Written Assignment 2 Reference Solutions

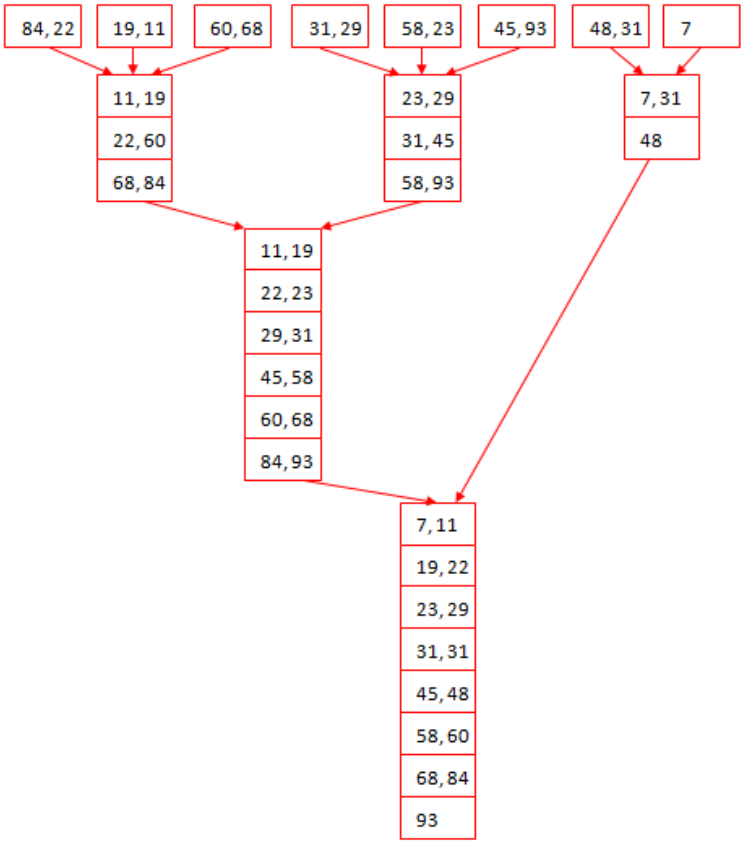
Assigned: Apr 13, 2022

Part I. Query Processing & Optimization

1. Suppose we wish to sort the following values: 84, 22, 19, 11, 60, 68, 31, 29, 58, 23, 45, 93, 48, 31, 7. Assume that: you have three pages of memory for sorting; you will use an external sorting algorithm with a 2-way merge; a page only holds two values. Answer the following questions:

(1) For each sorting pass, show the contents of all temporary files. (Use a similar diagram as slide 10 in our external sorting slide deck).
(2) If you have 100 pages of data, and 10 pages of memory, what is the minimum number of passes required to sort the data.
(3) If you have 500,000 pages of data, what is the minimum number of pages of memory required to sort the data in 3 passes.

Solution. (1) The contents of all temporary files are shown as below:



(2) $1 + \lceil \log_9 \lceil 100/10 \rceil \rceil = 3$

(3) $1 + \lceil \log_{M-1} \lceil (N)/(M) \rceil \rceil = 3$, where $N = 50{,}000$. Hence, $\lceil N/M \rceil \leq (M-1)^2$, and we got $N \leq M(M-1)^2$, and $M = 81$. You can also simplify this to $N \leq M^3$, in which case $M = 80$

2. Consider the Movie Stars database that contains the following relations:

StarsIn(movieTitle, movieYear, starName)
MovieStar(Name, address, gender, birthdate)

For the following SQL query:

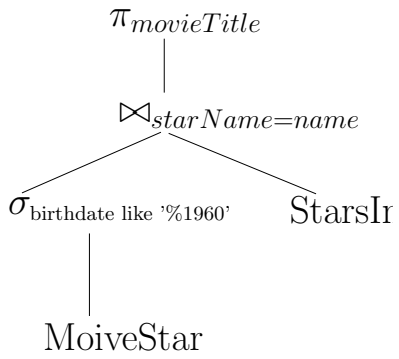
```
SELECT movieTitle
FROM StarsIn, MovieStar
WHERE starName = Name and birthdate LIKE '%1960';
```

Write an efficient relational algebra expression that is the equivalent to this query and give an evaluation plan for this expression (choose the algorithm to implement each operation). Note that you do not have any additional information other than the join is a many-to-one foreign key join. Justify your answer with brief explanation.

