COURSES REVIEW

Problems

Assessment

- 15-20 questions, each with 4 answers. Only one answer is correct. (20%)
- 2 problems (30%)
- Laboratory activity/test (50%)

Problem 1

Consider the following two transactions:

T1: *read(A);*
read(B);
if A = 0 then B := B + 1;
write(B).

T2: *read (B);*
read (A);
if B = 0 then A := A + 1;
write(A).

Let the consistency requirement be A = 0 $\vee$ B = 0, with A = B = 0 the initial values.

1. Show that every serial execution involving these two transactions preserves the consistency of the database.

2. Show a concurrent execution of T1 and T2 that produces a non-serializable schedule.

3. Is there a concurrent execution of T1 and T2 that produces a serializable schedule?

Problem 1 - Answer

1. Show that every serial execution involving these two transactions preserves the consistency of the database.

There are two possible executions: {T_1, T_2} and {T_2, T_1}

Case 1:

	A	B
initially	0	0
after T1	0	1
after T2	0	1

Consistency met

Case 2:

	A	B
initially	0	0
after T2	1	0
after T1	1	0

Consistency met

Problem 1 – Answer (cont)

2. Show a concurrent execution of T1 and T2 that produces a non-serializable schedule.

Any interleaving of T1 and T2 results in a non-serializable schedule.

T1	T2
read(A)	
	read(B)
	read(A)
read(B)	
if A = 0 then B = B + 1	
	if B = 0 then A = A + 1
	write(A)
write(B)	

Problem 1 – Answer (cont)

3. Is there a concurrent execution of T1 and T2 that produces a serializable schedule?

From part 1. we know that a serializable schedule results in

$$A = 0 \setminus/ B = 0.$$

Suppose we start with T1 read(A). Then when the schedule ends, no matter when we run the steps of T2, B = 1. Now suppose we start executing T2 prior to completion of T1. Then T2 read(B) will give B a value of 0. So when T2 completes, A = 1. Thus

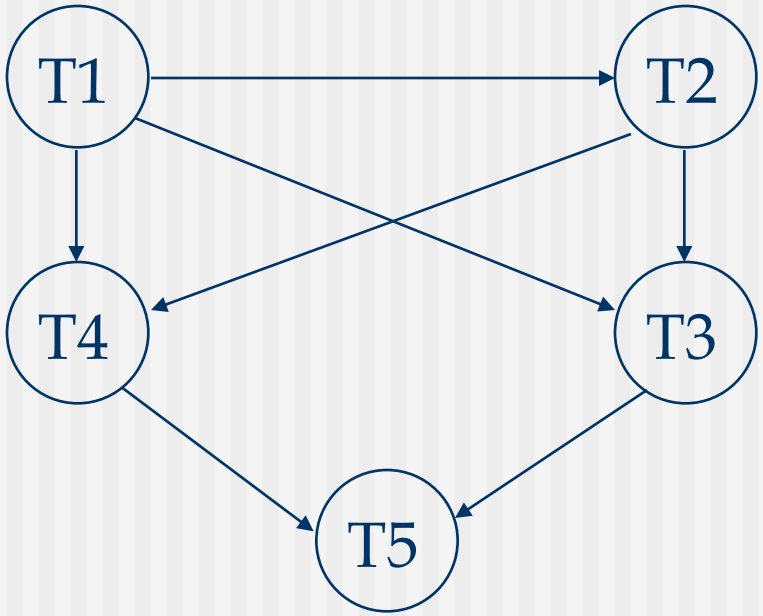
$$B = 1 \wedge A = 1 \Rightarrow \neg(A = 0 \vee B = 0).$$

Similarly for starting with T2 read(B).

There is no parallel execution resulting in a serializable schedule.

Problem 2

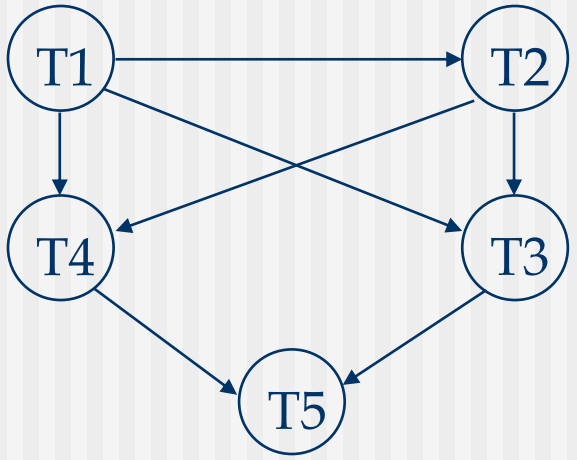
Consider the dependency graph below. Is the corresponding schedule conflict serializable? Explain your answer.



