INM370 – Advanced Databases
Tutorial 3 – Relational Database Management Issues
Model Answers

I)

Atomicity ensures that either the results of **all** the operations in a transaction are reflected properly in the database, or **none** of them is.
Example: a login/password is assigned to a student, but the student's debit card has not been debited (Note: an assumption is that both actions are part of a single transaction).

Consistency ensures that the execution of a transaction must preserve the consistency of the database, guaranteeing that any data written to the database must be valid according to all defined rules (including, but not limited to, constraints, cascades, triggers).
Example: The debit card of the student is debited such that the balance is negative (Note: an assumption is that the student's account has no overdraft). An alternative example: two students are assigned the same login/password details, despite database constraint precluding such a case.

Isolation ensures concurrent transactions are executed independently of one another.
Example: A transaction, T1, reads the data for all the students whose surname is "Jones". On the first execution of the read, T1 reads 14 rows. Transaction T2 executes concurrently and inserts a record about a new "Jones" student, and consequently T1 reads 15 rows when re-executing the same read operation.

Durability ensures that the effects of i) successfully completed transactions are stored in the database and not lost because of failure, and ii) unsuccessful transactions are not stored in the database permanently.
Example: The debiting is successfully completed - the balance of the bank account is decreased, but there is a crash in the system and the student login/password are not stored on the permanent storage.

II)

Transaction Isolation levels control the degree to which the execution of one transaction is isolated from the execution of other concurrent transaction(s) (which are accessing the same data).

ISO SQL-92 defines the following isolation levels (from the most relaxed to the strictest level), with respect to anomalies they do/do not guard against:
Serializable, **Repeatable read**, **Read committed** and **Read uncommitted**

Isolation Level	Transactions	Dirty Reads	Non-Repeatable Reads	Phantom Reads
TRANSACTION_NONE	Not supported	Not applicable	Not applicable	Not applicable
READ_UNCOMMITTED	Supported	Allowed	Allowed	Allowed
READ_COMMITTED	Supported	Prevented	Allowed	Allowed
REPEATABLE_READ	Supported	Prevented	Prevented	Allowed
SERIALIZABLE	Supported	Prevented	Prevented	Prevented

A *lower* isolation level increases concurrency (improves throughput), but this happens (usually) at the expense of data correctness. Conversely, a *higher*, i.e. stricter, isolation level ensures that data remain correct, but can negatively affect concurrency.

III)

This is a Demonstration exercise – please follow the instructions given in the exercise.

The first invocation of the SELECT statement in Terminal 1 (see line 5)) returns the value of Balance to be 10.
However, the second invocation of the same SELECT statement in Terminal 1 (see line 8)) returns the value of Balance to be 25 (different from the initial value read!).

Thus, the transaction executed in Terminal 1 sees different values returned by the same SELECT operation. This is referred to as *Non-repeatable* read anomaly.

“Repeatable read” and “Serializable” transaction isolation levels would guard against such anomaly.

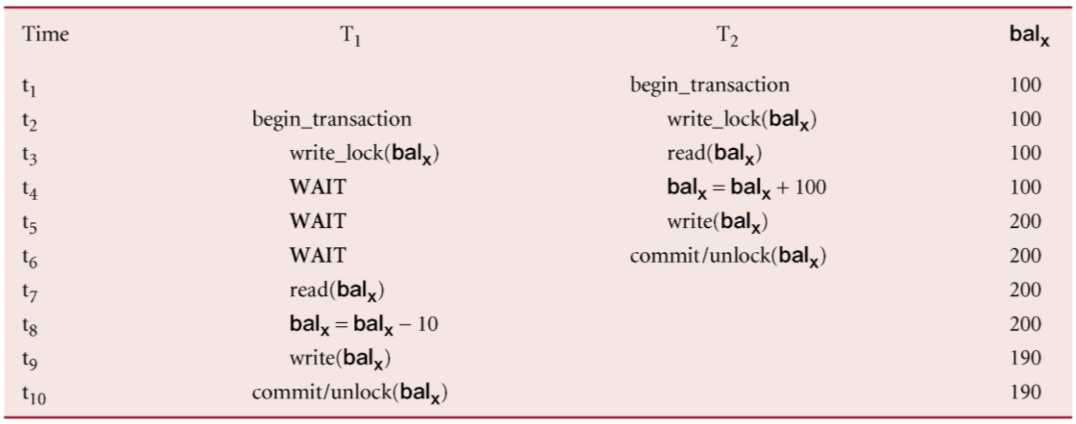
No, it makes no difference, because the problem occurs in, and the resolution is provided by, the database server software, and not the client applications such as *SQLDeveloper* or *SQLcl*, or a java application (using JDBC driver) etc.

