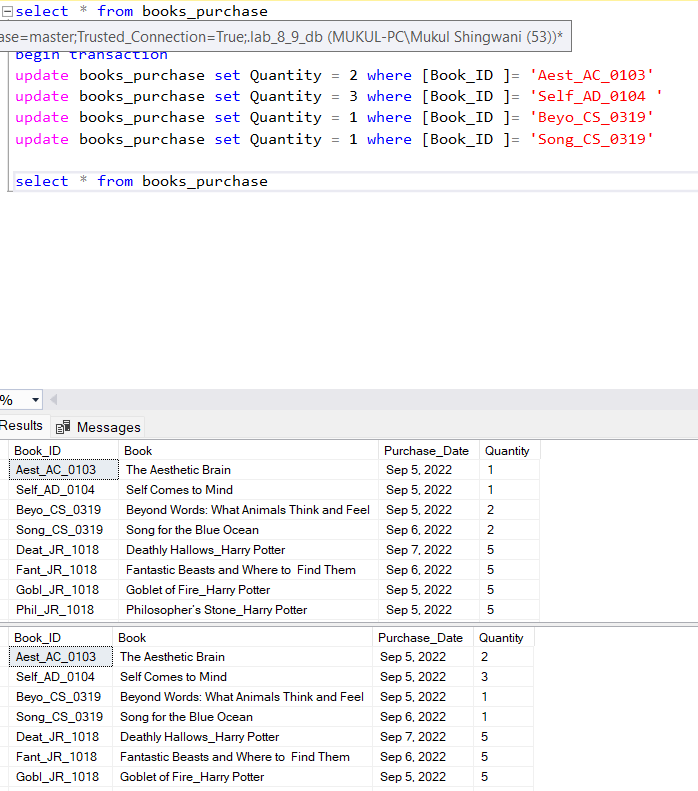
**Data Engineering**

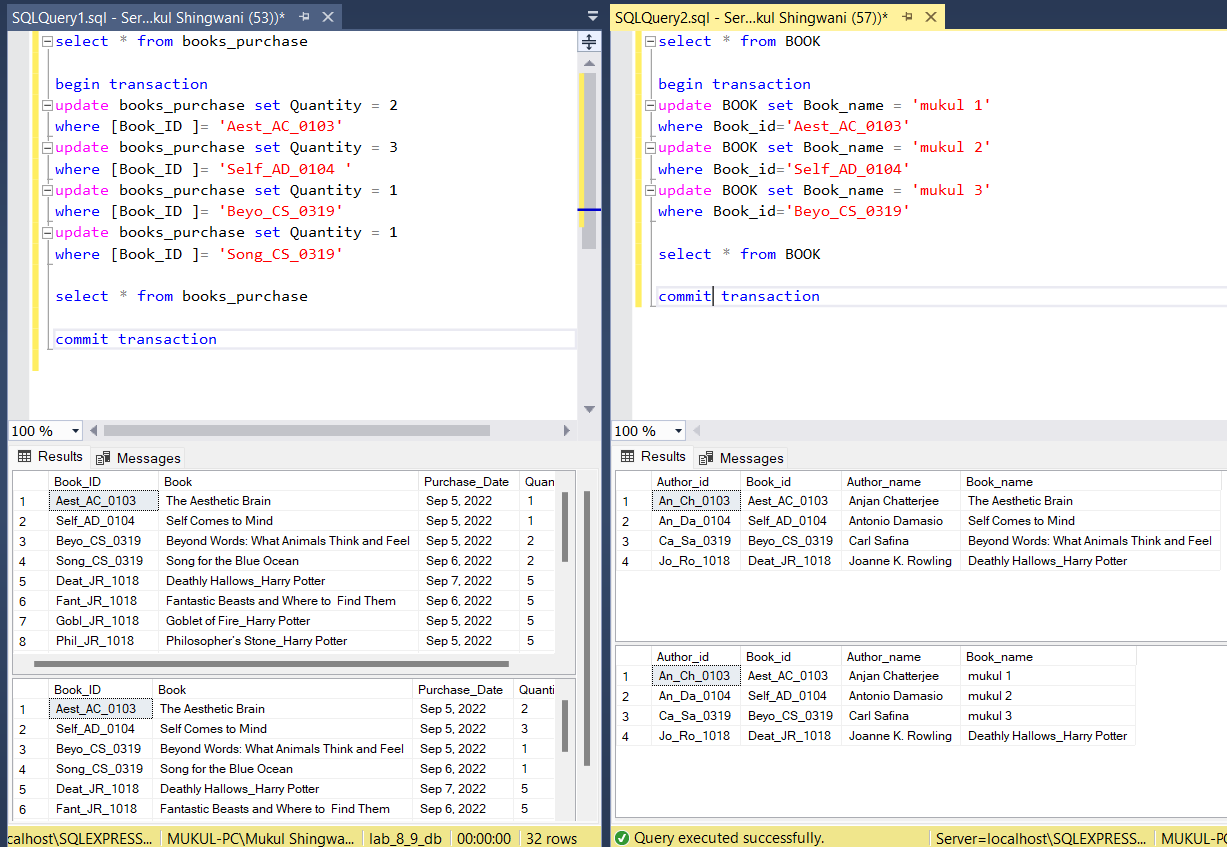
**Lab 8 and 9**

**Mukul Shingwani (B20AI023)**

—---------------------------------------------------------------------------------------------

**Q1) A)**



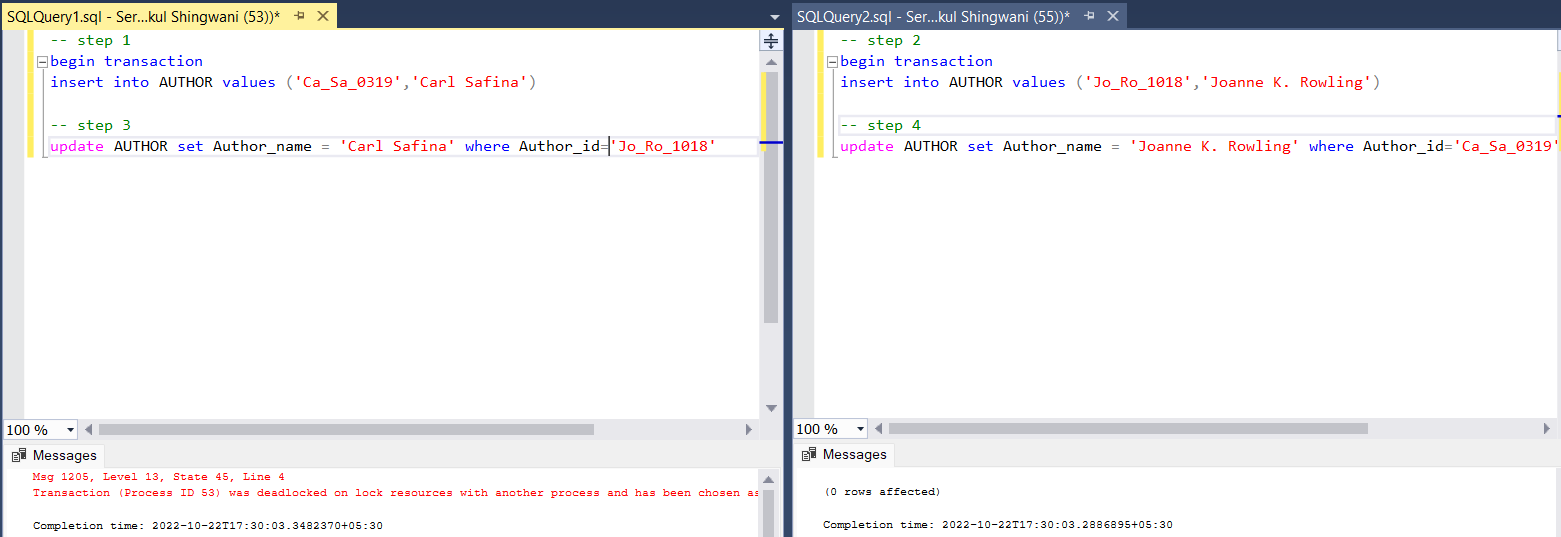


From these images we can see, that multiple updates during one transaction, and multiple transaction at the same time are accessing the same database, thus showing the property of multi-threaded and multi-transaction handling

**Q1) B) i)**

Following were the steps of execution:-

* Insertion in Author was done in query 1 (txn1)
* Insertion in Author was done in query 2 (txn2)
* Tuple in AUTHOR was attempted to update where author\_id = ‘Jo\_Ro\_1018‘ in query 1
* Tuple in AUTHOR was attempted to update where author\_id = ‘Ca\_Sa\_0319‘ in query 2



-- Session 1: Step 1.

BEGIN TRANSACTION

INSERT INTO AUTHOR

VALUES ('Ca\_Sa\_0319','Carl Safina')

-- ----------------------

-- Session 2: Step 2.