Solution. Relational algebra:

$$\pi_{movieTitle}((\sigma_{\text{birthDate like '\%1960'}} MovieStar) \bowtie_{starName=name} StarsIn)$$

An evaluation plan is shown in the following figure:



For the selection operation we can use sequential scan unless we assume that the MovieStar relation is sorted according to birthDate (or there is B+tree index on this attribute). For the join operation, hash join is a good choice in practice when no extra information is given. If the set of tuples selected from MovieStar is sufficiently small, a block nested loop join would also be a good choice with StarsIn as the inner relation.

3. Consider $R \bowtie S$ where the join predicate is $R.a = S.b$, given the following metadata about R and S:

- Relation R contains 2,000 tuples and has 10 tuples per block
- Relation S contains 10,000 tuples and has 10 tuples per block
- Attribute a of relation R is the primary key for R, and every tuple in R matches 5 tuples in S
- There exists a primary index on $S.b$ with height 3
- There exists a secondary index on $R.a$ with height 2
- The buffer can hold 5 blocks

Answer the following questions:

(1) If $R \bowtie S$ is evaluated with a block nested loop join, which relation should be the outer relation? Justify your answer. What is the cost of the join in number of I/O's?
(2) If $R \bowtie S$ is evaluated with an index nested loop join, what will be the cost of the join in number of I/O's? Show your analysis.
(3) What is the cost of a plan that evaluates this query using sort-merge join. Show the details of your cost analysis.
(4) If the buffer can hold 202 blocks rather than 5, which of your answers for 1-3 change (if any)? Explain with new cost analyses.
(5) Evaluate the cost of computing the $R \bowtie S$ using hash join assuming: i) The buffer can hold 202 blocks, ii) The buffer can hold 5 blocks.

Solution. Some basic numbers to refer across this question:

- Number of pages in S: $b_s = \frac{10{,}000}{10} = 1000$
- Number of pages in R: $b_r = \frac{2000}{10} = 200$
- Number of pages in buffer: $M' = M/B = 5$

(1) R as outer relation: Cost = $\lceil \frac{b_r}{M'-2} \rceil b_s + b_r = \lceil \frac{200}{3} \rceil 1000 + 200 = 67,200\ IOs$
S as outer relation: Cost = $\lceil \frac{b_s}{M'-2} \rceil b_r + b_s = \lceil \frac{1000}{3} \rceil 200 + 1000 = 67,800\ IOs$
Hence, choose R as outer relation.

(2) We check both relations to determine the best outer relation,

1. R as outer relation: Cost = $b_r + n_r \times c$ where c is the cost of getting all matches from index on $S.b$.
   $c = Height + \lceil \frac{SC(b,S)}{f_s} \rceil = 3 + \lceil \frac{5}{10} \rceil = 4$
   Cost = $200 + 2000 \times 4 = 8200$
2. S as outer relation: Cost = $b_s + n_s \times c$ where c is the cost of getting all matches from index on $R.a$.
   $c = Height + \lceil \frac{SC(a,R)}{f_r} \rceil = 2 + 1 = 3$
   Cost = $1,000 + 10,000 \times 3 = 31,000$

Clearly choose R as outer relation with cost 8200 IOs.

(3) Since there exists a primary index on $S.b$, we know that tuples in S are already sorted on b. We only need to sort R on a.
Total cost = $b_r + b_s + SortingCost(R)$
$SortingCost(R) = 2b_r \left[\lceil \log_{M'-1}(\frac{b_r}{M'}) \rceil + 1 \right] = 2 \times 200 \times \left[\lceil \log_4(\frac{200}{5}) \rceil + 1 \right] = 1,600$
Total cost = $200 + 1000 + 1600 = 2800\ IOs$

(4) The costs for the above problems will changes as follows with $M' = 202$,

1. Cost = $\lceil \frac{b_r}{M'-2} \rceil b_s + b_r = \lceil \frac{200}{200} \rceil 1000 + 200 = 1,200\ IOs$
2. The cost remains unchanged.
3. $SortingCost(R) = b_r = 200\ IOs$ since $b_r < M'$, we can perform sorting in memory.
   Total cost = $200 + 1000 + 200 = 1400\ IOs$

(5) Here we need to check for both S and R as build and probe relations.

