Register Number | | | | | | | | |

Sri Sivasubramaniya Nadar College of Engineering, Kalavakkam – 603 110

(An Autonomous Institution, Affiliated to Anna University, Chennai)

Department of Computer Science and Engineering

Continuous Assessment Test – III (Online Test)

Question Paper

Degree & Branch	B.E CSE				**Semester**	IV
Subject Code & Name	UCS1404 Database Management Systems				**Regulation: 2018**	
Academic Year	2019-20	**Batch**	2018-22	**Date**	**15.05.2020**	**FN**
Time: 90 Minutes	**ANSWER KEY**				**Maximum: 50 Marks**	

SET I + SET II + SET III = 16+18+16 marks

<K3>	1. a. For the deferred update recovery technique, the modified log entries are: [start_transaction, T1] Initial values:A=30; B=12; C=30; D=20 ~~[read_item, T1 , A]~~ ~~[read_item, T1 , D]~~ [write_item, T1 , D, 20, 25] [commit, T1] [checkpoint] [start_transaction, T2] ~~[read_item, T2 , B]~~ [start_transaction, T4] ~~[read_item, T4 , D]~~ [write_item, T4 , D, 25, 15] [write_item, T2 , B, 12, 18] [commit, T2] [start_transaction, T3] [write_item, T3 , C, 30, 40] ~~[read_item, T4 , A]~~ [write_item, T4 , A, 30, 20] [commit, T4] ~~[read_item, T3 , D]~~ [write_item, T3 , D, 15, 25] < –----- System Crash <u>Modified Log entries:</u> REDO type entries contains only the AFIM values in write_item operations: [start_transaction, T1] [write_item, T1, D, 25] [commit, T1] **[checkpoint]** **[start_transaction, T2]** **[start_transaction, T4]** [write_item, T4 , D, 15]	<CO4>

[write_item, T2 , B, 18]
[commit, T2]
[start_transaction, T3]
[write_item, T3 , C, 40]
[write_item, T4 , A, 20]
[commit, T4]
[write_item, T3, D, 25]

< –----- *System Crash*

Assuming only REDO operations,
two lists: commit list and active list.
The commit list contains → T4, T2
the write_item operations of T4 and T2 are REDONE in the same order as in log entry.
[write_item, T4 , D, 15]
[write_item, T2 , B, 18]
[write_item, T4 , A, 20]

(or) by starting from the end of the log and redoing only the *last update of each item X*. Whenever an item is redone, it is added to a list of redone items and is not redone again.
[write_item, T4 , A, 20]
[write_item, T2 , B, 18]
[write_item, T4 , D, 15]

Active list contains → T3
the write_item operations of T3 are ignored.

The final values of data items after the recovery process:
A=20; B=18; C=30; D=15

1.b
For ***immediate update***, the log entries contains the write_item (<u>UDO type entries</u>) operations:

[start_transaction, T1]   Initial values:A=30; B=12; C=30; D=20
[write_item, T1 , D, 20, 25]
[commit, T1]
[checkpoint]
[start_transaction, T2]
[start_transaction, T4]
[write_item, T4 , D, 25, 15]
[write_item, T2 , B, 12, 18]
[start_transaction, T3]
[write_item, T3, C, 30, 40]
[write_item, T4 , A, 30, 20]
[commit, T4]
[write_item, T3, D, 15, 25]

< –----- System Crash

Before the system crash, the final values of each data item:
A= 20; B=18 ; C=40 ; D= 25

By applying immediate update protocol, the two lists:
commit list contains T4 → REDO

active list contain T2 and T3 → UNDO

The following operations are REDONE.
The 2 write_item operations of T4 is REDONE in the same order as in log entry.
[write_item, T4 , D, 25, 15]
[write_item, T4 , A, 30, 20]

The write_item operations of T2 and T3 are UNDONE in the the reverse of the order in which they were written into the log.
The following 3 write_item operations are undone: given in the reverse order
[write_item, T3, D, 15, 25]
[write_item, T3, C, 30, 40]
[write_item, T2, B, 12, 18]

The concurrency control protocol (immediate update) produces strict schedules. However, deadlocks can occur in strict two-phase locking, thus requiring abort and UNDO of transactions.
From the log entry, there is no deadlock among the transactions. Hence cascading rollback does **not** takes place.

