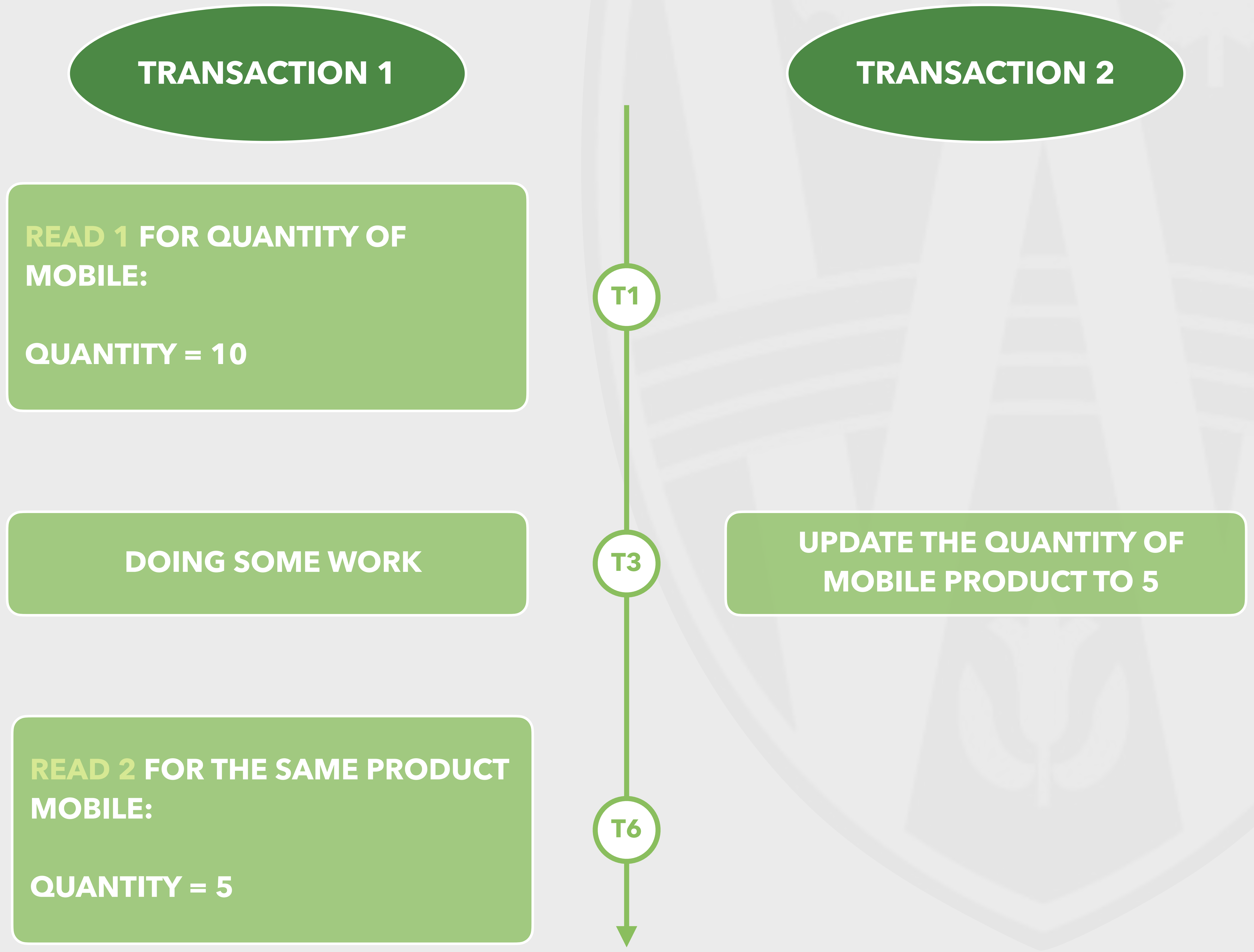
3

## Non-Repeatable Read Concurrency Problem

# Non-Repeatable Read Concurrency Problem

- When one transaction reads the same data twice while another transaction updates that data in between the first and second read of the first transaction.
- Example: Transactions 1 and 2 are going to work with the same data

| id | Name | Quantity |
|---|---|---|
| 1001 | Mobile | 10 |
| 1002 | Tablet | 20 |
| 1003 | Laptop | 30 |


