Solutions to Practice Problems On All Topics

[A Mix of Yes/No Questions]

1. True.

If we use simple sort-merge join, we first do the merge-sort on R,S, and then the join on the two merged lists. For the merge phase, #files generated has to be smaller than #blocks of memory. Say M is #blocks can fit into memory:

$$\#files\ generated\ from\ \mathrm{R} = |R|/M < M,$$
$$\#files\ generated\ from\ \mathrm{S} = |S|/M < M$$
$$M > \sqrt{max(|R|, |S|)}$$

For the join phase, each time we load one block of R and one block of S into memory, so M>1 is sufficient.

If we use efficient merge join, then for merge-join phase, it loads one block from both the sublists of R and sublists of S into memory.

$$\#all\ sublists = \frac{|R|}{M} + \frac{|S|}{M} < M,$$
$$M > \sqrt{|R| + |S|}$$

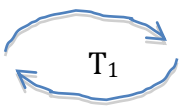
So the minimum memory requirement for the whole process is $\sqrt{|R| + |S|} = \theta(\sqrt{max(|R|, |S|)})$ blocks.

2. False. For Phase1, there are M-1 buckets for R and M-1 buckets for S. For Phase 2, each time it loads bucket Ri (suppose R is the smaller relation) into memory and one block of Si each time to do the join. So the size of one bucket of the smaller relation has to be fitted in memory:

$$\frac{\min\{|R|, |S|\}}{M-1} \leq M, M > \sqrt{\min\{|R|, |S|\}}$$

3. False. Checkpointing could reduce the amount of recovery work for UNDO logging since it reduces the search space for uncompleted transactions. However it does not reduce as significantly as for REDO logging. Because REDO logging without checkpoints has to redo everything from the start while UNDO logging does not need to repeat previous UNDO work.

4. False. For example, $T_1: w_1(Y)w_1(X)$, $T_2: w_2(Y)w_2(X)$, $T_3: w_3(X)$. Say we want the results same as a serial schedule $S = T_1T_2T_3$: X to be last written by T_3 and Y to be last written by T_2. Consider schedule $S_1 = w_1(Y)w_2(Y)w_2(X)w_1(X)w_3(X)$. It is not conflict serializable:



However, the results of S_1 is the same as the serial schedule. Therefore a schedule could meet the consistency constraint but not necessarily conflict serializable.

5.False. Counter example: $T_1 = r_1(A)r_1(B), T_2 = w_2(A)w_2(B)$
$S_1 = r_1(A)w_2(A)w_2(B)r_1(B), S_2 = w_2(A)r_1(A)r_1(B)w_2(B)$

S_1 cannot be transformed into S_2 by a series of swaps on non-conflicting actions. (the order of $r_1(A)w_2(A)$ has to be preserved, so is $w_2(B)r_1(B)$). So they are not conflict equivalent.

6. False. Projection could reduce the length of a tuple. So it is useful to push down projections in joins as it could fit more tuples in memory. However consider projection combined with selection: $\pi_x[\sigma_c(R)] \rightarrow \pi_x\{\sigma_c[\pi_{xz}(R)]\}$ (attribute z is involved in condition c); suppose there is an index on z.

If we don't push down the projection, we could use the index to select tuples fitting the condition and do the projection on each tuple when selected. So it takes only the time of selection, which should be fast. If we do it the other way, pushing down π_{xz} , then we need to first do projection on all tuples of R, which costs more time.

7. False. Sequential IOs are much faster than random IOs since random IOs have seek time and rotational delay while sequential IOs just read the nearest block.

8. True. If we have clustered indexes on both relations on the join attributes, then we can go through the different join attribute values in R and for each value load the blocks of S with the same join condition to produce resulting tuples:

$$B(R) + V(R, attr) \times \frac{B(S)}{V(S, attr)} \approx B(R) + B(S)$$

This is faster than other join techniques.

9. False. It does not makes sense to have a dense second level index since it would be redundant as the information is already in the first level.

10. True:
Associativity: $(R \bowtie S) \bowtie T = R \bowtie (S \bowtie T)$
Let c_{RS} be the join conditions on R and S. Similarly we can define c_{ST} and c_{RT}. Then we can use a series of transforms to reach the right algebra expression.

$$(R \bowtie S) \bowtie T = \left(\sigma_{c_{RS}}(R \times S)\right) \bowtie T \quad \text{//definition of join}$$
$$= \sigma_{c_{ST} \wedge c_{RT}}\left(\sigma_{c_{RS}}(R \times S)\right) \times T \quad \text{//definition of join}$$
$$= \sigma_{c_{ST} \wedge c_{RT} \wedge c_{RS}}\left((R \times S) \times T\right) \quad \text{//selection push-down/pull-up}$$
$$= \sigma_{c_{ST} \wedge c_{RT} \wedge c_{RS}}\left(R \times (S \times T)\right) \quad \text{//associativity of Cartesian}$$

$$= \sigma_{c_{RS} \wedge c_{RT}} \left(R \times \sigma_{c_{ST}} (S \times T) \right) \text{ //selection push-down/pull-up}$$
$$= \sigma_{c_{RS} \wedge c_{RT}} \left(R \times (S \bowtie T) \right) \text{ //definition of join}$$
$$= R \bowtie (S \bowtie T) \text{ //definition of join}$$

Commutativity: $R \bowtie S = S \bowtie R$
Proof: suppose tuple *r* appears *n* times in *R,* tuple *s* appears m times in *S. r* and *s* satisfies the join condition. Then the tuple *t,* which is the combination of attributes in *r* and attributes in *s* appears *nm* times in $R \bowtie S$. It also appears *mn* times in $S \bowtie R$.

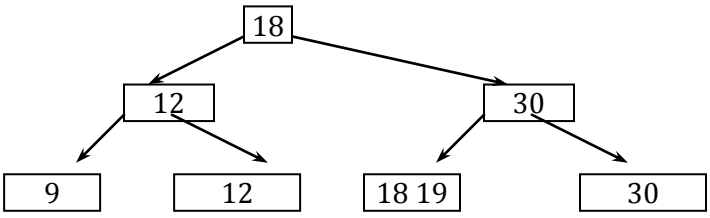
Another way to prove this is to use the commutativity of Cartesian:
$$R \bowtie S = \sigma_{cond1 \wedge cond2 \ldots} R \times S = \sigma_{cond1 \wedge cond2 \ldots} S \times R = S \bowtie R$$

11. False. For A>a, if R is indexed on attribute A, B+ tree could find the sub tree that have leafs with A>a thus efficiently narrows down the search. However for extensible hashing, it uses the last i bits of hash so the order of key A is not preserved during indexing. We have to iterate through all records.