1. $M' = 202$:
   - R is the probe relation:
     Check if $\frac{b_s}{M'-2} \leq M' - 1 \Rightarrow \frac{1000}{200} < 200$ is true. We don't need recursive partitioning and hence cost is given by,
     $Cost = 3(b_r + b_s)$
     $Cost = 3(1000 + 200) = 3600\ IOs$
     P.s. Each partition in probing phase should reserve at least one page for output, and one page for the probing relation. Thus, the max amount of pages for a partition after the building phase should be $M'-2$. Further, we can do at most $M'-1$ way partitioning every time during building phase. Thus, we need to check if $\frac{b_s}{M'-2} \leq M' - 1$
   - S is the probe relation:
     Check if $\frac{b_r}{M'-2} \leq M' - 1 \Rightarrow \frac{200}{200} < 200$ is true. We don't need recursive partitioning and hence cost is given by,
     $Cost = 3(b_r + b_s)$
     $Cost = 3(1000 + 200) = 3600\ IOs$

   Hence choose S (either one is fine) as probe with cost $3600\ IOs$.
2. $M' = 5$:
   - R is the probe relation:
     Check if $\frac{b_s}{M'-2} < M' \Rightarrow \frac{1000}{3} < 5$ is false, We need recursive partitioning and the cost is given by,
     $Cost = 2(b_r + b_s) \times \lceil \log_{M'-1}(\frac{b_s}{M-2}) \rceil + b_r + b_s = 2(200+1000)\lceil \log_4(1000) - 1 \rceil + 200 + 1000 = 10,800$
   - S is the probe relation:
     Check if $\frac{b_r}{M'-2} < M' \Rightarrow \frac{200}{3} < 5$ is false, We need recursive partitioning and the cost is given by,
     $Cost = 2(b_r + b_s) \times \lceil \log_{M'-1}(\frac{b_r}{M-2}) \rceil + b_r + b_s = 2(200+1000)\lceil \log_4(200) - 1 \rceil + 200 + 1000 = 8,400$

   Hence choose S as probe with cost $8,400\ IOs$.

4. Consider the following schema:

Sailors(sid, sname, rating, age)
Reserve(sid, did, day)
Boats(bid, bname, size)

Reserve.sid is a foreign key to Sailors and Reserves.bid is a foreign key to Boats.bid. We are given the following information about the database:

- Reserves contains 10,000 records with 40 records per page.
- Sailors contains 1000 records with 20 records per page.
- Boats contains 100 records with 10 records per page.
- There are 50 values for Reserves.bid.
- There are 10 values for Sailors.rating(1..10).
- There are 10 values for Boat.size
- There are 500 values for Reserves.day.

Consider the following query:

```
SELECT S.sid, S.sname, B.bname
FROM Sailors S, Reserves R, Boats B
WHERE S.sid=R.sid AND R.bid = B.bid AND B.size>5 AND
R.day='July 4, 2003'
```

(1) Assuming uniform distribution of values and column independence, estimate the number of tuples returned by this query.

Consider the following query:

```
SELECT S.sid, S.sname, B.bname
FROM Sailors S, Reserves R, Boats B
WHERE S.sid=R.sid AND R.bid = B.bid
```

(2) Draw all possible left-deep query plans for this query.

(3) For the first join in each query plan (the one at the bottom of the tree), what join algorithm would work best? Assume that you have 50 pages of memory. There are no indexes, so indexed nested loop is not an option. You must consider all possible join algorithms.

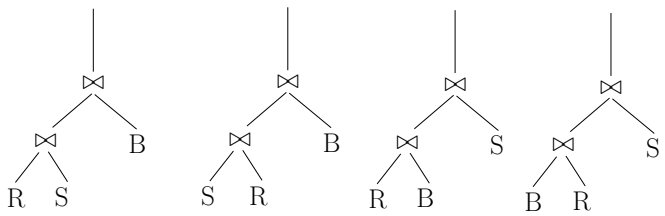
Solution. The answer for each questions shows as below:

(1) The maximum carnality of this query is $|R| \times |S| \times |B| = 10^9$. And the reduction factors are:

$1/2$ for B.size>5
$1/500$ for R.day='July 4, 2003'
$1/100$ for R.bid =B.bid, and
$1/1000$ for S.sid=R.sid

So number of tuples returned is $\frac{1}{2} \cdot \frac{1}{500} \cdot \frac{1}{100} \cdot \frac{1}{1000} \cdot 10^9 = 10$.