TRANSACTION 1
TRANSACTION 2
READ 1 FOR QUANTITY OF MOBILE:
QUANTITY = 10
T1
DOING SOME WORK
T3
UPDATE THE QUANTITY OF MOBILE PRODUCT TO 5
READ 2 FOR THE SAME PRODUCT MOBILE:
QUANTITY = 5
T6

# Non-Repeatable Read Concurrency Problem (Example)

```
-- Re-use the code for creating Products table



SET TRANSACTION ISOLATION LEVEL READ COMMITTED
BEGIN TRANSACTION
SELECT Quantity FROM Products WHERE Id = 1001

-- Do Some work

WAITFOR DELAY '00:00:15'
SELECT Quantity FROM Products WHERE Id = 1001
COMMIT TRANSACTION
```

Transaction 1

```
SET TRANSACTION ISOLATION LEVEL READ COMMITTED
UPDATE Products SET Quantity = 5 WHERE Id = 1001
```

Transaction 2

**READ COMMITTED** and **READ UNCOMMITTED** transaction isolation level produces the Non-Repeatable Read Concurrency Problem.

## Non-Repeatable Read Concurrency Problem (Solution)

- Use either **Repeatable Read** transaction isolation level or any other higher isolation level such as **Snapshot** or **Serializable**.

Transaction 1

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ
BEGIN TRANSACTION
SELECT Quantity FROM Products WHERE Id = 1001

-- Do Some work
WAITFOR DELAY '00:00:15'
SELECT Quantity FROM Products WHERE Id = 1001
COMMIT TRANSACTION
```

Transaction 2

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ
UPDATE Products SET Quantity = 5 WHERE Id = 1001
```

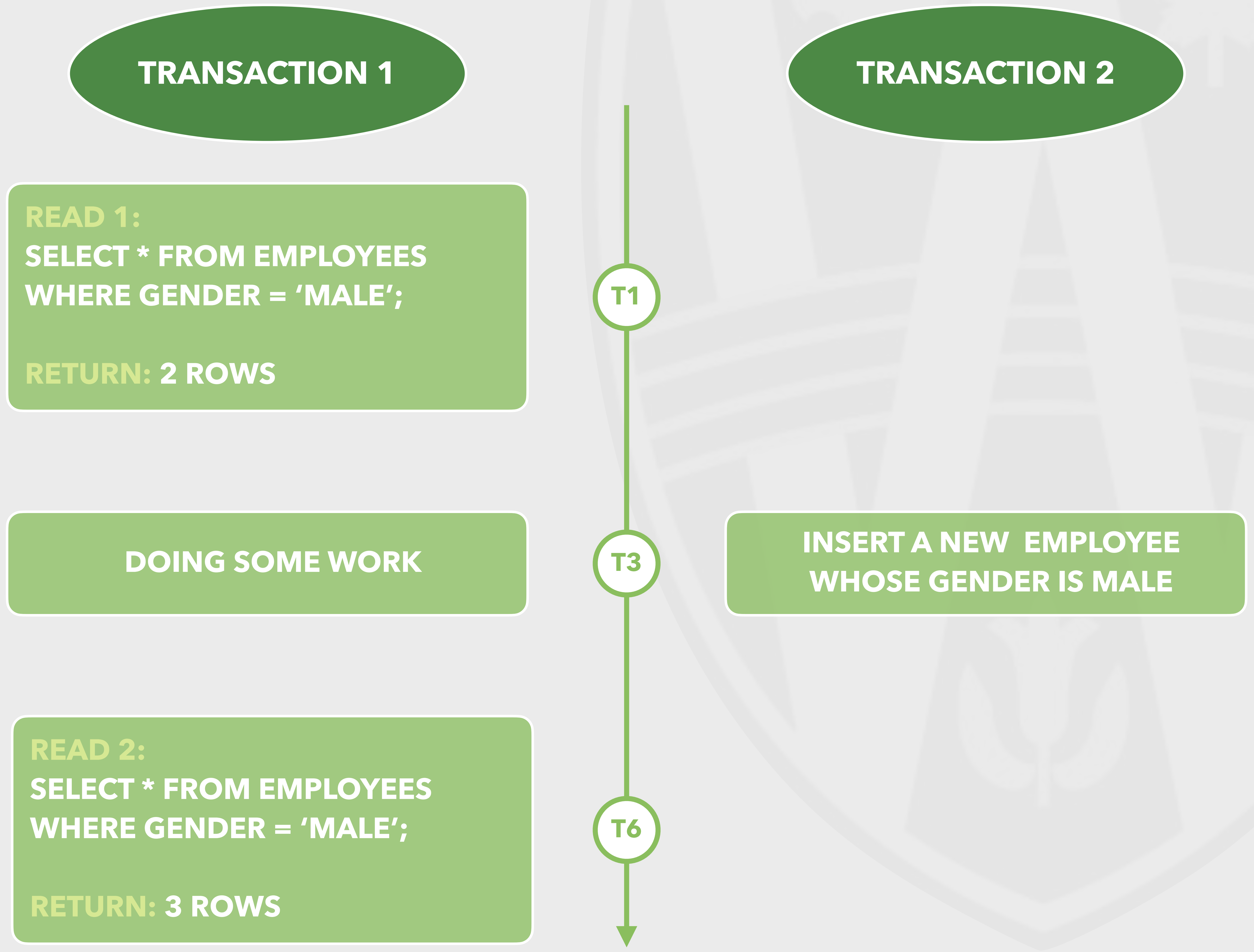
This will ensure that the data that Transaction 1 has read will be prevented from being updated or deleted elsewhere.