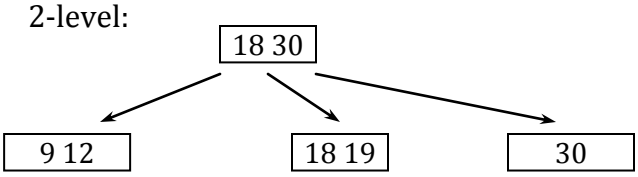
[Data organization]
Not all operations operate on the sequence that R is stored. For example, R(A, B, C) is stored in ascending order of A and we have an index on B. We want to do a selection $\sigma_{R.B=3} R$. Through the index, several accesses are made to the blocks of R and they could spread anywhere on the disk. So random access is better when we read from random places.

Not all operations need to scan from the beginning of records of *R.* For example, we use the same table R as above to do the selection $\sigma_{R.A=100} R$ and only the last record of R has A=100. If we use sequential IO, we have to scan from the beginning of R to the end**.** With random IO, we could jump to the last record using the index.

[Indexing 1]
3-level:



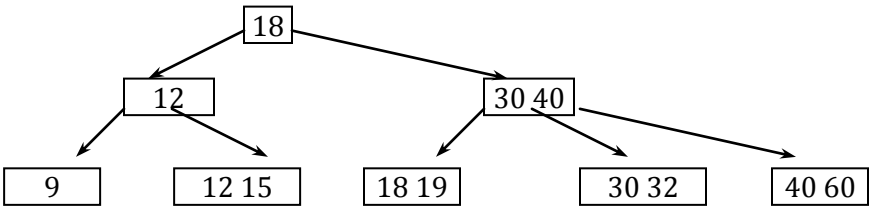
2-level:



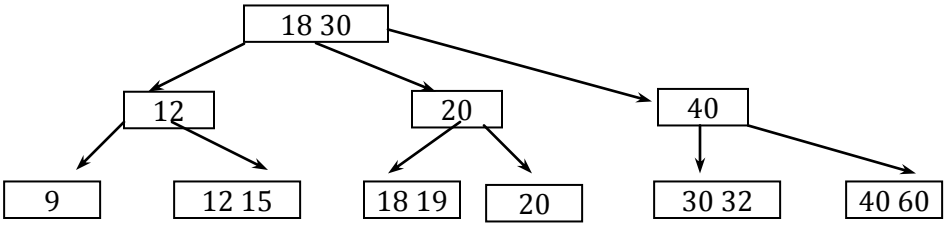
[Indexing 2]

We could use the pair(A,B) as a composite key to build the B+ tree. Define a comparator for pairs using lexicographic ordering: (A,B) < (A',B') if and only if A<A' or A=A' && B<B'. The comparison of individual keys in the composite key is according to the data type.

[Indexing 3]



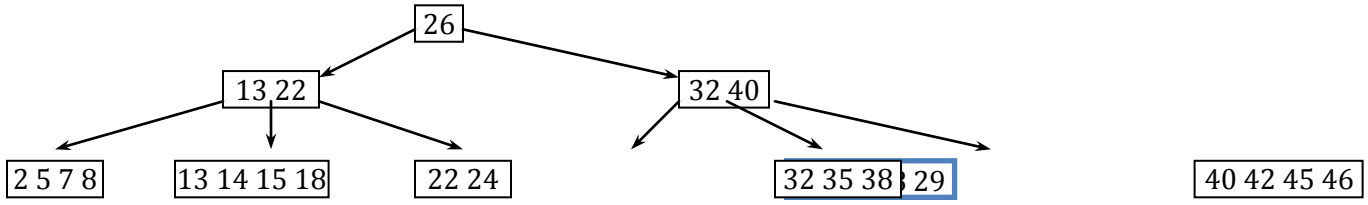
After inserting key 20:



[Indexing 4]

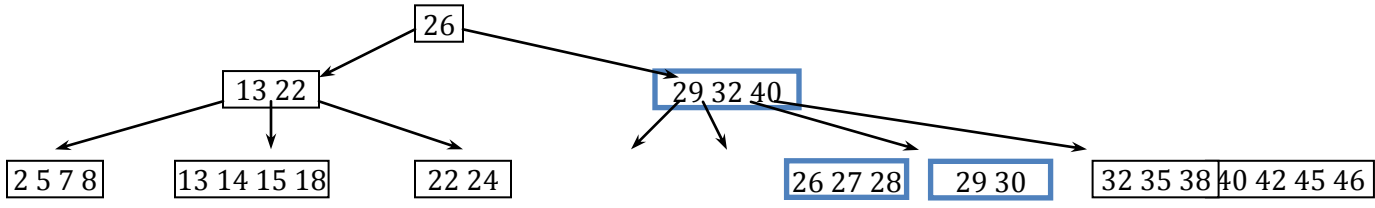
1. Insert an entry with key 29.

Insert directly:



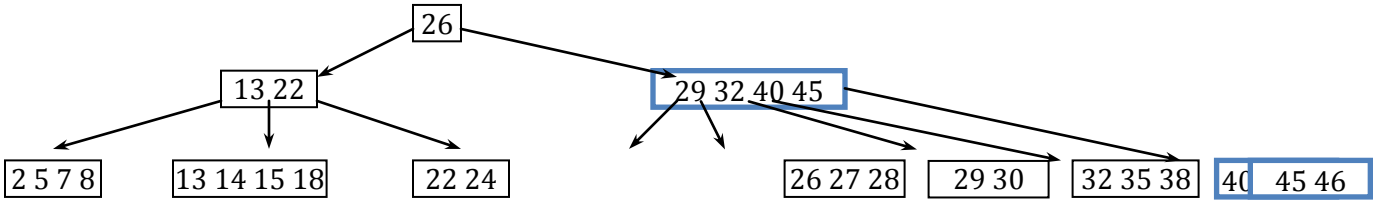
2. Insert an entry with key 30.