(2) There are 4 possible left-deep query plans shown below. *Note that we rule out $S \times B$ as the first join since it is a Cartesian Product that should not be considered.*



(3) We have 50 pages of memory. There are no indexes, so we don't have to consider index nested loop join. And we know $|R| = 250$, $|S| = 50$, $|B| = 10$. The cost of first join for each query plan is shown as below:

R join S:

- page nested loop join: $|R| + |R| \times |S| = 250 + 250 \times 50 = 12570$
- block nested loop join: $|R| + \lceil |R|/48 \rceil \times |S| = 250 + 6 \times 50 = 550$
- sort-merge join: note that $\sqrt{250} \leq 50$, so external sorting can be done in 2 passes. Hence, the cost is $3(|R| + |S|) = 3(250 + 50) = 900$
- hash join: we can use hash join since each of R's partitions from phase one (the partition phase) fits into the memory. So the cost is $3(|R| + |S|) = 900$
- the best choice is block nested loop join.

S join R:

- page nested loop join: $|S| + |S| \times |R| = 50 + 50 \times 250 = 12550$
- block nested loop join: $|S| + \lceil |S|/48 \rceil \times |R| = 50 + 2 \times 250 = 550$
- sort-merge join: note that $\sqrt{250} \leq 50$, so external sorting can be done in 2 passes. Hence, the cost is $3(|R| + |S|) = 3(250 + 50) = 900$
- hash join: we can use hash join since each of S's partitions from phase one (the partition phase) fits into the memory. So the cost is $3(|R| + |S|) = 900$
- the best choice is block nested loop join

B join R:

- page nested loop join: $|B| + |B| \times |R| = 10 + 10 \times 250 = 2510$
- block nested loop join: $|B| + \lceil |B|/48 \rceil \times |R| = 10 + 250 = 260$
- sort-merge join: note that $\sqrt{250} \leq 50$, so external sorting can be done in 2 passes. Hence, the cost is $3(|R| + |S|) = 3(10 + 250) = 780$
- hash join: we can use hash join since each of B's partitions from phase one (the partition phase) fits into the memory. So the cost is $3(|B| + |R|) = 780$
- the best choice is block nested loop join

R join B:

- page-oriented nested loop join: The important thing to note is that we can fit the entire Boats relation into memory, so even though theoretically speaking we need to scan B 250 times when using R as the outer relation, but in this case we don't. As a result, the cost is only $|R| + |B| = 10 + 250 = 260$
- block nested loop join: $|R| + \lceil |R|/48 \rceil \times |B| = 250 + 6 \times 10 = 310$
- note that $\sqrt{250} \leq 50$, so external sorting can be done in 2 passes. Hence, the cost is $3(|R| + |S|) = 3(10 + 250) = 780$
- hash join: we can use hash join since each of R's partitions from phase one (the partition phase) fits into the memory. So the cost is $3(|B| + |R|) = 780$
- the best choice is block nested loop join

Part II. Concurrency Control & Recovery

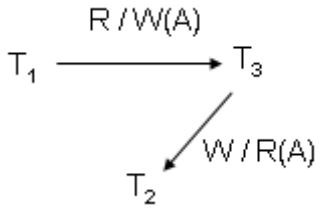
1. In the following schedules, $R_i(A)$ stands for a Read(A) operation by transaction i and $W_i(A)$ stands for a Write(A) operation by transaction i. For each of the following schedules show if it is conflict-serializable and give a conflict-equivalent serial schedule if it is one. Hint: use its dependency graph.

