**CIS*3530 Data Base Systems and Concepts**
Fall 2020
Instructor: Fangju Wang

**Assignment 5 (20%)**
Due: Monday Dec 7, 2020
Submission By: Mitchell Van Braeckel, Student ID: 1002297

---

**Question 1:** (5%)
The following is a schedule of the read/write operations of $T_1$ and $T_2$:

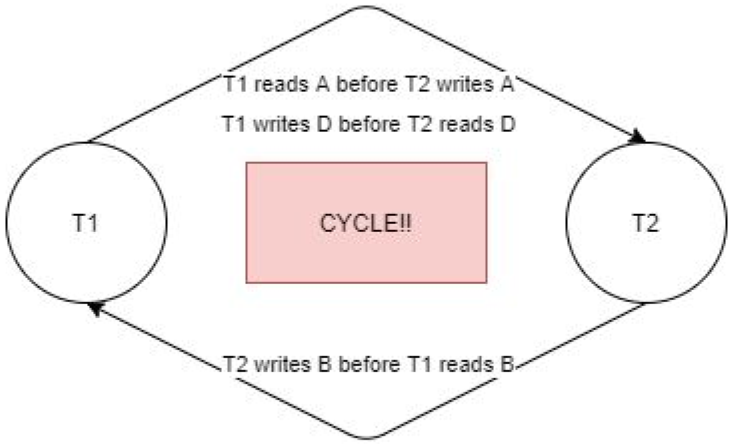
$r_1(A)$, $r_2(B)$, $r_1(D)$, $r_2(A)$, $w_2(A)$, $w_2(B)$, $r_1(B)$, $w_1(D)$, $commit_1$, $r_2(D)$, $commit_2$

where $r_i(X)$ (i=1,2 and X=A,B,D) denotes transaction $T_i$ reads database object X from disk, $w_i(X)$ denotes $T_i$ writes database object X to disk, and committee denotes $T_i$ commits.

| $T_1$ | $T_2$ |
|---|---|
| $r_1(A)$ | |
| | $r_2(B)$ |
| $r_1(D)$ | |
| | $r_2(A)$ |
| | $w_2(A)$ |
| | $w_2(B)$ |
| $r_1(B)$ | |
| $w_1(D)$ | |
| $commit_1$ | |
| | $r_2(D)$ |
| | $commit_2$ |

1. Create a precedence graph, then determine if the schedule is serializable. At each edge, indicate the transaction operations for adding the edge.

From the theorem, we know the schedule is serializable if and only if its precedence graph contains no cycles, so let's create this to test serializability where we'll use the read-before-write assumption. Thus, the edges are created between concurrently executing transaction vertices if there exists a data item R for which $T_1$ writes/reads R before $T_2$ reads/writes R (i.e. $T_1 \rightarrow T_2$).




Therefore, the schedule is NOT serializable because there is a cycle in the precedence graph (between T1 and T2).

2. Apply the protocol PSC to enforce the serializability of this schedule. Use $sl_i(X)$ to denote transaction $T_i$ requests for a shared lock on object X, $xl_i(X)$ to denote transaction $T_i$ requests for an exclusive lock or requests to promote a shared lock into an exclusive lock on object X, $u_i(X)$ to denote transaction $T_i$ unlocks object X. Visualize your schedule as above.

| | **T1** | **T2** |
|---|---|---|
| 1 | $sl_1(A)$ | |
| 2 | $r_1(A)$ | |
| 3 | | $sl_2(B)$ |
| 4 | | $r_2(B)$ |
| 5 | $sl_1(D)$ | |
| 6 | $r_1(D)$ | |
| 7 | | $sl_2(A)$ |
| 8 | | $r_2(A)$ |
| 9 | | $xl_2(A)$ (denied) |
| 10 | $sl_1(B)$ | (wait) |
| 11 | $r_1(B)$ | … |
| 12 | $xl_1(D)$ | … |

| | | |
|---|---|---|
| 13 | $w_1(D)$ | … |
| 14 | $u_1(A)$, $u_1(B)$, $u_1(D)$ | … |
| 15 | $commit_1$ | ... |
| 16 | | $xl_2(A)$ (granted) |
| 17 | | $w_2(A)$ |
| 18 | | $xl_2(B)$ |
| 19 | | $w_2(B)$ |
| 20 | | $sl_2(D)$ |
| 21 | | $r_2(D)$ |
| 22 | | $u_2(A)$, $u_2(B)$, $u_2(D)$ |
| 23 | | $commit_2$ |

**Question 2:** (5%)

The following is a schedule of the read/write operations of transactions $T_1$, $T_2$, and $T_3$.