Cannot insert directly, split and update the parent:



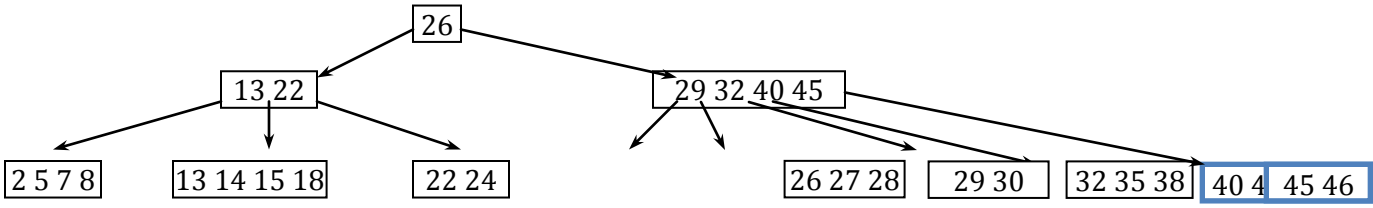
3. Insert 44

Cannot insert directly, split and update the parent:



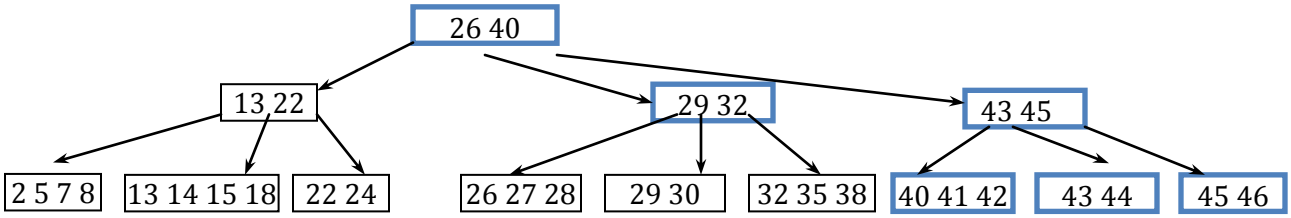
4. Insert 43

Insert directly:



5. Insert 41

Cannot insert directly, split to six leaves. Then split the parent to two nodes and update the root.



[Indexing 5]

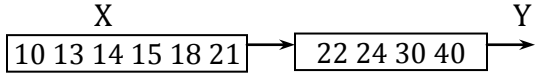
We could do the insertion similar as B+ tree with some variation. Consider splitting a node X: if it has 3m/2+1 elements after insertion, we have to split it. Since m>1, a node must have a sibling, say Y.

Scenario 1. If Y is not full, we could split the elements in X and Y to half and half. (The number of elements in XY is from 5m/2+1 to 3m. Therefore the new X and new Y still satisfies the constraints).

Scenario 2. If Y is full, then we have 3m+1 elements in XY. We could split them to three new nodes with m+1, m and m elements each.

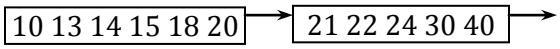
Example: m=4, insert 20.

Scenario 1, split to two new nodes:

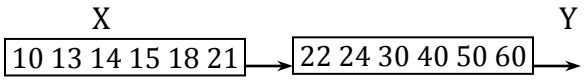


After insertion:

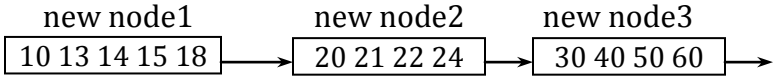
new X new Y



Scenario 2:



After insertion, split to three new nodes with 5, 4, 4 elements:



Pseudo code:
(Notation: K_i is the *i*th key in a node, P_i is the *i*th pointer in a node, |*node*| is the number of elements in *node*)

Insert(root, leftSibling, entry)
If root is a leaf:
    Insert entry in root
    If |*root*| <= *3m/2* then return
    Else begin
        If root has a right sibling, then let Y be that sibling, else let Y be leftSibling
        If Y is not full:
            Split root and Y by keeping the first half of elements in the first new node and the second half in the second new node.
        Else:
            Split root and Y into three new nodes with m+1, m, m elements each.
        Update the parent/ancestor nodes with new pointers/keys. If the number of elements exceeds 3m/2 after update, then do the same thing as above to split.
    End
Else:
    For each element in root, find *i* such that $K_i \leq entry.key < K_{i+1}$.
    Insert(P_{i+1}, P_i, entry)

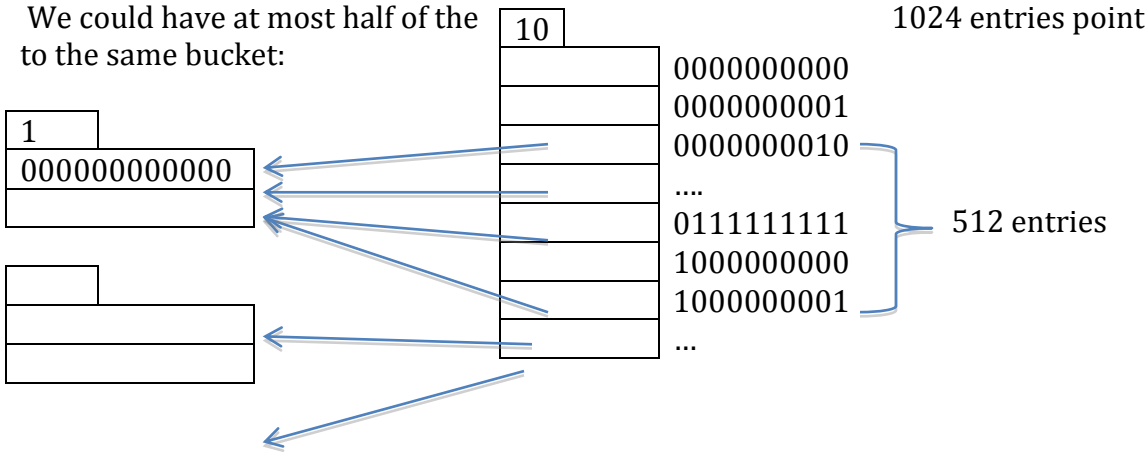
For this algorithm, finding the leaf to insert takes O(depth of tree) = O(logn) time. Insertion is O(1). Updating parent/ancestors also takes O(depth of tree) = O(logn) time.

[Indexing 6]

1. Consider an extensible hash table where 200 buckets are actually allocated at this point. What is the size of the smallest possible directory at this point?

The size of the directory is 2^i. Each bucket has to be pointed to by at least on entry in the directory. Therefore the size has to be at least 200. So we want to have the smallest i possible: $min_i\{2^i \geq 200\}$. $i = 8$, size=256.

2. Consider an extensible hash table whose directory has 1024 entries. What is the largest number of entries that could point to the same bucket?