1. $R_1(A), R_2(B), W_3(A), R_2(A), R_1(B), C(T_1), C(T_2), C(T_3)$
2. $R_1(A), R_2(B), W_1(A), R_3(C), W_2(B), W_3(C), R_4(D), R_4(A), W_4(D), C(T_1), C(T_2), C(T_3), C(T_4)$
3. $R_3(E), R_1(D), W_2(C), W_3(A), R_1(E), W_4(B), R_1(B), W_3(E), R_4(A), W_4(C), C(T_1), C(T_2), C(T_3), C(T_4)$

Solution. The answer to each question shows as below:

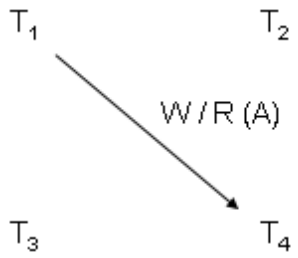
1. Yes, since there are no cycles as shown in the figure below. The serial schedule is $< T_1, T_3, T_2 >$:

T_1	T_2	T_3
R(A)		
	R(B)	
		W(A)
	R(A)	
R(B)		



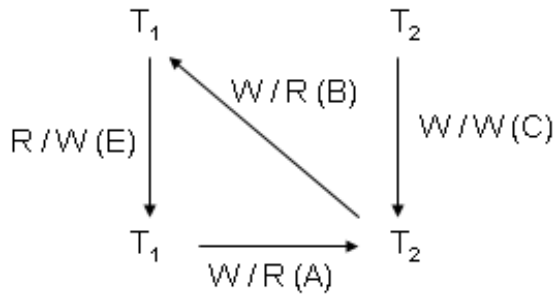
2. Yes, since there are no cycles in the figure below. The serial schedule is $< T_1, T_2, T_3, T_4 >$.

T_1	T_2	T_3	T_4
R(A)			
	R(B)		
W(A)			
		R(C)	
	W(B)		
		W(C)	
			R(D)
			R(A)
			W(D)



3. No, since is a cycle in the figure below.

T_1	T_2	T_3	T_4
		R(E)	
R(D)			
	W(C)		
		W(A)	
R(E)			
			W(B)
R(B)			
		W(E)	
			R(A)
			W(C)



2. For the following 2 schedules, show if each is allowed in SS2PL, and if not, what happens.

An example is given: (the schedule will not be allowed in SS2PL as it is).

$S_1(A), R_1(A), X_2(A), W_2(A), X_1(B), R_1(B), W_1(B), C(T_1), X_2(B), W_2(B), C(T_2)$

T_1	T_2
S(A)	
R(A)	
	X(A), Blocked
X(B)	
R(B)	
W(B)	
Commit	
Release S(A)	
	W(A)
	X(B), Blocked
Release X(B)	
	W(B)
	Commit+Release its locks

OR

T_1	T_2
S(A)	
R(A)	
	X(A), Blocked
X(B)	
R(B)	
W(B)	
Commit	
Release S(A)	
Release X(B)	
	W(A)
	X(B)
	W(B)
	Commit+Release its locks

(1) $S_1(A), R_1(A), S_2(B), R_2(B), S_3(C), R_3(C), X_3(D), W_3(D), C(T_3), X_2(C), W_2(C), C(T_2), X_1(B), W_1(B), C(T_1)$

(2) $X_1(A), R_1(A), X_2(B), R_2(B), X_3(C), R_3(C), S_1(B), R_1(B), S_2(C), R_2(C), S_3(A), R_3(A), W_1(A), C(T_1), W_2(B), C(T_2), W_3(C), C(T_3)$

Solution. For each question, the answers are shown as below:

(1) This schedule is allowed as shown in the figure below:

T_1	T_2	T_3
S(A)		
R(A)		
	S(B)	
	R(B)	
		S(C)
		R(C)
		X(D)
		W(D)
		Commit
		Release X(D)
		Release S(C)
	X(C)	
	W(C)	
	Commit	
	Release X(C)	
	Release S(B)	
X(B)		
W(B)		
Commit		
Release X(B)		
Release S(A)		

(2) A schedule with locks is shown as below. The transactions will go into a deadlock.