IV)

You need to review, and understand, the referred problems (in the lecture slides and the stated book sections) and the suggested solutions (in the relevant book sections, too).

For example, see the following:

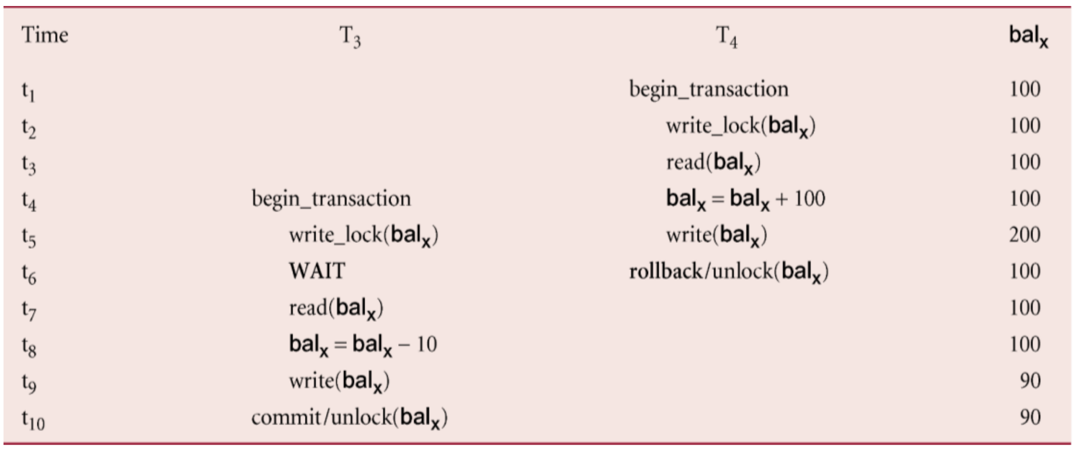
Preventing Lost Update problem using 2PL



Time	T_1	T_2	$\mathbf{bal_x}$
t_1		begin_transaction	100
t_2	begin_transaction	write_lock($\mathbf{bal_x}$)	100
t_3	write_lock($\mathbf{bal_x}$)	read($\mathbf{bal_x}$)	100
t_4	WAIT	$\mathbf{bal_x} = \mathbf{bal_x} + 100$	100
t_5	WAIT	write($\mathbf{bal_x}$)	200
t_6	WAIT	commit/unlock($\mathbf{bal_x}$)	200
t_7	read($\mathbf{bal_x}$)		200
t_8	$\mathbf{bal_x} = \mathbf{bal_x} - 10$		200
t_9	write($\mathbf{bal_x}$)		190
t_{10}	commit/unlock($\mathbf{bal_x}$)		190

Loss of T_2's update avoided by preventing T_1 from reading bal_x before the update and corresponding commit take place (use of strict 2PL).