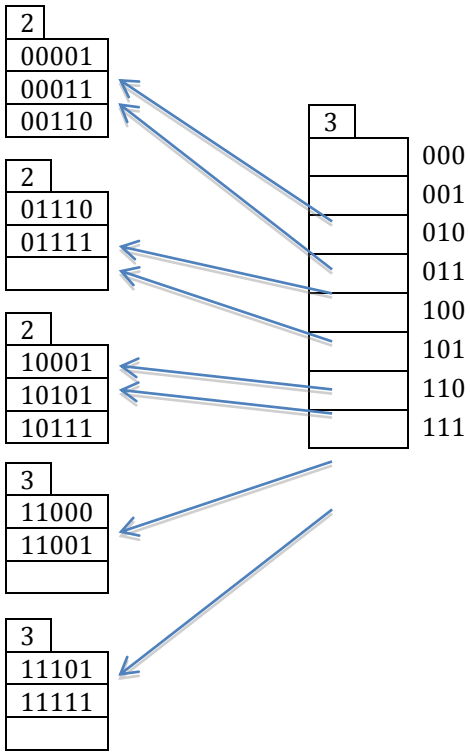


3. Consider a linear hash table with max used bucket 110 (in binary). Which bucket should a key with hash value 111 (binary) be sent to?

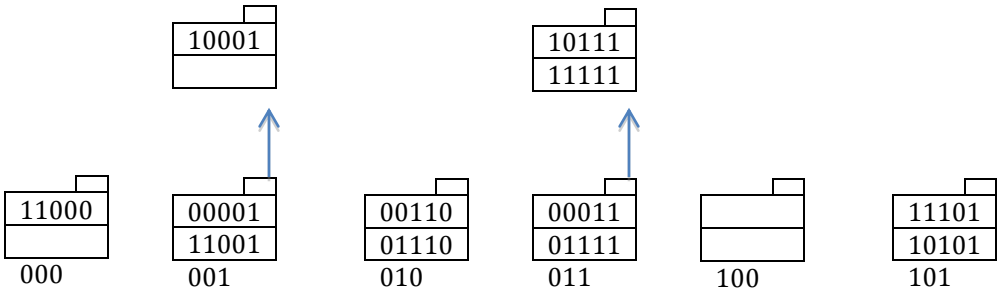
$m = 110_2, i = \lceil log(110_2) \rceil = 3$. 111 is greater than m=110. So we use bucket $h(k)[i] - 2^{i-1} = 111_2 - 2^{3-1} = 111_2 - 100_2 = 011_2$

[Indexing 7]

The structure is the same for both orders:



[Indexing 8]



Provide reasons why we should not keep an index on an attribute A of some relation R.

- A is a large-size attribute. Suppose A is used to store the abstract of a publication, then indexing A would be too expensive.
- A is never used in any selection/join operations. Therefore indexing is not needed.

[Query Processing 1]

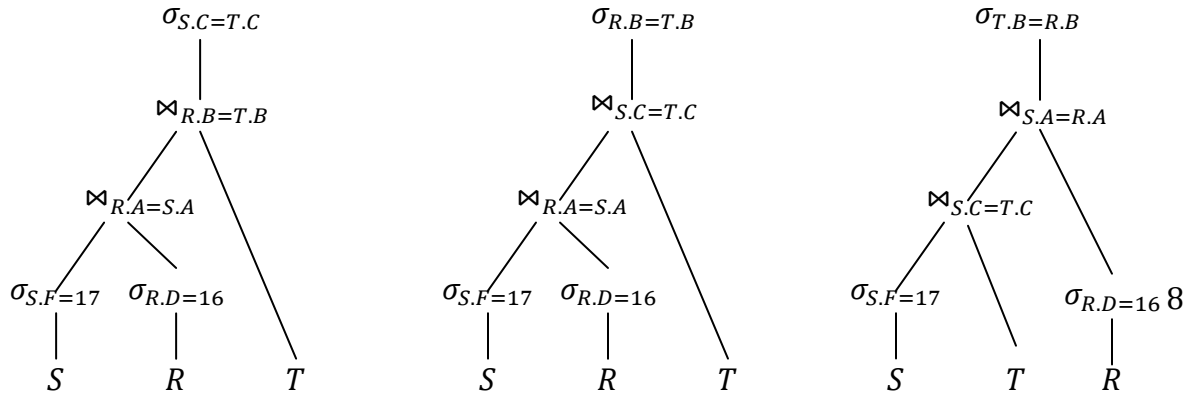
Yes. For example, consider joining R(A,B), S(B,C).
$T(R) = 10000, B(R) = 1000, V(R,B) = 1000$
$T(S) = 10^9, B(S) = 10^7, V(S,B) = 10^9, M = 100$

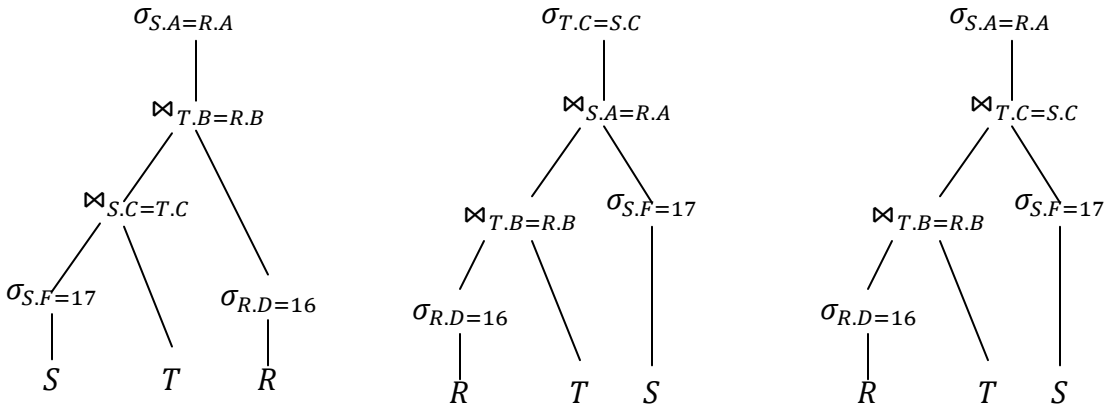
We first build an index on S, we have to read all blocks of S and write the index to disk. So the cost of building index is $B(S) + B(index)$. The cost of index-based join is $B(R) + T(R)\lceil 1 + T(S)/V(S,B)\rceil = 1000 + 10000 \times (1+1) = 21000$. So the total cost is $B(S) + B(index) + 21000 = 1.02 \times 10^7 + B(index)$. Note that the size of index is usually much smaller than the size of the relation. So the total cost is $\sim 1.02 \times 10^7$

Consider using hash join, the cost is one read and one write of R and S to create buckets and another read of buckets of R and S to join: $3B(R) + 3B(S) \approx 3 \times 10^7$
Therefore, temporarily creating an index could be better than using hash join when B(R), T(R) are relatively small, and V(S,B) is large.