T_1	T_2	T_3
X(A)		
R(A)		
	X(B)	
	R(B)	
		X(C)
		R(C)
S(B), Blocked		
	S(C), Blocked	
		S(A), Blocked
Deadlock	Deadlock	Deadlock

3. For the following schedules, show what happens in each case.

An example of non-strict 2PL is given (There could be even more, but showing one is enough):

$S_1(A), R_1(A), X_1(B), X_2(A), W_2(A), R_1(B), W_1(B), T1.Commit, X_2(B), W_2(B), T2.Commit$

T_1	T_2
S(A)	
R(A)	
X(B)	
	X(A), Blocked
Release S(A)	
	W(A)
R(B)	
W(B)	
Release X(B)	
Commit	
	X(B)
	W(B)
	Release X(A), Release X(B)
	Commit

OR

T_1	T_2
S(A)	
R(A)	
X(B)	
Release S(A)	
	X(A)
	W(A)
R(B)	
W(B)	
Release X(B)	
Commit	
	X(B)
	Release X(A)
	W(B)
	Release X(B)
	Commit

(1) $X_1(B), W_1(B), X_2(A), W_2(A), S_2(B), R_2(B), S_1(A), R_1(A), C(T_1), C(T_2)$. (Using **strict 2PL**)
(2) $X_1(B), W_1(B), X_2(A), W_2(A), S_2(B), R_2(B), S_1(A), R_1(A), C(T_1), C(T_2)$. (Using **non-strict 2PL**)
(3) $X_1(B), W_1(B), S_1(A), S_2(B), R_2(B), R_1(A), X_2(A), W_2(A), C(T_1), C(T_2)$. (Using **non-strict 2PL**)

Solution. The answer for each question is shown as below:

(1) This schedule is not allowed.

T_1	T_2
X(B)	
W(B)	
	X(A)
	W(A)
	S(B), Blocked
S(A), Blocked	
Deadlock	Deadlock

(2) No difference from (1), same as above. Since no one can release lock early to get rid of the deadlock, as they both still need to acquire lock(s) down the road.

T_1	T_2
X(B)	
W(B)	
	X(A)
	W(A)
	S(B), Blocked
S(A), Blocked	
Deadlock	Deadlock

(3) This schedule is allowed in non-strict 2PL as shown the figure below.

T_1	T_2
X(B)	
W(B)	
S(A)	
Release X(B)	
	S(B)
	R(B)
R(A)	
Release S(A)	
	X(A)
	Release S(B)
	W(A)
	Relase X(A)
Commit	
	Commit

4. Consider the following database, answer questions below:

- Employee-Table(ssn, name, salary)
  - $E_1(132, Smith, 20K)$
  - $E_2(456, Kelley, 40K)$
  - $E_3(678, Johnson, 400K)$
  - $E_4(792, Preeston, 40K)$
  - $E_5(865, Johnson, 60K)$ ...
- DPT-table(dnumber, dname, budget)
  - $D_1(1, Marketing, 1M)$
  - $D_2(2, Engineering, 2M)$
  - $D_3(3, R\&D, 4M)$
  - $D_4(4, HR, 1M)$ ...

(1) Consider the following schedules, explain if they are valid in serializable isolation level and repeatable read isolation level.

(a) $R_1(E_1), W_1(E_1), R_2(E_2), W_2(E_2), R_1(D_1), C(T_1), C(T_2)$

(b) $R_1(E_1), W_1(E_1), R_2(E_2), W_2(E_2), W_2(E_1), R_1(E_1), C(T_1), C(T_2)$

(c) $R_1(E$ where $salary > 40K$ and $salary < 100k), \text{Insert}_2(\text{into E}, (999, Bob, 50k)), C(T_2),$
$R_1(E$ where $salary > 40k$ and $salary < 100k), C(T_1)$

(d) $R_1(E$ where $salary > 40K$ and $salary < 100k), \text{Insert}_2(\text{into E}, (999, Bob, 50k)),$
$R_1(E$ where $salary > 40k$ and $salary < 100k), C(T_2), C(T_1)$

(e) $R_1(E$ where $salary > 40K$ and $salary < 100k), \text{Update}_2(\text{set } E.name = Alien \text{ where } salary = 60k),$
$C(T_2), R_1(E$ where $salary > 40k$ and $salary < 100k), C(T_1)$

(2) Consider the following scenario:

T_1	T_2
Begin Transaction	
	Begin Transaction
$R(E_1)$	
Update(set E_1.salary=E_1.salary*1.10)	
R(Select * from Employee)	
	Update(set D_4.budget=2M)
	Delete(E_5)
Insert(Into E, (999, Bob, 50k))	
R(Select * from Employee)	
Commit	
System Crash	

(a) Show the content of the table Employee after the system has recovered from the system crash.

(b) Show the results of $R_1(E_1)$, R_1(Select * from Employee) in Line 4 and Line 9, suppose we are in the repeatable read isolation level.