BEGIN TRAN

INSERT INTO AUTHOR

VALUES ('Jo\_Ro\_1018','Joanne K. Rowling')

-- ----------------------

-- Session 1: Step 3.

update AUTHOR set Author\_name = 'Carl Safina' where Author\_id='Jo\_Ro\_1018'

-- ----------------------

-- Session 2: Step 4.

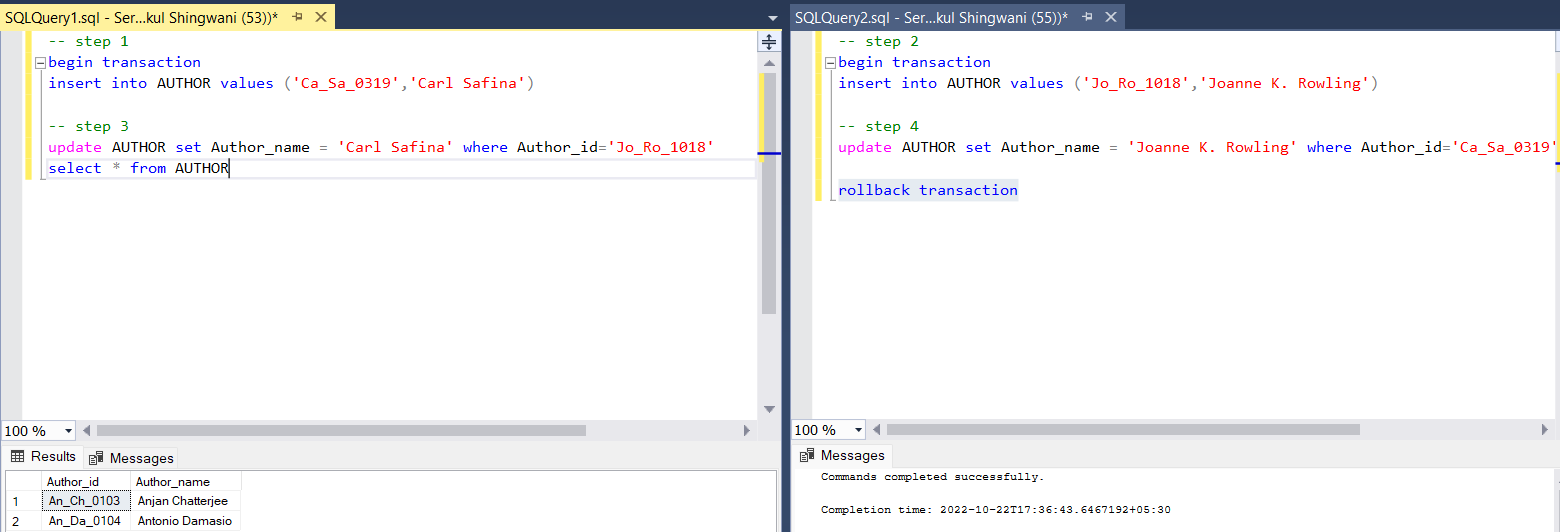
update AUTHOR set Author\_name = 'Joanne K. Rowling' where Author\_id='Ca\_Sa\_0319'

**Q1) B) ii)**

I would have rolled back the second transaction i.e session 2 since it’s first execution started later than query 1, so rolling it back will make query 1 run, this could be thought of as a FIFO (first-in-first-out) approach, since query 1 came first let it be completed first or given priority over other transactions.

**Q1) B) iii)**

As can been seen, after rolling back session 2, session 1 which was the victim of deadlock, successfully executed its operations.



**Q1) B) iv)**

This code is not conflict serializable, since it cannot be resolved into a serial schedule i.e. one of its conflict equivalent and neither its precedence graph is acyclic.

T1 T2

W(A)

W(A)

W(A)

W(A)

**Precedence graph**

T1

T2

Cyclic!!

**Q2A)**

Following were the steps of execution :-

* Insertion in books\_purchase was done in query 1 (txn1)
* Insertion in books\_purchase was done in query 2 (txn2)
* Tuple in books\_purchase was attempted to update where book\_id = ‘pqrs\_ZZ\_10‘in query 1
* Tuple in books\_purchase was attempted to update where book\_id = ‘abcd\_YY\_12 ‘in query 2

select \* from books\_purchase

-- T1 🡪 step 1

begin transaction

insert into books\_purchase

values ('abcd\_YY\_12','MUKUL 1','sep 7, 2022',3)

-----------------------------------

-- T2 🡪 step 2

begin transaction

insert into books\_purchase

values ('pqrs\_ZZ\_10','MUKUL 2','sep 8, 2022',4)

-----------------------------------

-- T1 🡪 step 3

update books\_purchase set Quantity = 1

where [Book\_ID] = 'pqrs\_ZZ\_10'

select \* from books\_purchase

-----------------------------------

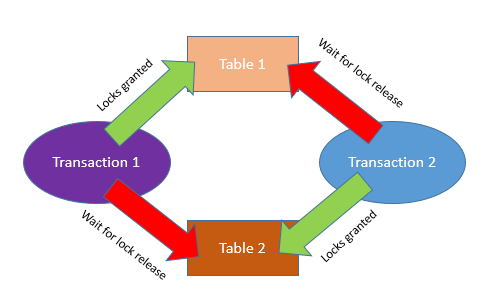
-- T2 🡪 step 4

update books\_purchase set Quantity = 2

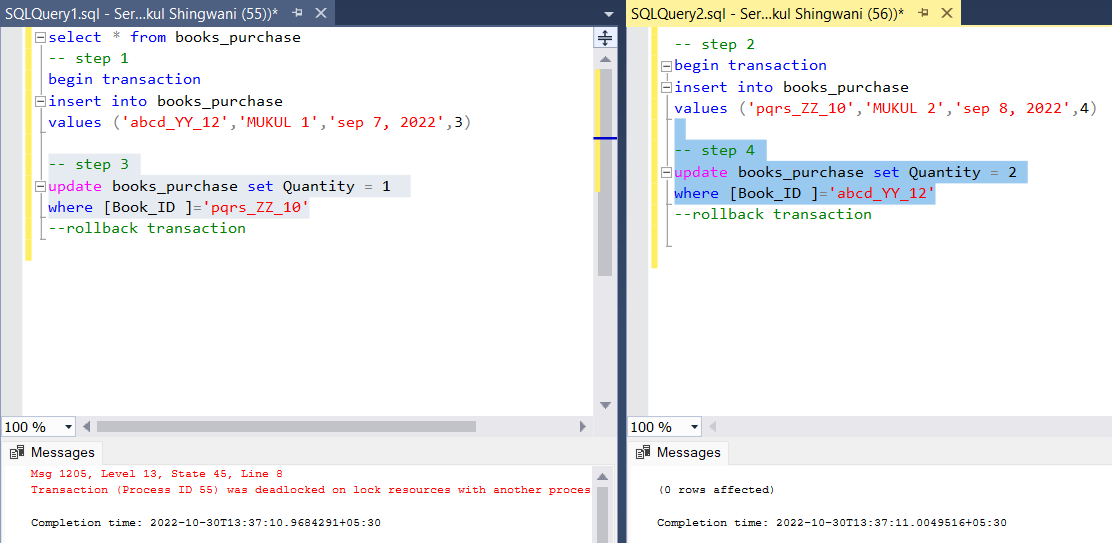
where [Book\_ID] = 'abcd\_YY\_12'

rollback transaction

In a database, a deadlock occurs when two or more processes have a resource locked, and each process requests a lock on the resource that another process has already locked. Neither of the transactions here can move forward, as each one is waiting for the other to release the lock



Since all the necessary conditions are seen to be satisfying, hence the situation is indeed of a deadlock, reference can be drawn from the diagram as well and query snapshots are also attached below.