Problem 2 - Answer

There is a serializable schedule corresponding to the dependency graph, since the graph is acyclic.

A possible schedule is obtained by doing a topological sort, that is:
T1, T2, T3, T4, T5



Problem 3

Consider the following two transactions:

T1:	***T2***:
read(A);	*read (B);*
read(B);	*read (A);*
if A = 0 then B := B + 1;	*if B = 0 then A := A + 1;*
write(B).	*write(A).*

1. Add lock and unlock instructions to transactions T1 and T2 so that they observe the two-phase locking protocol.

2. Can the execution of the transactions result in a deadlock?

Problem 3 - Answers

1. Lock and unlock instructions:

T1:	T2:
lock-S(A)	*lock-S(B)*
read(A)	read(B)
lock-X(B)	*lock-X(A)*
read(B)	read(A)
if A = 0 then B := B + 1	if B = 0 then A := A + 1
write(B)	write(A)
unlock(B)	*unlock(A)*
unlock(A)	*unlock(B)*

Problem 3 – Answers (cont)

2. Execution of these transactions can result in deadlock. For example, consider the following partial schedule:

T1	T2
lock-S(A)	
	lock-S(B)
	read(B)
read(A)	
lock-X(B)	
	lock-X(A)

The transactions are now deadlocked.

Problem 4

Given the following log file:

[start_transaction, T1]
[W, T1, D, 20]
[commit, T1]
[checkpoint]
[start_transaction, T2]
[W, T2, B, 12]
[start_transaction, T4]
[W, T4, D, 15]
[start_transaction, T3]
[W, T3, C, 30]
[W, T4, A, 20]
[commit, T4]
[W, T2, D, 25] <- system crash

Suppose the immediate update protocol with checkpointing is used. Describe the recovery process from the system crash. Specify which transactions are rolled back, which operations are redone and which are undone.

Problem 4 - Answer

- T1 committed before the checkpoint, so all its update operations are recorded in the log and stored on disk. No need to redo T1's write operations.
- T4 committed after the checkpoint, so all its update operations are recorded in the log but not stored on disk. Redo T4's update operations from log.
- T2 was active at the time of the crash, therefore, it must be rolled back since it updated database (wrote objects B and D)

Undo its write operations in reverse order

- T3 was active at the time of the crash, therefore, it must be rolled back since it updated database (wrote object C).

Undo its write operation

Problem 5

Consider the join $R \otimes_{R.a = S.b} S$, given the following information about the relations to be joined. What is the cost of joining R and S using a page-oriented nested loops, block nested loops and sort-merge join methods? What is the min number of buffer pages for this cost to remain unchanged? The cost metric is the number of I/O pages (without writing the results).

- relation R contains 10.000 tuples and has 10 tuples per page.
- relation S contains 2.000 tuples and has 10 tuples per page.
- attribute b of relation S is the primary key for S.
- both relations are stored as simple heap files.
- neither relation has any indexes built on it.
- 52 buffer pages are available

Problem 5 - Answer

Let M = 1000 the # of pages in R, N = 200 be # of pages in S and B=52 number of available buffer pages

1. What is the cost of joining R and S using a page-oriented nested loops join? What is the minimum number of buffer pages for this cost to remain unchanged?

Basic idea is to read each page of the outer rel. and for each page scan the inner relation for matching tuples. Total cost would be:

*#PagesInOuter + (#PagesInOuter * #PagesInInner)*

which is minimized by having the smaller relation be the outer relation.

*TotalCost = N + (N*M) = 200.200*

The minimum number of buffer pages for this cost is 3.

Simple Nested Loops Join

foreach tuple r in R do
  foreach tuple s in S do
    if r_i == s_j then add <r, s> to result

- For each tuple in the *outer* relation R, we scan the entire *inner* relation S.
  - Cost: M + p_R * M * N = 1000 + 100*1000*500 I/Os.
- Page-oriented Nested Loops join: For each *page* of R, get each *page* of S, and write out matching pairs of tuples <*r*, *s*>, where *r* is in R-page and *s* is in S-page.
  - Cost: M + M*N = 1000 + 1000*500
  - If smaller relation (S) is outer, cost = 500 + 500*1000

Problem 5 – Answer (cont)

2. What is the cost of joining R and S using a block nested loops join? What is the minimum number of buffer pages for this cost to remain unchanged?