## 4 Phantom Read Concurrency Problem

# Phantom Read Concurrency Problem

- When one transaction executes a query twice and it gets a different number of rows in the result set each time.
- Example: Transactions 1 and 2 are going to work with the same data

| id | Name | Quantity |
|---|---|---|
| 1001 | Anurag | Male |
| 1002 | Priyanka | Female |
| 1003 | Pranaya | Male |
| 1004 | Hina | Female |


TRANSACTION 1
TRANSACTION 2
READ 1:
SELECT * FROM EMPLOYEES WHERE GENDER = 'MALE';
RETURN: 2 ROWS
T1
DOING SOME WORK
T3
INSERT A NEW EMPLOYEE WHOSE GENDER IS MALE
READ 2:
SELECT * FROM EMPLOYEES WHERE GENDER = 'MALE';
RETURN: 3 ROWS
T6

# Phantom Read Concurrency Problem (Example)

```
CREATE TABLE Employees
 (
     Id INT PRIMARY KEY,
     Name VARCHAR(100),
     Gender VARCHAR(10)
 )
 Go
 -- Insert some dummy data
 INSERT INTO  Employees VALUES(1001,'Anurag', 'Male')
 INSERT INTO  Employees VALUES(1002,'Priyanka', 'Female')
 INSERT INTO  Employees VALUES(1003,'Pranaya', 'Male')
 INSERT INTO  Employees VALUES(1004,'Hina', ‘Female')
```

Create Table

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ
BEGIN TRANSACTION
SELECT * FROM Employees where Gender = 'Male'
-- Do Some work
WAITFOR DELAY '00:00:10'
SELECT * FROM Employees where Gender = 'Male'
COMMIT TRANSACTION
```

Transaction 1

The **Read Committed**, **Read Uncommitted**, and **Repeatable Read** transaction isolation levels causes Phantom Read Concurrency Problem in SQL Server.

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ
BEGIN TRANSACTION
INSERT into Employees VALUES(1005, 'Sambit', 'Male')
COMMIT TRANSACTION
```

Transaction 2

## Phantom Read Concurrency Problem (Solution)

- Use the **Serializable** or **Snapshot** transaction isolation level to solve the Phantom Read Concurrency Problem in SQL Server.

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE
BEGIN TRANSACTION
SELECT * FROM Employees where Gender = 'Male'
-- Do Some work
WAITFOR DELAY '00:00:10'
SELECT * FROM Employees where Gender = 'Male'
COMMIT TRANSACTION
```

Transaction 1

## Transaction Isolation Levels

| Isolation Level | Lost Update | Dirty Read | Non-Repeatable Reads | Phantom Read |
|---|---|---|---|---|
| Read Uncommitted | Don’t Occur | May Occur | May Occur | May Occur |
| Read Committed | Don’t Occur | Don’t Occur | May Occur | May Occur |
| Repeatable Read | Don’t Occur | Don’t Occur | Don’t Occur | May Occur |
| Serializable | Don’t Occur | Don’t Occur | Don’t Occur | Don’t Occur |