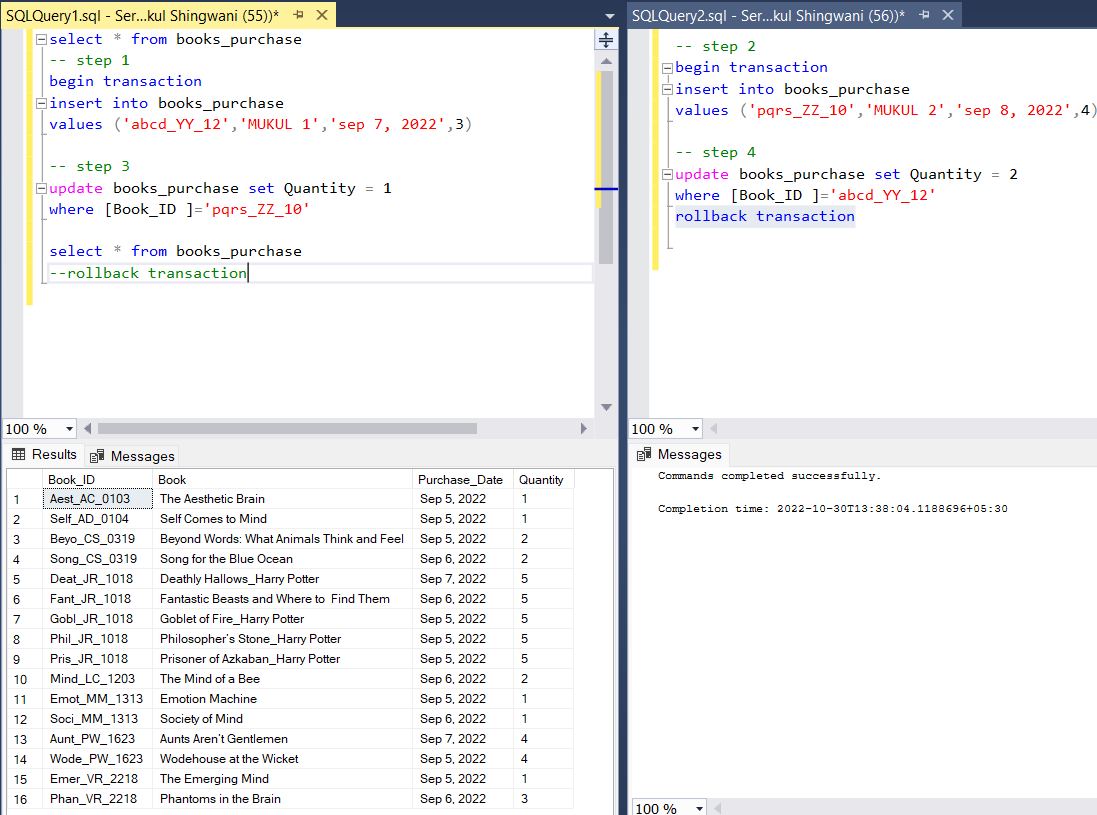
When deadlock occurs the system will choose one of the processes as the victim for deadlock, rollback that process so that the other could move forward.

**Q2) B)**

I would have rolled back the second transaction i.e. query 2 since it’s first execution started later than query 1, so rolling it back will make query 1 run, this could be thought of as a FIFO (first-in-first-out) approach, since query 1 came first let it be completed first or given priority over other transactions.

**Q2) C)**

As can been seen below, after rolling back query 2, query 1 where deadlock caused a block, successfully executed its operations.



**Q2) D)**

This code is not conflict serializable, since it cannot be resolved into a serial schedule i.e. one of its conflict equivalent and neither its precedence graph is acyclic.

T1 T2

W(A)

W(A)

W(A)

W(A)

**Precedence graph**

T1

T2

Cyclic!!

**Q3)**

* Check the ***system\_health*** session for deadlocks
* Create an **extended event** session to capture the deadlocks
* Analyse the deadlock reports and graphs to figure out the problem
* If it is possible to make improvements or change the queries involved in the deadlock

The **system\_health** is the default extended event session of the SQL Server, and it started automatically when the database engine starts. The system\_health session collects various system data, and one of them is deadlock information. The following query reads the .xel file of the system\_health session and gives information about the deadlock problems which were occurred. The system\_health session can be a good starting point to figure out the deadlock problems. The below query helps to find out the deadlock problems which is captured by the system\_health session

DECLARE @xelfilepath NVARCHAR(260)

SELECT @xelfilepath = dosdlc.path

FROM sys.dm\_os\_server\_diagnostics\_log\_configurations AS dosdlc;

SELECT @xelfilepath = @xelfilepath + N'system\_health\_\*.xel'

DROP TABLE IF EXISTS  #TempTable

SELECT CONVERT(XML, event\_data) AS EventData

        INTO #TempTable FROM sys.fn\_xe\_file\_target\_read\_file(@xelfilepath, NULL, NULL, NULL)

         WHERE object\_name = 'xml\_deadlock\_report'

SELECT EventData.value('(event/@timestamp)[1]', 'datetime2(7)') AS UtcTime,

            CONVERT(DATETIME, SWITCHOFFSET(CONVERT(DATETIMEOFFSET,

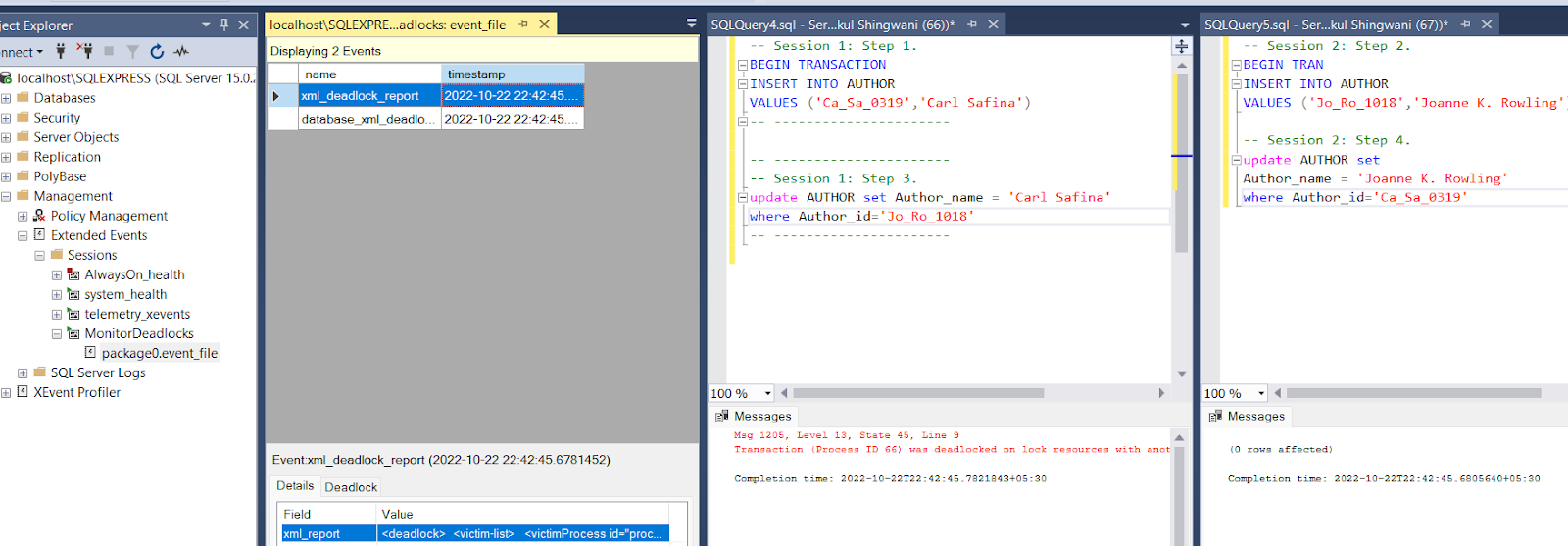
      EventData.value('(event/@timestamp)[1]', 'VARCHAR(50)')), DATENAME(TzOffset, SYSDATETIMEOFFSET()))) AS LocalTime,

            EventData.query('event/data/value/deadlock') AS XmlDeadlockReport

     FROM #TempTable

     ORDER BY UtcTime DESC;

When we click any row of the XmlDeadlockReport column, the deadlock report will appear. The database administrator found some clues about the deadlock problem through the captured data by the system\_health session. However, he thought that the system\_health session shows the more recent events because of the file size limitations, so it cannot be reliable to detect all deadlocks in SQL Server. So, he decided to create a new extended event session that can capture all the deadlocks. Extended Event is a system monitoring tool that helps to collect events and system information from SQL Server. With the help of the XEvent, we can also capture deadlock information from SQL Server. Firstly, we will launch SQL Server Management Studio and navigate to Session, which is placed under the Management folder. Right-click on the Sessions folder and select New Session.

****

**Q4**

Following were the steps of execution :-

* BOOK\_PURCHASE was updated in query 1 (txn1) → locked table 1
* BOOK was updated in query 2 (txn2) → locked table 2
* BOOK was attempted to update in query 1 → request from table 1
* BOOK\_PURCHASE was attempted to update in query 2 → request from table 2

In a database, a deadlock occurs when two or more processes have a resource locked, and each process requests a lock on the resource that another process has already locked. Neither of the transactions here can move forward, as each one is waiting for the other to release the lock

-- Session 1: Step 1.

BEGIN TRANSACTION

-- txn 1

update BOOK\_PURCHASE set Book\_name = 'Mukul 1' where Book\_id = 'Aest\_AC\_0103'

-- ----------------------

-- Session 2: Step 2.

BEGIN TRANSACTION

–- txn 2

update BOOK set Book\_name = 'Mukul 2' where Book\_id = 'Aest\_AC\_0103'

-- ----------------------

-- Session 1: Step 3.