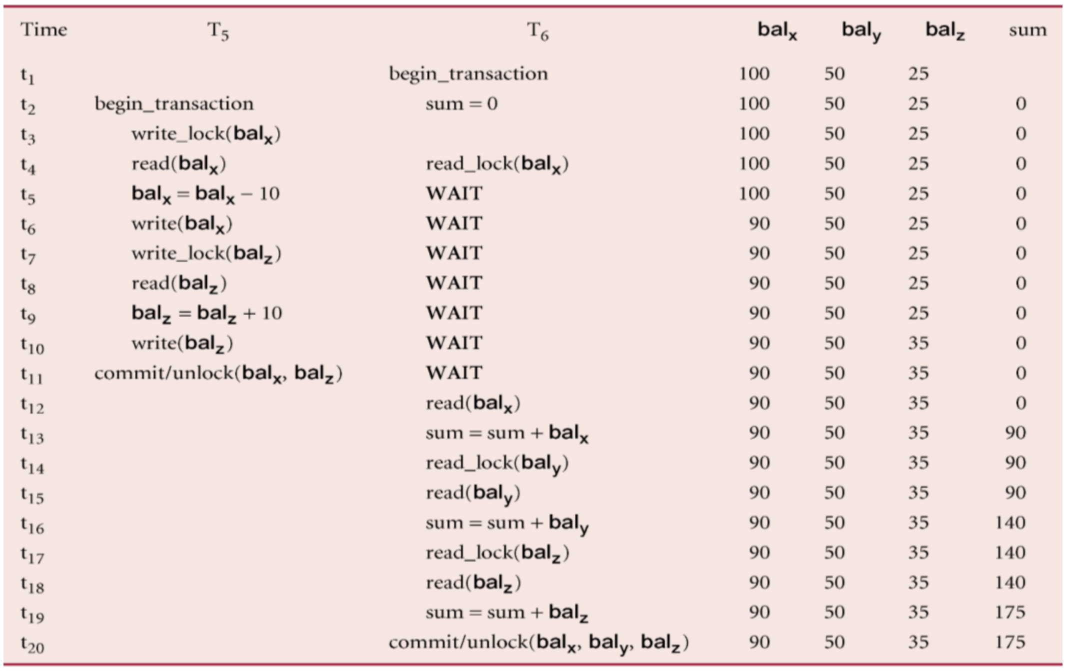
Preventing Uncommitted Dependency Problem using 2PL



Time	T_3	T_4	bal_x
t_1		begin_transaction	100
t_2		write_lock(bal_x)	100
t_3		read(bal_x)	100
t_4	begin_transaction	$bal_x = bal_x + 100$	100
t_5	write_lock(bal_x)	write(bal_x)	200
t_6	WAIT	rollback/unlock(bal_x)	100
t_7	read(bal_x)		100
t_8	$bal_x = bal_x - 10$		100
t_9	write(bal_x)		90
t_{10}	commit/unlock(bal_x)		90

Problem avoided by preventing T_3 from reading bal_x until after T_4 aborts/rolls back, or commits.

Preventing Inconsistent Analysis Problem using 2PL



Time	T_5	T_6	bal_x	bal_y	bal_z	sum
t_1		begin_transaction	100	50	25	
t_2	begin_transaction	sum = 0	100	50	25	0
t_3	write_lock(bal_x)		100	50	25	0
t_4	read(bal_x)	read_lock(bal_x)	100	50	25	0
t_5	$bal_x = bal_x - 10$	WAIT	100	50	25	0
t_6	write(bal_x)	WAIT	90	50	25	0
t_7	write_lock(bal_z)	WAIT	90	50	25	0
t_8	read(bal_z)	WAIT	90	50	25	0
t_9	$bal_z = bal_z + 10$	WAIT	90	50	25	0
t_{10}	write(bal_z)	WAIT	90	50	35	0
t_{11}	commit/unlock(bal_x, bal_z)	WAIT	90	50	35	0
t_{12}		read(bal_x)	90	50	35	0
t_{13}		sum = sum + bal_x	90	50	35	90
t_{14}		read_lock(bal_y)	90	50	35	90
t_{15}		read(bal_y)	90	50	35	90
t_{16}		sum = sum + bal_y	90	50	35	140
t_{17}		read_lock(bal_z)	90	50	35	140
t_{18}		read(bal_z)	90	50	35	140
t_{19}		sum = sum + bal_z	90	50	35	175
t_{20}		commit/unlock(bal_x, bal_y, bal_z)	90	50	35	175

Problem avoided by preventing T_6 from reading bal_x and bal_z (the two values that changed), and bal_y too, until after T_5 completed updates.