Solution. The answer for each question is shown as below:

(1) The following bullets explains if each schedule satisfy serializable or repeatable read.

(a) It is serializable. Its effect is the same as T_1 then T_2. It is also repeatable read, since there are no dirty reads, and no items have been read multiple times in the same transaction (i.e., no need to check the second condition for repeatable read).

(b) It is NOT serializable. Its effect is not the same to either T_1 then T_2 or T_2 then T_1. In essence, T_1 needs to read the value of E_1 written by T_2. But if we were to execute T_2 first then T_1, T_1 will have to read the value of E_1 written by itself.

It is also clearly NOT reapeatable read, since it has dirty read (the second $R_1(E_1)$) and item read multiple times could have changed value (E_1 read twice by T_1, and in-between has been updated by T_2).

(c) It is NOT serializable, but it is repeatable read. It is not serializable since (i) it is not the same to T_1 then T_2 (in this case, both select statements in T_1 will not be able to read the inserted record by T_2); (ii) and it is not the same to T_2 then T_1 (in this case, both select statements in T_1 will have to read the inserted record by T_2).

It is repeatable read for two reasons. First, there is no dirty read by T_1 (since the second select statement from T_1 will read the newly inserted record by T_2, but it's been committed already).

Secondly, all items read by T_1 **MULTIPLE TIMES** from the two select statements **DO NOT** change values. Note that the newly inserted record by T_2 is only read ONCE by T_1.

(d) It is NOT serializable for the same reason as stated in (c).

It is also NOT repeatable read, because now T_1 has a dirty read (T_2 only commits after the second select statement in T_1).

(e) It is NOT serializable for similar reason as stated in (c) and (d).

It is also NOT repeatable read. It does not have any dirty read, but the record E_5 is read twice by T_1 and it has changed its value in the two times being retrieved by T_1.

(2) The answer for each bullet is shown as below:

(a) The final state of table E is as follows:

- $E_1(132, Smith, 22K)$
- $E_2(456, Kelley, 40K)$
- $E_3(678, Johnson, 400K)$
- $E_4(792, Preeston, 40K)$
- $E_5(865, Johnson, 60K)$
- $E_6(999, Bob, 50K)$ ...

(b) $R_1(E_1)$: 132, Smith, 20K.

R_1(Select * from Employee) in line 4:

- $E_1(132, Smith, 22K)$
- $E_2(456, Kelley, 40K)$
- $E_3(678, Johnson, 400K)$
- $E_4(792, Preeston, 40K)$
- $E_5(865, Johnson, 60K)$ ...

R_1(Select * from Employee) in line 9:

- $E_1(132, Smith, 22K)$
- $E_2(456, Kelley, 40K)$
- $E_3(678, Johnson, 400K)$
- $E_4(792, Preeston, 40K)$
- $E_5(865, Johnson, 60K)$
- $E_6(999, Bob, 50K)$ ...

5. Consider the log below. The records are of the form: (t-id, object-id, old-value, new-value).

Assumptions: (a) The PrevLSN has been omitted (it's easy to figure it out yourself); (b) for simplicity we assume that A, B, C and D each represents a page;

1. (T1, start)
2. (T1, A, 45, 10)
3. (T2, start)
4. (T2, B, 5, 10)
5. (T2, C, 35, 10)
6. (T1, D, 15, 5)
7. (T1, commit)
8. (T3, start)
9. (T3, A, 10, 15)
10. (T2, D, 5, 20)
11. (begin checkpoint, end checkpoint)
12. (T2, commit)
13. (T4, start)
14. (T4, D, 20, 30)
15. (T3, C, 10, 15)
16. (T3, commit)
17. (T4, commit)

(1) What are the values of pages A, B, C and D in the buffer pool **after** recovery? Please specify which transactions have been redone and which transactions have been undone, and you **are not required** to show the details of the intermediate step.

- **(a)** if the system crashes just before line 6 is written to the stable storage?
- **(b)** if the system crashes just before line 10 is written to the stable storage?
- **(c)** if the system crashes just before line 13 is written to the stable storage?
- **(d)** if the system crashes just before line 17 is written to the stable storage?