--txn 2

update BOOK set Book\_name = 'Mukul 1' where Book\_id = 'Aest\_AC\_0103'

select \* from BOOK

-- commit Transaction

-- ----------------------

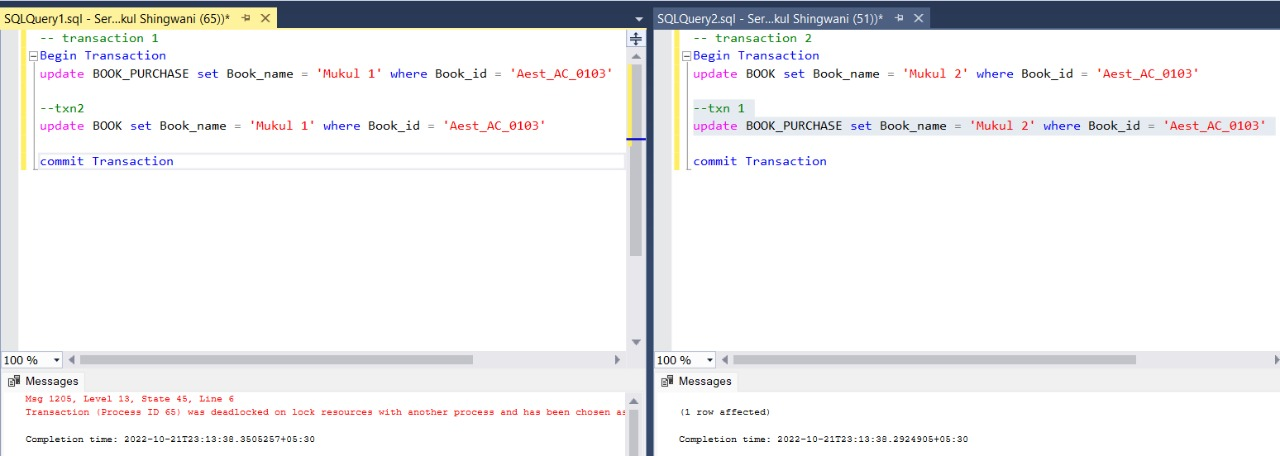
-- Session 2: Step 4.

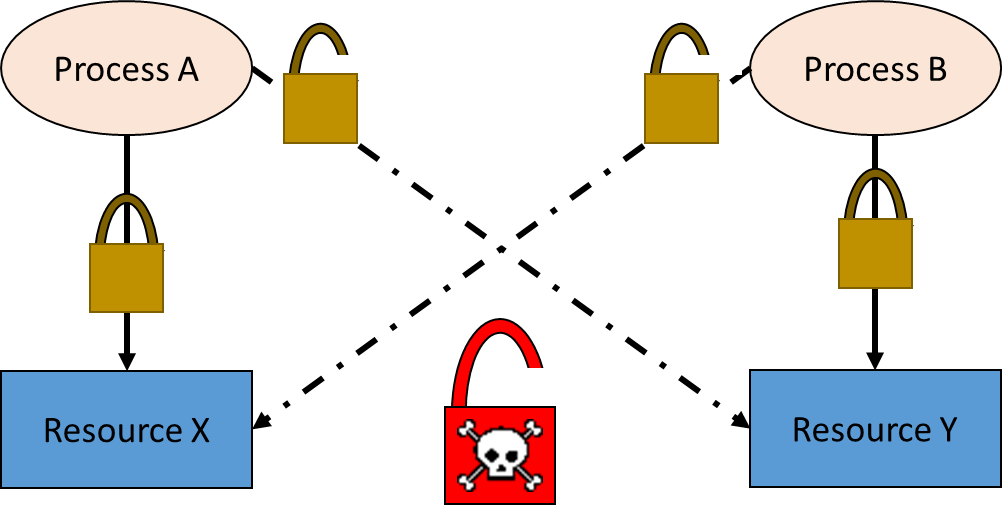
--txn 1

update BOOK\_PURCHASE set Book\_name = 'Mukul 2' where Book\_id = 'Aest\_AC\_0103'

-- commit Transaction

Since all the necessary conditions are seen to be satisfying, hence the situation is indeed of a deadlock, reference can be drawn from the diagram as well and query snapshots are also attached below.

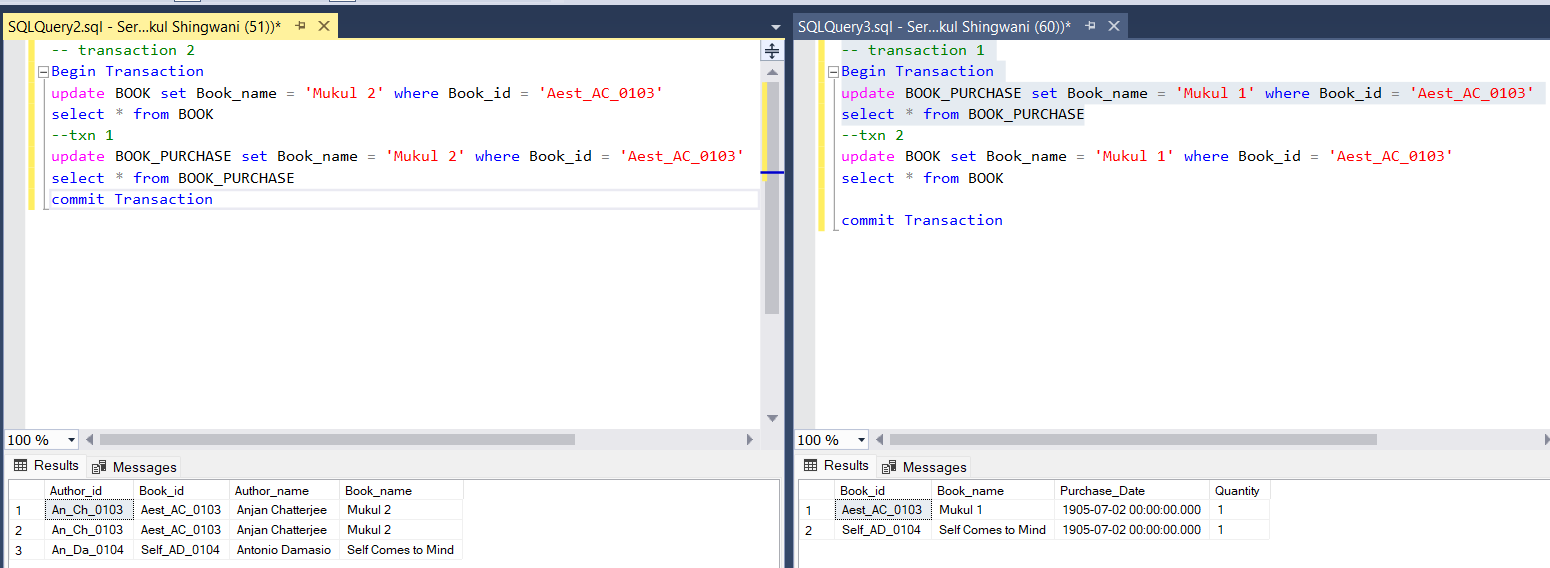


****

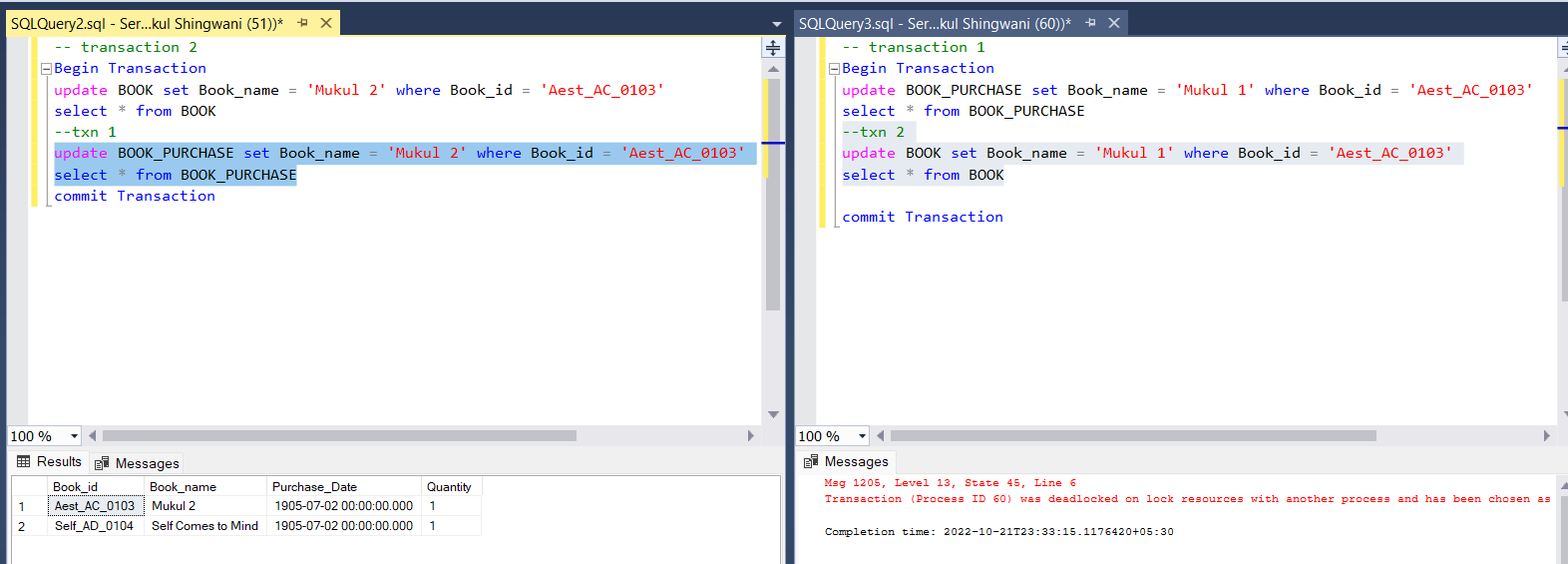
**Q5**

Taking reference of the above code, we know there arises a deadlock situation, and using rollback it can be resolved as can be seen in the images below.