<K3>

2.
Consider the following relational schema: (4)
Department (did, dname, location)
Employee (eid, fname, lname, designation, salary, dept)

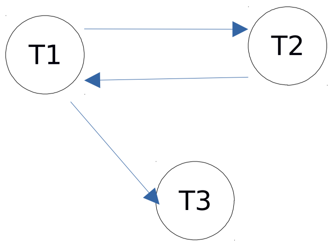
```
db.department.insertMany([
{   _id: "A100", name: "Admin", location: "Chennai"},
  {   _id: "R200", name: "Research", location: "NewDelhi"}
]}


db.employee.insertMany([
{_id: "E100", name:(fname:"Vishal",lname:"Ram"}, desig:"Manager",
  salary:50000, dept:"A100"},
{_id: "E101", name:(fname:"Rakesh",lname:"Sinha"}, desig:"Asst.",
  salary:42000, dept:"A100"},
{_id: "E200", name:(fname:"John",lname:"Mathew"}, desig:"Sw.Eng",
  salary:55000, dept:"A200"}
{_id: "E201", name:(fname:"Tushal",lname:"Varma"}, desig:"Team
  Lead", salary:60000, dept:"A200"},
]}
```

<CO5>

<K3>

3. S_1: $r_1(X)$, $r_3(Y)$, $\mathbf{r_2(X)}$, $\mathbf{w_1(X)}$, $w_2(X)$, $w_3(Y)$, $r_1(Z)$, $w_1(Z)$, $r_3(Z)$, $w_3(Z)$

a) Draw the precedence graph for S_1 and state whether the schedule is serializable or not. Why or why not? (2)



Not serializable, as there is a cycle between T1 and T2.

b) Now swap the operations in S_1 that is highlighted in bold and consider as S_2.

<CO4>

Check whether the schedule S_2 is *conflict serializable* through the swapping of operations. If so, give the equivalent serial schedule(s). (6)

After Swapping of operations, schedule S2:

T1	**T2**	**T3**
r(X)		
		r(Y)
w(X)		
	r(X)	
	w(X)	
		w(Y)
r(Z)		
w(Z)		
		r(Z)
		w(Z)

After swapping non-conflict operations:

T1	**T2**	**T3**
r(X)		
w(X)		
r(Z)		
w(Z)		
	r(X)	
	w(X)	
		r(Y)
		w(Y)
		r(Z)
		w(Z)

The equivalent serial schedule: T1 → T2 → T3

T1	**T2**	**T3**
r(X)		
w(X)		
r(Z)		
w(Z)		
		r(Y)
		w(Y)
		r(Z)
		w(Z)
	r(X)	
	w(X)	

The equivalent serial schedule: T1 → T3 → T2

c) Is the schedule S_2 is view serializable or not? Justify. If so, determine the equivalent serial schedule(s). (4)

Justification of S2 as view serializable .

For schedule S2:
View of X: T1 writes → T2 reads
view of Z: T1 writes → T3 reads
Last Write(X): T2
Last Write(Z): T3

For serial: T1 → T2 → T3
View of X: T1 writes → T2 reads
view of Z: T1 writes → T3 reads
Last Write(X): T2
Last Write(Z): T3
Hence S2 is <u>view serializable to serial T1 → T2 → T3</u>

For serial: T1 → T3 → T2
View of X: T1 writes → T2 reads
view of Z: T1 writes → T3 reads
Last Write(X): T2
Last Write(Z): T3
Hence S2 is <u>view serializable to serial T1 → T3 → T2</u>

<K3>

4. Consider the three transactions T1, T2, T3.

Schedule:S3

T1	**T2**	**T3**
w(A)		
r(B)		
r(C)		
		w(B)
w(B)		
		w(C)
	r(C)	
	w(B)	
	w(C)	
		r(A)

a) Is this schedule conflict serializable? Why or why not? If so, give the equivalent serial schedule.