[Query Processing 2]

1.





2. Each block has 4096-96=4000bytes of data. So each block has 4000bytes/80bytes=50 tuples. $M = 10^9 bytes/4096 bytes = 244140$
$T(T) = 10^9$, $B(T) = T(T)/50 = 2 \times 10^7$, $B(R) = 2 \times 10^7$, $B(\square) = 2 \times 10^7$
*************I use * to tag the optimal physical sub-plan.******************

Plan1:
$R' = \sigma_{R.D=16} R$.
*INDEX: $T(R') = T(R)/V(D, R) = \mathbf{10^6}$ (tuples with R.D=16 are not clustered together so we cannot count B(R') as cost)
SCAN: read all blocks of R: $B(R) = 2 \times 10^7$

$S' = \sigma_{S.F=17} S$
*INDEX: $T(S') = T(S)/V(F, S) = \mathbf{10^6}$
SCAN: $B(S) = 2 \times 10^7$

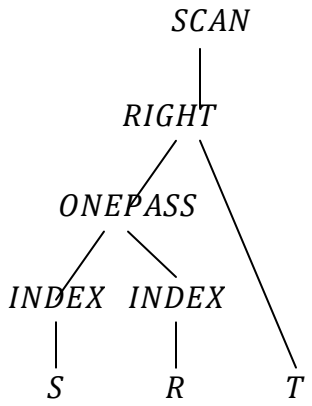
$R' \bowtie S'$:
*ONE-PASS: both can be fit in memory, so cost=**0**
RIGHT-INDEX: not applicable
SORT-MERGE: $2B(R') + 2B(S') = 80000$
$T(R' \bowtie S') = T(R')T(S')/\max\{V(A, R'), V(A, S')\} = 10^3$
$S(R' \bowtie S') = 160$bytes. 25 tuples per block. $B(R' \bowtie S') = \frac{10^3}{25} = 40$

$(R' \bowtie S') \bowtie_{R.B=T.B} T$:
ONE-PASS: left can be fit in memory, it takes $B(T) = 2 \times 10^7$
*RIGHT-INDEX: $T(\text{left})\left(1 + \left\lceil \frac{T(T)}{V(T,B)} \right\rceil\right) = 1000 * 2 = \mathbf{2000}$
SORT-MERGE: $2B(\text{left}) + 3B(T) \approx 6 \times 10^7$
$T((R' \bowtie S') \bowtie_{R.B=T.B} T) = T(\text{left})T(T)/\max\{V(B, \text{left}), V(B, T)\} = 10^3$

$$\mathrm{S}(.) = 240\text{bytes}.\ \mathrm{B}(.) = \frac{10^3}{4000/240} = 60$$

$\sigma_{S.C=T.\square}((R' \bowtie S') \bowtie_{R.B=T.B} T)$
SCAN: do the selection for each tuple as soon as a tuple is generated. Cost=0

So the first plan takes 2000000+2000=**2002000** block accesses.



Consider other plans:

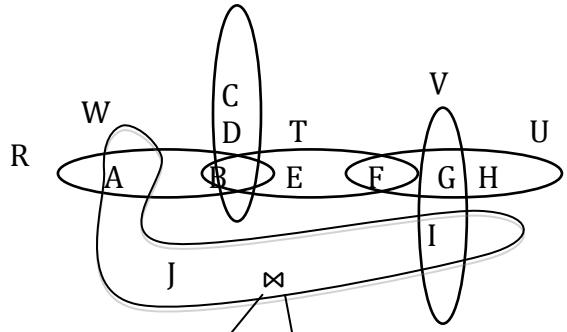
Plan2, join on T.C=S.C first:
$(R' \bowtie S') \bowtie_{\square.C=S.C} T$:
*ONE-PASS: left can be fit in memory, it takes $\mathrm{B(T)} = \mathbf{2 \times 10^7}$
RIGHT-INDEX: not applicable
SORT-MERGE: $2\mathrm{B(left)} + 3\mathrm{B(T)} \approx 6 \times 10^7$
Already exceeds the optimal plan!

Plan 3,4: we do $S' \bowtie T$ first:
*ONE-PASS: S' can be fit in memory, it takes $\mathrm{B(T)} = \mathbf{2 \times 10^7}$
RIGHT-INDEX: note applicable
SORT-MERGE: $2\mathrm{B(left)} + 3\mathrm{B(S')} \approx 6 \times 10^7$
Already exceeds the optimal plan!

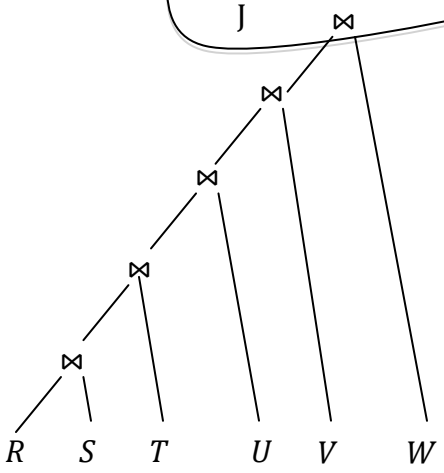
Plan 5,6: we do $R' \bowtie T$ first:
*ONE-PASS: R' can be fit in memory, it takes $\mathrm{B(T)} = \mathbf{2 \times 10^7}$
RIGHT-INDEX: note applicable
SORT-MERGE: $3\mathrm{B(T)} + 2\mathrm{B(R')} \approx 6 \times 10^7$
Already exceeds the optimal plan!
So the first plan is overall the optimal plan.

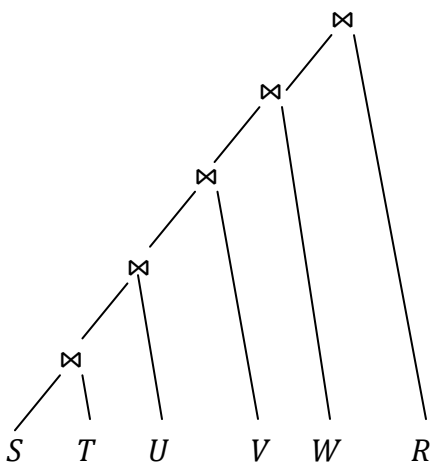
[Query Processing 3]
Hypergraph:

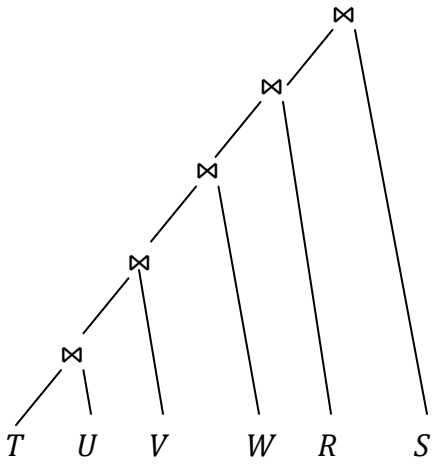
S

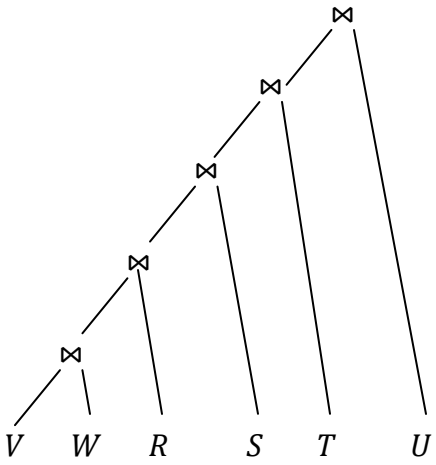


left-deep algebraic expressions:









[Query Processing 4]

$T(R) = 10^6, B(R) = \frac{10^6}{20} = 50000, V(R, B) = 500$

$T(S) = 10^5, B(S) = \frac{10^5}{100} = 1000, V(S, B) = 500$

$M = 1500$

(a)cost of $R \times S$: we can fit all blocks of S in memory. So we can load all S in memory and load one block of R each time to do the Cartesian.

- Pipelining: we do the selection as soon as a tuple is produced.
  $B(R) + B(S) = 50000 + 1000 = 51000$

- No-pipelining: write results back and then read again to do the selection.
  $T(R \times S) = 10^6 \times 10^5 = 10^{11}, B(R \times S) = \frac{T(R \times S)}{1/(1/20+1/100)} = 6 \times 10^9$ (Note that tuple length changed after Cartesian)

  $2B(R \times S) + B(R) + B(S) = 12 \times 10^9 + 51000 \approx 1.2 \times 10^{10}$

(b) Sort-merge join for $SELECT_{A=a}(R\ join\ S)$. Cost of sort: $2B(R) + 2B(S)$. Merge and selection:

- Pipelining: we do the selection as soon as a joined tuple is produced.
  $3B(R) + 3B(S) = 3 * (50000 + 1000) = 155000$
- No-pipelining: write results back and then read again to do the selection.

$$T(R \bowtie S) = 10^6 \times \frac{10^5}{\max\{V(R,B), V(S,B)\}} = 2 \times 10^8,$$

$$B(R \bowtie S) = \frac{T(R \bowtie S)}{1/(1/20 + 1/100)} = 1.2 \times 10^7$$

$$2B(R \bowtie S) + 3B(R) + 3B(S) = 2.4 \times 10^7 + 155000 = 2.42 \times 10^7$$

(c) Assume clustered index for R and S (tuples with same values of B are stored together.) We could load the blocks of R with the same value of B in the memory (B(R)/V(R,B)=100), and fetch blocks of S that match the condition.

$$B(R) + V(R,B) \times \frac{B(S)}{V(S,B)} = B(R) + B(S) = 51000$$

- Pipelining: we do the selection as soon as a joined tuple is produced.
  $B(R) + B(S) = 51000$
- No-pipelining: write results and then read again to do the selection.
  $2B(R \bowtie S) + B(R) + B(S) = 2.4 \times 10^7 + 51000 \approx 2.4 \times 10^7$

[Query Processing 5]

$$PROJECT_{attr}(R - S) = (PROJECT_{attr}R) - (PROJECT_{attr}S)$$

Counter example for sets and sets with duplicates:
Consider these two tables

R	
A	B
3	1
2	2

S	
A	B
3	2
2	2

$PROJECT_A R$
A
3
2

$PROJECT_A S$
A
3
2

$PROJECT_A(R - S)$
A
3

$\neq$

$(PROJECT_A R) - (PROJECT_A S)$
A

[Recovery 1]
UNDO Logging
To solve this, keep in mind the rules of UNDO logging:

- A log record of X is on disk before X is modified on disk.
- All writes on disk before commit is flushed to log.

1. DB(A:5, B:5), LOG(<T,start>) , where the notation means that the persistent part of the DB has the value 5 in A and 5 in B.
Legal: both A and B haven't been changed.

2. DB(A:10, B:5), LOG(<T,start>)
Illegal: log of <T, A, 5> should've been on disk before A changed to 10 on disk.

3. DB(A:10, B:5), LOG(<T,start>, <T,A,5>, <T,B,5>, <T,commit>)
Illegal: all writes should be on disk (DB(A:10, B:10)) before <T, commit> is on disk.

4. DB(A:5, B:10), LOG(<T,start>, <T,A,5>, <T,B,5>)
Legal: log of A is on disk before A is changed on disk.

REDO Logging
To solve this, keep in mind the rules of REDO logging:

- All log records of X (including commit) is on disk before X is modified on disk.
- Flush log at commit.

1. DB(A:10, B:5), LOG(<T,start>, <T, A, 10>)
Illegal: commit should be on disk before A is changed on disk.

2. DB(A:10, B:5), LOG(<T,start>, <T, A, 10>, <T, B, 10>)
Illegal: commit should be on disk before A is changed on disk.

3. DB(A:10, B:5), LOG(<T,start>, <T, A, 10>, <T, B, 10>, <T, commit>)
Legal: every change of A and B logged on disk, commit on disk.

UNDO/REDO Logging
To solve this, keep in mind the rules of UNDO/REDO logging:

- X can be flushed before or after Ti commit.
- Log record of X is on disk before X is modified.
- Flush log at commit.

1. DB(A:10, B:5), LOG(<T,start>, <T, A, 5, 10>)
Legal: log on disk then modification on disk.

2. DB(A:10, B:5), LOG(<T,start>)
Illegal: <T, A, 5, 10> should be on disk before A is modified.

3. DB(A:5, B:10), LOG(<T,start>, <T, A, 5, 10>, <T, B, 5, 10>)
Legal: log on disk before change of A.

4. DB(A:5, B:10), LOG(<T,start>, <T, A, 5, 10>, <T, B, 5, 10>, <T, commit>)
Legal: modification could be on disk after or before commit on disk.

[Recovery 2]
1. What was the initial state of the system before T1 and T2 began executing?
A:5, B:5. According to log record <T1, A, 5>. Initial state of A should be 5. According to <T2, B, 5>, initial state of B should be 5.