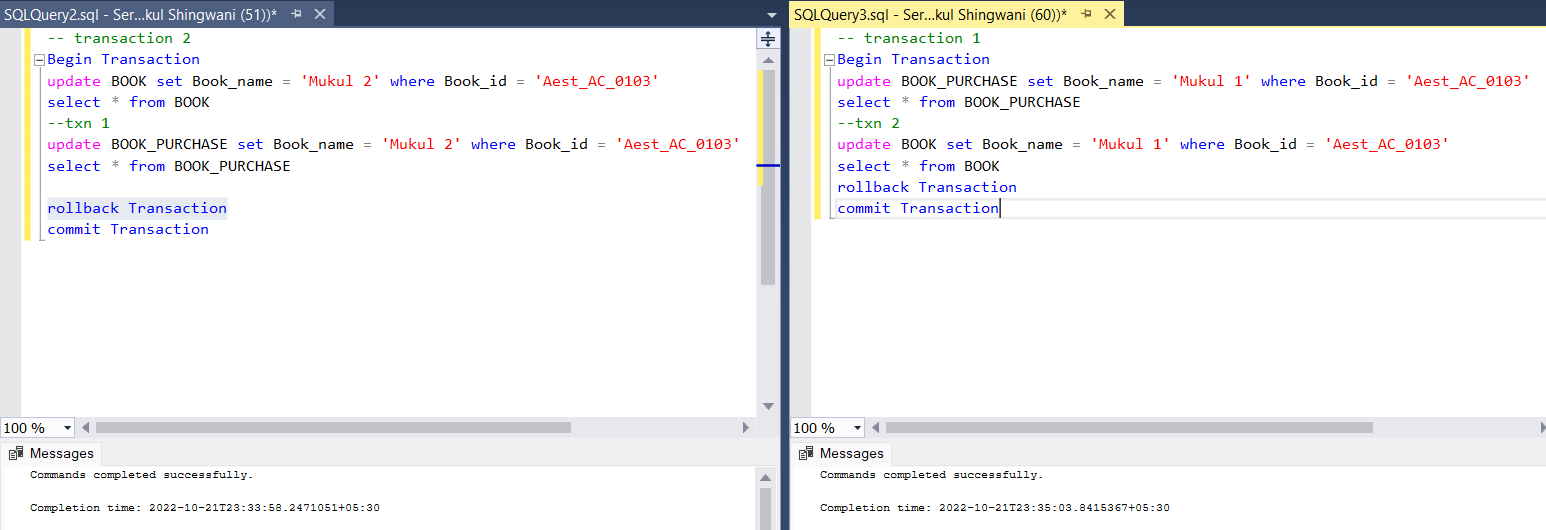
**Query**



**Deadlock**



**Rollback**



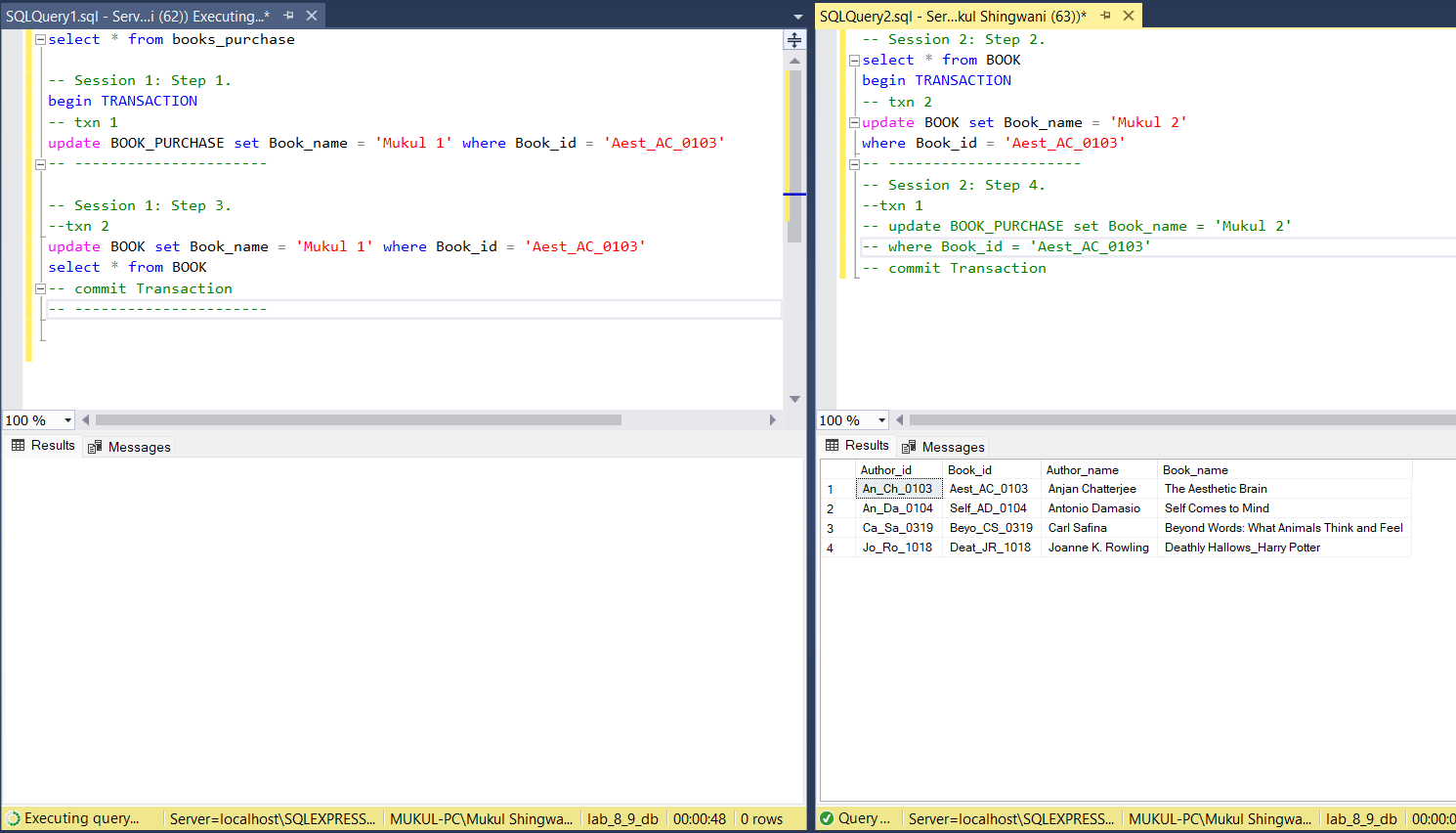
**Q6**

Wait and Die scheme

In this scheme, if a transaction requests for a resource which is already held with a conflicting lock by another transaction then the DBMS simply checks the timestamp of both transactions. It allows the older transaction to wait until the resource is available for execution.