The schedule is not conflict serializable. Since many write operations are not constrained writes. w(A) in T1, w(B) in T2 and w(B), w(C) are **Blind writes**. Also the precedence graph is cyclic between T1 → T3 → T1.

b) Is this schedule view serializable? Why or why not? If so, give the equivalent serial schedule.

Equivalent serial schedule:

<CO4>

T1	T3	T2
write(A)		
read(B)		
read(C)		
write(B)		
	write(B)	
	write(C)	
	read(A)	
		read(C)
		write(B)
		write(C)

For schedule S3:
View of A: T1 writes → T3 reads
view of C: T3 writes → T2 reads
Last Write(A): T1
Last Write(B) & Write(C): T2

For serial: T1 → T3 → T2
View of A: T1 writes → T3 reads
view of C: T3 writes → T2 reads
Last Write(A): T1
Last Write(B) & Write(C): T2
Hence S3 is view serializable to serial T1 → T3 → T2

<K3>

5. Consider the two transactions T_1 and T_2 :

$T_1 : r_1(Y); r_1(X); w_1(X);$

$T_2 : r_2(X); w_2(X);$

a) Without changing the order of operations in transaction, write a serializable schedule that follows *basic* 2PL.

Serializable schedule will have interleaving of operations. Basic 2PL has growing and shrinking phase in locking.
Assumption: No upgradation/downgradation of locks. Uses shared/exclusive locks.
This is the only possible serializable schedule using 2PL

T1	T2
S_lock(Y)	
r(Y)	
	X_lock(X)
	r(X)
	w(X)
	Unlock (X)
X_lock(X)	
Unlock(Y)	
r(X)	
w(X)	
Unlock(X)	

b) Can you have a strict 2PL schedule with out the *commit/abort* operation? Why or why not?
Strict 2PL releases the **write locks** only after reaching the termination (*commit /*

<CO4>

abort) of transaction.

Now consider the modified version of transaction T_2 *as* T_2'' :
T_2''*:* $r_2(X); r_2(Y); w_2(X);$ and add the operation *commit* at the end of each transaction T_1 and T_2''. Write a serializable schedule that implements *strict* 2PL.

T1	**T2’’**
S_lock(Y)	
r(Y)	
	X_lock(X)
	r(X)
	S_lock(Y)
	r(Y)
	w(X)
	Commit;
X_lock(X)	
r(X)	
w(X)	
Commit:	

Above is the serializable strict 2PL using the transactions T1 and T2”.

c) Now considering the pair of transactions [T_1 ,T_2] and [T_1 ,T_2''] which pair can be serialized to follow *rigorous* 2PL? Why?

For rigorous 2PL, both read and write locks will be released only after the transaction termination.
By considering T1 and T2, **rigorous 2PL is possible**, because T2 does not depends on any lock from T1.

T1	**T2**
S_lock(Y)	
r(Y)	
	X_lock(X)
	r(X)
	w(X)
	Commit;
X_lock(X)	
r(X)	
w(X)	
Commit:	

By considering T1 and T2”, **rigorous 2PL is not possible**. Because, T2” depends on S_lock(Y) which is hold by T1. This lock will not be released until T1 terminates.

<K3>

6. Write the operations in MongoDB:
a) Find the order documents whose name starts with ‘R’ and order amount greater than 2000.

```
db.orders.find( { cust: /^R/, amt: { $gt: 2000 }} )
```

<CO5>

	b) Find all documents where the `items` array has at least one embedded document that contains the field `qty` whose value is greater than or equal to 5.
	
	`db.orders.find( { 'items.qty': { $gte: 5 } } )`
	
	c) Find the order number and customer who ordered for at least one “italian” pizza.
	
	`db.orders.find( {"items":{ $elemMatch:{ qty:{$gte:1}, pizza: "italian" } } }, {cust:1} )`	