2. What will be the state after the recovery?
A:15, B:5. The system will find transactions without <T, commit>: S={T2} and recover according to <T2, …> in reverse order. So A is first written to 15, B is written to 5.

3. Repeat questions (1) and (2) assuming it is a REDO log.
Q1: A: unknown, B: 10. Since in REDO logging, only the new value is recorded, there is no way to know the overwritten value of A.
Q2: A: 5, B: 10. During REDO logging, the system finds transactions that have <T, commit> in log: S={T1} and then brings data back to new values according to <T1, …>. So A is modified to 5 and B stays 10.

[Recovery 3]
Initial state: DB(A: 5, B: 5).
T: A=A+5; B=B+5;
1. Undo logging

T	MEM:A	DB: A	MEM:B	DB:B	log
					<T, Start>
read(A, t); t←t+5;	5	5	5	5	
write(A,t);	10	5	5	5	
read(B, t); t←t+5;	10	5	5	5	
write(B,t);	10	5	10	5	
Output(A)	10	10	10	5	
CRASH!					

After the crash, DB(A:10, B:5), log(<T, Start>). So during UNDO recovery, A won't be recovered to its initial value. DB remains: DB(A:10, B:5), an inconsistent state.

2. Redo logging

T	MEM:A	DB: A	MEM:B	DB:B	log
					<T, Start>
read(A, t); t←t+5;	5	5	5	5	

write(A,t);	10	5	5	5	<T, A, 5>
read(B, t); t←t+5;	10	5	5	5	
write(B,t);	10	5	10	5	<T, B, 5>
Output(A)	10	10	10	5	
CRASH!					

After the crash, DB(A:10, B:5), log(<T, Start>, <T, A, 5>, <T, B, 5>). So during REDO recovery, T is not eligible for redo since it does not have <T, commit> on record. DB remains: DB(A:10, B:5), an inconsistent state.

3. Undo logging

T	MEM:A	DB: A	MEM:B	DB:B	log
					<T, Start>
read(A, t); t←t+5;	5	5	5	5	
write(A,t);	10	5	5	5	<T, A, 5>
read(B, t); t←t+5;	10	5	5	5	
write(B,t);	10	5	10	5	<T, B, 5>
Output(A)	10	10	10	5	
					<T, commit>
CRASH!					

After the crash, DB(A:10, B:5), log(<T, Start>, <T, A, 5>, <T, B, 5>, <T, commit>). So during UNDO recovery, T is not eligible for UNDO. A won't be recovered to its initial value. DB remains: DB(A:10, B:5), an inconsistent state.

[Recovery 4]

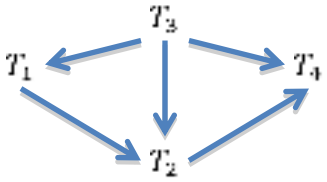
	UNDO	**LOGGING**	**reason**
	possible	**impossible**	
1	□		Log first then modify db.
2		□	WAL
3		□	Writes must be on db before commit flushed to log.
4	□		Log first then modify db.
5		□	Writes on db before commit flushed to log.
6		□	Write B should be on log before commit on log.

	REDO	**LOGGING**	**reason**
	possible	**impossible**	
1		□	Commit has to be on log before modifying db.
2		□	WAL
3	□		Commit on log before modifying db.
4		□	Commit has to be on log before modifying db.
5	□		Commit on log before modifying db.
6		□	Write B should be on log before commit on log.

[Concurrency Control 1]

$S = w_3(E)r_1(D)w_2(C)w_3(A)r_1(E)w_1(B)r_1(B)w_2(E)r_4(A)w_4(C)$

Consider the precedence relation for each value involved:
$E: T_3 < T_1;\ T_3 < T_2; T_1 < T_2, C: T_2 < T_4, A: T_3 < T_4$



This is conflict equivalent to: $S' = T_3T_1T_2T_4$:
$S' = w_3(E)w_3(A)r_1(D)r_1(E)w_1(B)r_1(B)w_2(C)w_2(E)r_4(A)w_4(C)$

[Concurrency Control 2]
We could use the definitions to solve these questions:
2PL: no unlocks during growing phase and no locks during shrinking phase.
Strict 2PL: all locks are released only after transaction ends.
Conflict serializable: conflict equivalent to some serial schedule.
ACR: each transaction may read only those values written by committed transactions.
Strict: each transaction may read and write only those values written by committed transactions.
Serial: transactions are not interleaved.
Deadlock: cycle of transactions waiting for locks to be released by each other.

Relationship:
serial => strict 2PL => strict => ACR,
serial => strict 2PL => strict => 2PL => Conflict Serializable

A: locks and unlocks are interleaved, so it's not 2PL, therefore not strict 2PL or strict or serial. It is also not ACR since there are no commits. It does not always result in a conflict serializable schedule:

$$S = .. OP_1(A)OP_2(B)OP_2(A)OP_1(B) \ldots$$

Since the operations on A/B are wrapped in locks, they could be considered as atomic operation, denoted as OP. In this case it is not conflict serializable.
It is deadlock free: lock on B is acquired after unlock of A in T1; lock on A is acquired after unlock of B in T2. So there cannot be a case where a transaction tries to acquire lock when already holding lock on another object.

B: unlocks after commit, so it is strict 2PL. So it is also strict, 2PL, conflict serializable and ACR. It does not always result in a serial schedule: $S = L_1(C)T_2$ …rest of T1…
Also there could be a deadlock:

$$S' = L_1(C)..L_1(A)L_2(B) \ldots L_2(A) \ldots L_1(B)$$

T2 waits for T1 to release lock on A while T1 waits for T2 to release lock on B.

C: unlocks after commit, so it is strict 2PL. So it is also strict, 2PL, conflict serializable and ACR. It does not always result in a serial schedule: $S = L_1(C)T_2$ …rest of T1…

Also there could be a deadlock:

$$S = L_1(C)L_1(A)L_2(B) \dots L_2(A) \dots L_1(B)$$

T2 waits for T1 to release lock on A while T1 waits for T2 to release lock on B.

D: T1=$L_1(B) \dots U_1(B)$, T2=$L_2(B) \dots U_2(B)$. So once T1 starts, T2 cannot acquire lock on B until T1 unlocks. This is the same when T2 starts first. Therefore they will result in a serial schedule. So it will also be strict, strict 2PL, 2PL, conflict serializable, ACR, no deadlock.