(2) Assume only the 3rd crash as listed in the (1) has really happened and a recovery has then been performed, and the dirty pages caused by T1 have been flushed to disk before line 8, show the **details** of the Analysis phase (the content of two tables at both the beginning and the end of this phase), REDO phase and UNDO phase (show the contents of ToUnDoList at each step and CLRs to be written to the LOG). You may assume that both the transaction table and dirty page table are empty at the beginning of line 1 of the log. Finally, show the content of log when the recovery has completed.

Solution. The answer for each question is shown as below:

(1) The values for A, B, C, D and redo/undo transactions for each bullets are shown as the following:

- **(a)** Undo T_1 and T_2, $A = 45, B = 5, C = 35, D = 15$.
- **(b)** Redo T_1 and Undo $\{T_2, T_3\}$, $A = 10, B = 5, C = 35, D = 5$.
- **(c)** Redo T_2 and Undo $\{T_3\}$, $A = 10, B = 10, C = 10, D = 20$.
- **(d)** Redo T_3 and Undo $\{T_4\}$, $A = 15, B = 10, C = 15, D = 20$.

(2) Analysis phase:

1. Load the transaction table and dirty page table written at the latest checkpoint (line 11):

(a) Transaction Table

T_{id}	lastLsn	status
T_2	10	Active
T_3	9	Active

(b) Dirty Page Table

P_{id}	recLSN
A	9
B	4
C	5
D	10

Table 1: Loading two tables at the last checkpoint.

Pay special attention why recLSN for A is 9 instead of 2, similarly for D, why it is 10 instead of 6. This is explained by the condition given on T1's commit. All pages caused dirty by T1 are flushed to the disk at the time T1 commits. This will delete entries on A and D from dirty page table. They will later be inserted by T3 (LSN 9) and T2 (LSN 10) respectively.

2. Scan the log records from line 11 till the end of the log (line 12), update two tables accordingly. The final contents of the two tables will be (assuming that all dirty pages caused by T_2 have not made to the disk):

(a) Transaction Table

T_{id}	lastLsn	status
T_3	9	Active

(b) Dirty Page Table

P_{id}	recLSN
A	9
B	4
C	5
D	10

Table 2: Two tables right before the crash.

REDO phase:

Find the smallest recLSN, which is 4, and scan the log forward till the end of log. Apply REDO action when necessary. More specifically,

- LSN 4: REDO, update B from 5 to 10, set pageLSN of B to 4;
- LSN 5: REDO, update C from 35 to 10, set pageLSN of C to 5;
- LSN 6: Don't have to redo. Since LSN (6) $<$ recLsn (9) of A in dirty page table;
- LSN 7: Do nothing;
- LSN 8: Do nothing;
- LSN 9: REDO, update A from 10 to 15, set pageLSN of A to 9;
- LSN 10: REDO, update D from 5 to 20, set pageLSN of D to 10;
- LSN 11: Do nothing
- LSN 12: Do nothing

UNDO phase:

1. construct the initial ToUNDOList= $\{9\}$. Only T3 is active at the time of crash and its lastLSN is 9. This is an easy case to handle.

2. Repeatedly select (and delete) the maximum LSN from the ToUNDOList and check that log record. Undo the action if necessary. Insert the prevLSN of the log record being checked to the ToUNDOList if the prevLSN is not null. Write a CLR to the log. If the log record being checked is a CLR, check its undonextLSN and act accordingly. So:

- ToUNDO= $\{9\}$
- Select 9 and delete it from ToUNDO, check LSN= 9;
- update A to 10. Write CLR: UNDO T3 LSN 9; undonextLSN=8;
- insert 8 (the prevLSN) to ToUNDO;
- Select 8 and delete it from ToUNDO, check LSN== 8
- Since it is a start log record; prevLSN=null. Don't insert anything to ToUNDO; No CLR created; Write T3 End to the log.
- ToUNDO is NULL. Done with ToUNDO.

The LOG:

1. (T1, start)
2. (T1, A, 45, 10)
3. (T2, start)
4. (T2, B, 5, 10)
5. (T2, C, 35, 10)
6. (T1, D, 15, 5)
7. (T1, commit)
8. (T3, start)
9. (T3, A, 10, 15)
10. (T2, D, 5, 20)
11. (begin checkpoint, end checkpoint)
12. (T2, commit)
13. (CLR, UNDO T3 LSN 9, undonextLSN=8)
14. (T3, end)