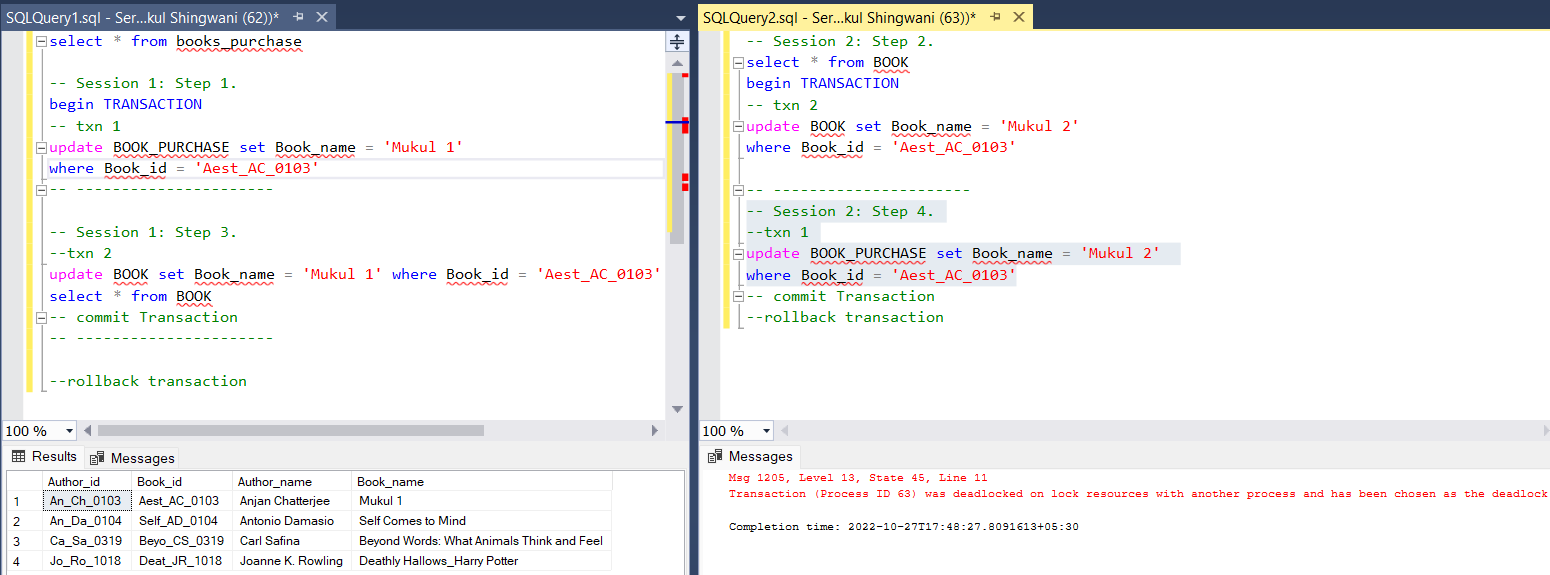
Let's assume there are two transactions Ti and Tj and let TS(T) is a timestamp of any transaction T. If T2 holds a lock by some other transaction and T1 is requesting for resources held by T2 then the following actions are performed by DBMS:

1. Check if TS(Ti) < TS(Tj) - If Ti is the older transaction and Tj has held some resource, then Ti is allowed to wait until the data-item is available for execution. That means if the older transaction is waiting for a resource which is locked by the younger transaction, then the older transaction is allowed to wait for resource until it is available.



As can be seen in this image, T1 started earlier than T2 transaction, T1 had a lock on table BOOK\_PURCHASE and T2 had a lock on BOOK table, Now, when T1 requested for BOOK table, it was allowed to wait, since it was the older transaction among the two.

1. Check if TS(Ti) < TS(Tj) - If Ti is older transaction and has held some resource and if Tj is waiting for it, then Tj is killed and restarted later with the random delay but with the same timestamp.

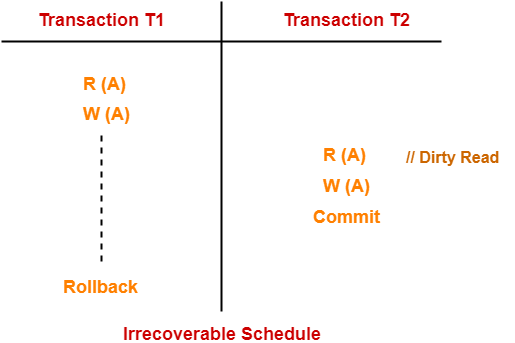


As can be seen in this image, T1 started earlier than T2 transaction, T1 had a lock on table BOOK\_PURCHASE and T2 had a lock on BOOK table, Now, when T1 requested for BOOK table, it was allowed to wait, but when T2 requested for a resource held by T1, T2 transaction got killed immediately and a deadlock warning came, since it was not older than T1 transaction.