Question	A	B	C	D
Is 2PL	**N**	**Y**	**Y**	**Y**
Is strict 2PL	**N**	**Y**	**Y**	**Y**
Will always result in conflict serializable schedule	**N**	**Y**	**Y**	**Y**
Will always result in schedule that avoids Cascading rollback	**N**	**Y**	**Y**	**Y**
Will result in a strict schedule	**N**	**Y**	**Y**	**Y**
Will result in a serial schedule	**N**	**N**	**N**	**Y**
May result in a deadlock	**N**	**Y**	**Y**	**N**

[Concurrency Control 3]

(a) Consider all possible serial schedules: S=T0T1: A=0, B=1. S'=T1T0: A=1, B=0. Therefore every serial execution involving these two transactions meets the consistency requirement.

(b) As long as $r_0(A)$ happens before $w_1(A)$ and $r_1(B)$ happens before $w_0(B)$, T1 and T2 would read the initial values of A,B. Therefore both of them would enter the if statement resulting in A=1, B=1.

An example:

T0	T1	Db:A	Db:B
read(A)		0	0
read(B)		0	0
	read(B)	0	0
	read(A)	0	0
if (A = 0) then		0	0
B = B + 1;			
	if (B = 0) then	0	0
	A = A + 1;		
write(B);		0	1
	write(A);	1	1

(c) No. As discussed above, as long as $r_0(A)$ happens before $w_1(A)$ and $r_1(B)$ happens before $w_0(B)$, it would not be serializable. Therefore to make it serializable, we have to schedule $r_0(A)$ after $w_1(A)$ or $r_1(B)$ after $w_0(B)$. However, both would

result in a serial schedule ($S = T_1T_0, S' = T_0T_1$ respectively) rather than a concurrent one.

[Concurrency Control 4]
1. w1(A),w2(B),w3(C),r3(A),r3(B)
Conflict serializable: $T_1T_2T_3$

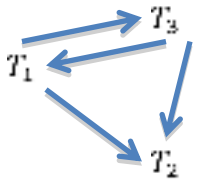
2. w1(A),w2(B),w3(C),r3(A)r3(B)
Conflict serializable: $T_1T_2T_3$

3. r1(A),w2(A),w2(B),r1(B)
Not conflict serializable: r1(A)<w2(A) => $T_1 < T_2$; w2(B)<r1(B) => $T_2 < T_1$

4. w1(A),w1(B),r2(A),w2(B),w1(C),r3(B),w3(C)
Conflict serializable: $T_1T_2T_3$
We can simply move w1(C) before r2(A) since there is no operation on C in between.

[Concurrency Control 5]
w3(B),w3(A),w1(B),r2(B),r2(A),w1(A)
Not conflict serializable: w1(B)<r2(B) => $T_1 < T_2$; r2(A)<w1(A) => $T_2 < T_1$

[Concurrency Control 6]
w1(A)<r2(A) => T1<T2;
w3(B)<w1(B)<w3(B)<r2(B) => T3<T1; T1<T3; T1<T2; T3<T2



It's not conflict serializable as there is a cycle in the precedence graph.

[Concurrency Control 7]
In any serial schedule equivalent to S, T2 precedes T1: so there must exists a transaction T3 such that T2<T3, T3<T1. Since T1 precedes T2 in S and T1 does not precede T2 in a serial schedule equivalent to S, there is no arrow between T1 and T2.

Example:

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

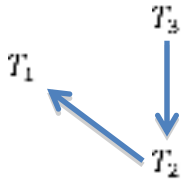
T1 precedes T2 in S. S is conflict equivalent with $S'=T_2T_3T_1$ where $T_2<T_1$.

[Concurrency Control 8]

	Equivalent	Non Equivalent	none	Cannot hold	reason
1	☒				Two reads are not conflict operations
2		☒			Each T writes A once AND S,S' are serial AND S, S' are not identical => S,S' write A in different order => they are not equivalent.
3	☒				P(S)=P(S') and both acyclic => Each of S and S' is equivalent with a same serial schedule S'' (topological ordering on P(s)) => S and S' are equivalent.
4		☒			P(S)≠P(S')(acyclic and cyclic) => they are non-equivalent
5			☒		Suppose S is equivalent with a serial schedule S1, S' is equivalent with a serial schedule S2. S1 could be the same as S2, therefore S, S' are equivalent. However if $S_1 \neq S_2$, then S1 and S2 are non-equivalent. So S, S' are non-equivalent.

[Concurrency Control 9]

Schedule 1. It is serializable since P(S) is acyclic.



So it is serializable with $S' = T_3T_2T_1$

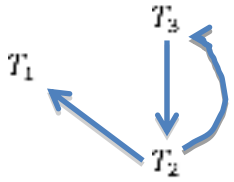
It could also be produced by a 2PL scheduler:

T1	T2	T3
		X(D)
		r(D)
X(A)		
w(A)		
	S(B)	
	r(B)	
		w(D)
		U(D)
	S(D)	
	U(B)	
X(B)		
w(B)		

U(B) U(A)	r(D) U(D)	

It cannot be produced by a Strict 2PL scheduler: we need to put all the unlocks after all reads/writes in a transaction. So we need to put $U_2(B)$ after $r_2(D)$. Therefore T1 has to wait after unlock of B to proceed with $w_1(B)$: $r_2(D) < U_2(B) < X_1(B) < w_1(B)$. So $w_1(B)$ cannot happen before $r_2(D)$. It cannot be produced by a Strict 2PL scheduler.

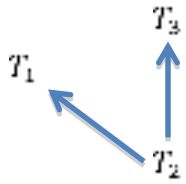
Schedule 2. It is not serializable since P(S) is cyclic.



It is not 2PL or Strict 2PL. For T3, we have to place lock before the first write of D and unlock it after the last write. $X_3(D) < w_3(D) < w_3(D) < U_3(D)$. Therefore $r_2(D)$ cannot happen when T3 is holding the lock. (If we place $S_2(D)$ before T3, T3 cannot execute after $r_2(D)$)**.** Therefore it cannot be produced by a 2PL scheduler or a strict 2PL scheduler. (Strict 2PL is more restrictive).

Another way to think about this is that 2PL => serializablity. So a non-serializable schedule cannot be produced by a 2PL scheduler.

Schedule 3. It is serializable since P(S) is acyclic:



So it is serializable with $S' = T_2T_1T_3$ or $S' = T_2T_3T_1$
It can be produced by a 2PL scheduler:

T1	T2	T3
		S(D) r(D)
X(A) w(A)	S(B) r(B) S(D) U(B)	
X(B) w(B)		
	r(D)	

U(B) U(A)	U(D)	X(D) w(D) U(D)

It cannot be produced by a Strict 2PL scheduler. The reason is the same as schedule1.

	Serializable	2PL	Strict 2PL
1	✓	✓	
2			
3	✓	✓	