This time read the outer relation in blocks, and for each block scan the inner relation for matching tuples. So the outer relation is still read once, but the inner relation is scanned only once for each outer block, of which there are

$$\lceil \#\text{PagesInOuter} / \text{BlockSize} \rceil = \lceil 200/50 \rceil = 4$$

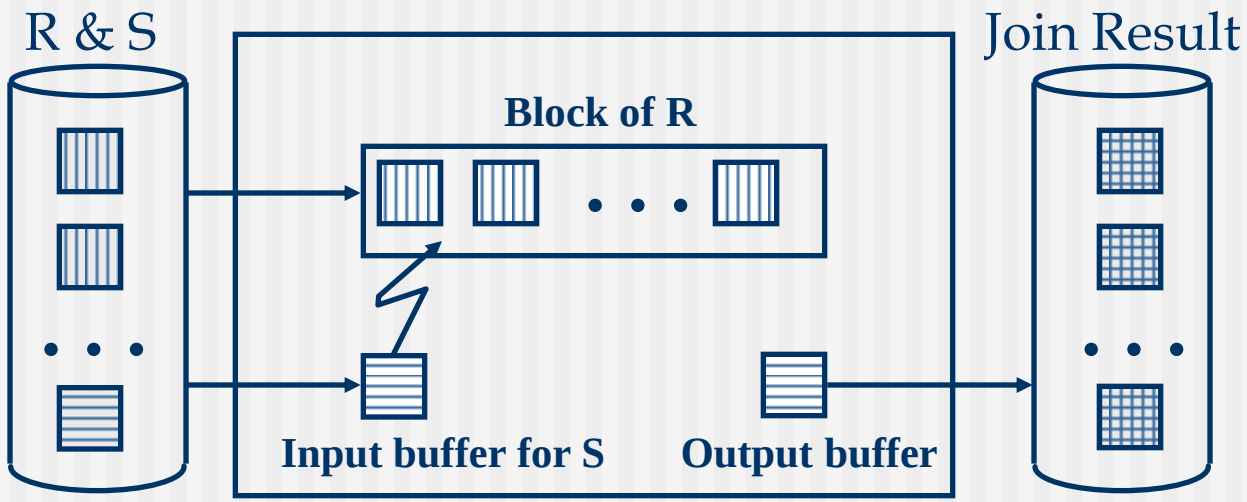
$$\textit{Total Cost} = N + M * \lceil N/(B-2) \rceil = 4.200$$

If the number of buffer pages is < 52*, the number of scans of then inner would be more than 4* ($\lceil 200/50 \rceil = 5$) $\rightarrow$ *the minimum pages for this cost is 52!*

Block Nested Loops Join

- Use one page as an input buffer for scanning the inner S, one page as the output buffer, and use all remaining pages to hold ``block'' of outer R.

  For each matching tuple r in R-block, s in S-page, add <r, s> to result. Then read next R-block, scan S, etc.



Problem 5 – Answer (cont)

3. What is the cost of joining R and S using a sort-merge join? What is the minimum number of buffer pages for this cost to remain unchanged?

Since $B > \sqrt{M} > \sqrt{N}$ *we can use the refinement to Sort-Merge Join.*

$$\textit{Total Cost} = 3 * (M+N) = 3.600$$

The minimum number of buffer page required is 25 $\rightarrow$ *the initial sorting pass will split R into 20 runs (size 50) and split S in 4 runs of size 50 (aprox.). These 24 runs can then be merged in one pass, with one page left over to be used as an output buffer.*

With fewer than 25 buffer pages the number of runs produced by the first pass over both relations would exceed the number of available pages, making one-pass merge impossible

Sort-Merge Join ($R \otimes_{i=j} S$)

- Sort R and S on the join column, then scan them to do a ``merge'' (on join col.), and output result tuples.
  - Advance scan of R until current R-tuple > current S tuple, then advance scan of S until current S-tuple > current R tuple; do this until current R tuple = current S tuple.
  - At this point, all R tuples with same value in R_i (*current R group*) and all S tuples with same value in S_j (*current S group*) *match*; output <r, s> for all pairs of such tuples.
  - Then resume scanning R and S.
- R is scanned once; each S group is scanned once per matching R tuple. (Multiple scans of an S group are likely to find needed pages in buffer.)

Refinement of Sort-Merge Join

- We can combine the merging phases in the *sorting* of R and S with the merging required for the join.
  - With B > $\sqrt{L}$, where L is the size of the larger relation, using the sorting refinement that produces runs of length 2B in Pass 0, number of runs of each relation is < B/2.
  - Allocate 1 page per run of each relation, and `merge' while checking the join condition.
  - Cost: read+write each relation in Pass 0 + read each relation in (only) merging pass (+ writing of result tuples).
- In practice, cost of sort-merge join, like the cost of external sorting, is *linear*.