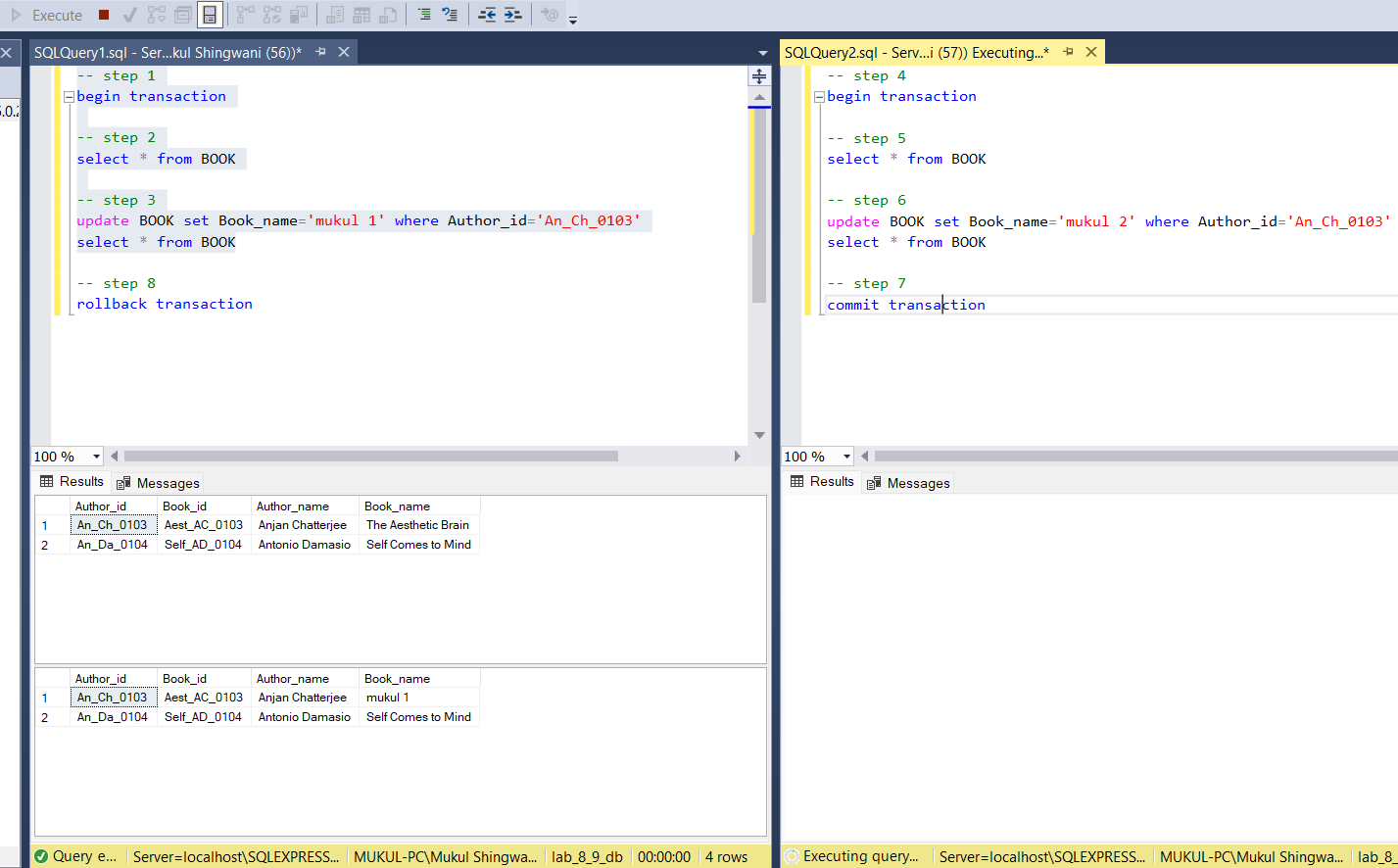
**Q7**

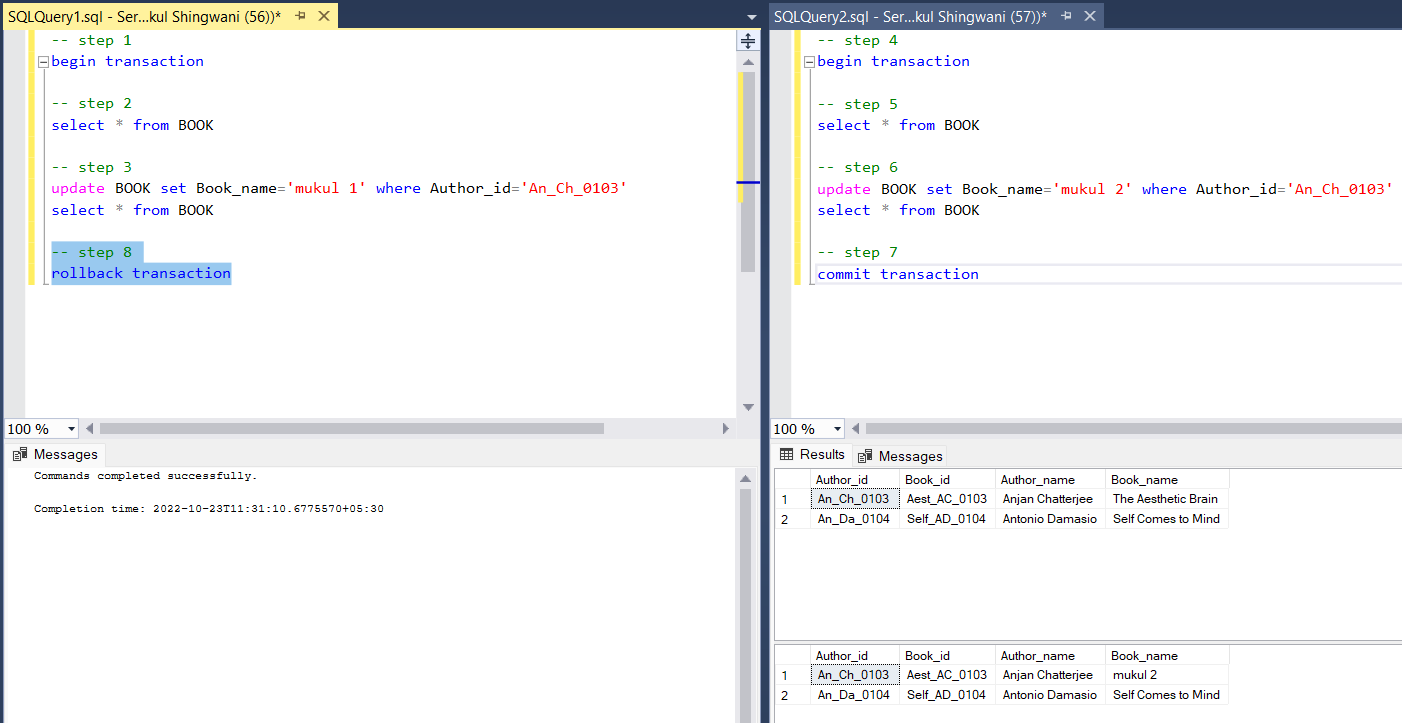
If a transaction reads a data item after it is written by an uncommitted transaction it leads to an **irrevocable transaction**

i.e. a dirty read operation from an uncommitted transaction and commits before the transaction from where it has read the value, then such a schedule is called an irrecoverable schedule



Here such kind of a transaction flow was followed, where session 2 reads, writes and then commits before session 1 and finally session 1 rollbacks, but since session 2 has alread read the dirty values after write of session 1, it falls into the trap of Dirty Read problem, and this is an irrecoverable schedule since session 2 committed before session 1





**T1 T2**

-- step 1 -- step 4

begin transaction

begin transaction

-- step 2 -- step 5

select \* from BOOK

select \* from BOOK

-- step 6

update BOOK set Book\_name='mukul 2' where Author\_id='Ca\_Sa\_0319'

select \* from BOOK

-- step 3

insert into BOOK values

('Ca\_Sa\_0319','Beyo\_CS\_0319','Carl Safina',

'Beyond Words: What Animals Think and Feel')

select \* from BOOK -- step 7

commit transaction

-- step 8

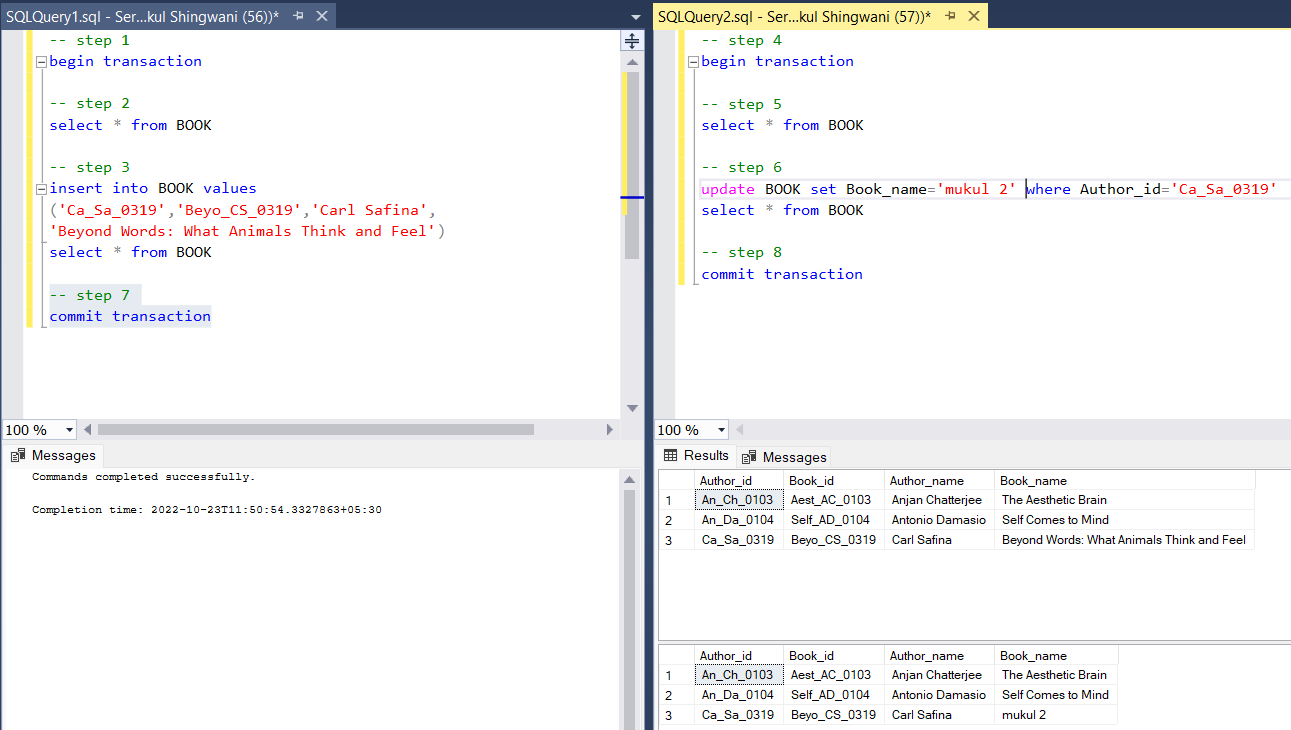
rollback transaction

This example will also lead to an irrecoverable schedule, since T2 updates on the newly inserted row by T1, which finally gets rolled back and lost, so T2 updates are not meaningful.

To make them recoverable we need to follow a recoverable schedule approach

A **recoverable schedule** is basically a schedule in which the commit operation of a particular transaction that performs read operation is delayed until the uncommitted transaction either commits or roll backs

i.e. schedules in which transactions commit only after all transactions whose changes they read commits.



Here, T2 was committed after T1 committed or rollback was also an option, Now this has shaped into a form of a recoverable schedule, and the workflow for the same is given below.

**T1 T2**

-- step 1 -- step 4

begin transaction

begin transaction

-- step 2 -- step 5

select \* from BOOK

select \* from BOOK

-- step 6

update BOOK set Book\_name='mukul 2' where Author\_id='Ca\_Sa\_0319'

select \* from BOOK

-- step 3

insert into BOOK values

('Ca\_Sa\_0319','Beyo\_CS\_0319','Carl Safina',

'Beyond Words: What Animals Think and Feel')

select \* from BOOK

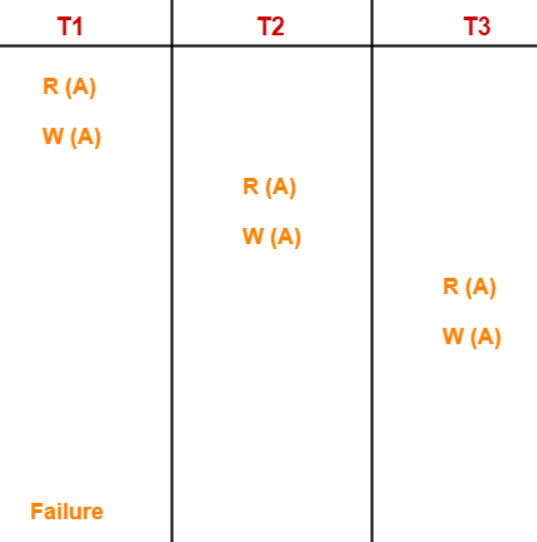
-- step 7

commit transaction -- step 8

commit transaction

**Q8)**

A cascading rollback occurs in database systems when a transaction (T1) causes a failure and a rollback must be performed. Other transactions dependent on T1's actions must also be rollbacked due to T1's failure, thus causing a cascading effect. That is, one transaction's failure causes many to fail.



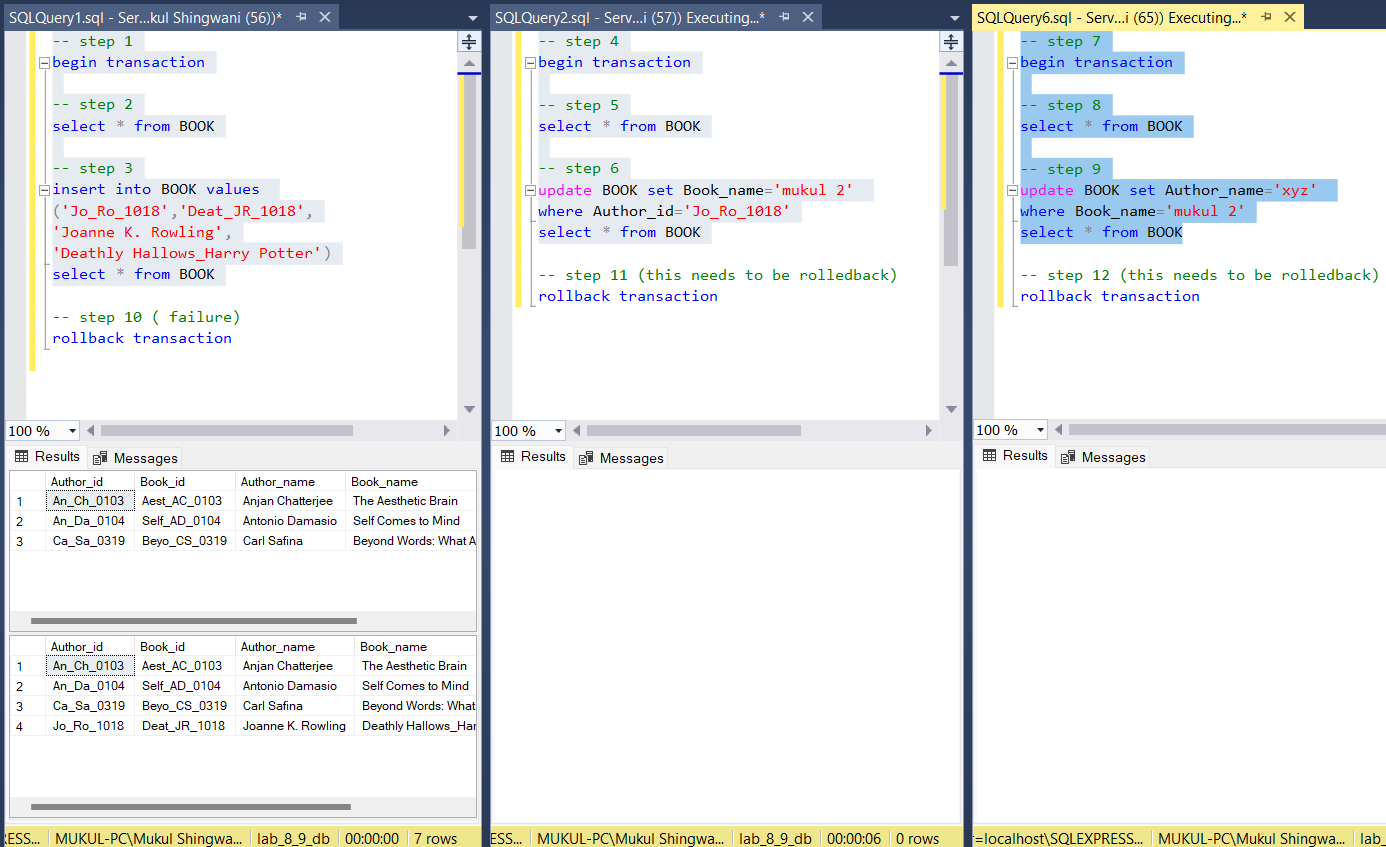
Here,

* Transaction T2 depends on transaction T1.
* Transaction T3 depends on transaction T2.

In this schedule,

* The failure of transaction T1 causes the transaction T2 to rollback.
* The rollback of transaction T2 causes the transaction T3 to rollback.

Such a rollback is called as a **Cascading Rollback**.



**T1 T2 T3**

-- step 1 -- step 4 -- step 7

begin transaction begin transaction begin transaction

-- step 2 -- step 5 -- step 8

select \* from BOOK select \* from BOOK select \* from BOOK

-- step 3 -- step 6

insert into BOOK values update BOOK set Book\_name

='mukul 2' where Author\_id

='Jo\_Ro\_1018'

select \* from BOOK

('Jo\_Ro\_1018','Deat\_JR\_1018',

'Joanne K. Rowling',

'Deathly Hallows\_Harry Potter') -- step 9

select \* from BOOK update BOOK set

Author\_name='xyz'

where Book\_name='mukul2'

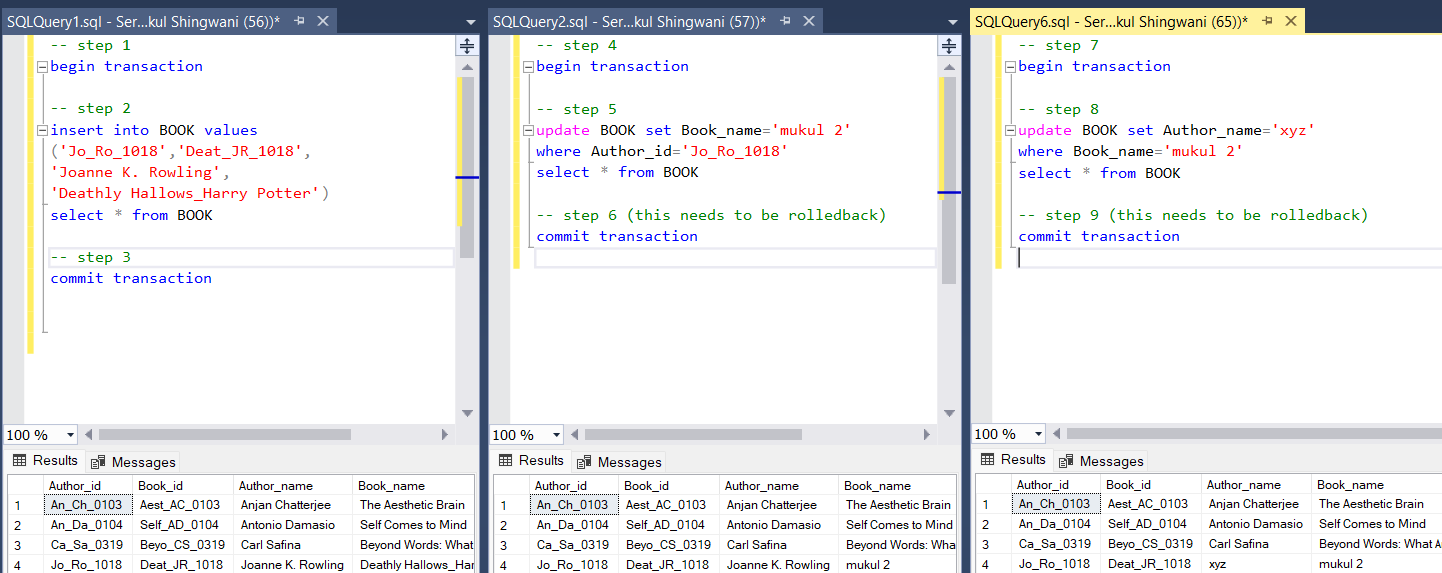
select \* from BOOK

-- step 10 (failure) -- step 11 -- step 12

rollback transaction rollback transaction rollback transaction

**Cascadeless schedule:** when a transaction is not allowed to read data until the last transaction which has written it is committed or aborted, these types of schedules are called cascadeless schedules. Here, the updated value of X is read by transaction T2 only after the commit of transaction T1

We converted the above cascading rollback schedule to a cascadeless schedule, it can be seen from the image attached below, I’ve also shown the code and flow of execution the transactions below, as can be seen it performs without any discrepancies or errors.



**T1 T2 T3**

-- step 1 -- step 4 -- step 7

begin transaction begin transaction begin transaction

-- step 2 -- step 5

insert into BOOK values update BOOK set Book\_name

='mukul 2' where Author\_id

='Jo\_Ro\_1018'

select \* from BOOK

('Jo\_Ro\_1018','Deat\_JR\_1018',

'Joanne K. Rowling',

'Deathly Hallows\_Harry Potter') -- step 8

select \* from BOOK update BOOK set

Author\_name='xyz'

where Book\_name='mukul2'

select \* from BOOK

-- step 3 -- step 6 -- step 9

commit transaction commit transaction commit transaction

----------------- X ---------------------- END ---------------------- X -----------------