$r_1(A)$, $r_2(D)$, $r_1(C)$, $r_3(B)$, $w_2(D)$, $w_1(A)$, $r_3(C)$, $r_1(B)$, $w_2(C)$, $r_3(D)$, $w_1(B)$, $w_3(C)$

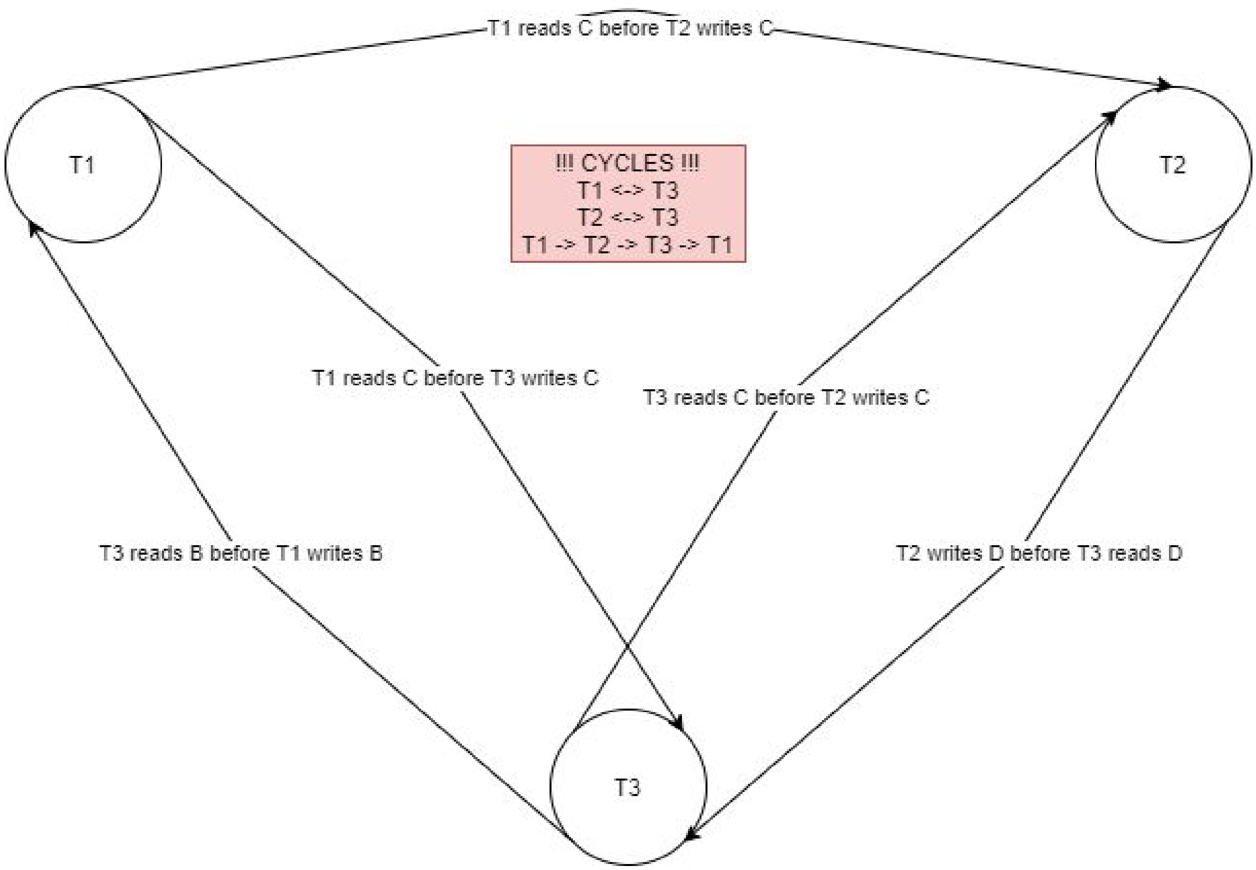
where $r_i(X)$ (i=1,2,3 and X=A,B,C,D) denotes transaction $T_i$ reads database object X from disk, $w_i(X)$ denotes $T_i$ writes database object X to disk.

The schedule can be visualized as:

| $T_1$ | $T_2$ | $T_3$ |
|---|---|---|
| $r_1(A)$ | | |
| | $r_2(D)$ | |
| $r_1(C)$ | | |
| | | $r_3(B)$ |
| | $w_2(D)$ | |
| $w_1(A)$ | | |
| | | $r_3(C)$ |
| $r_1(B)$ | | |
| | $w_2(C)$ | |
| | | $r_3(D)$ |
| $w_1(B)$ | | |
| | | $w_3(C)$ |

1. Create the precedence graph, then determine if the schedule is serializable. At each edge, indicate the transaction operations for adding the edge

Using the same rules as mentioned in Q1.1, let's test for serializability by creating a precedence graph and checking if any cycles are present. Please note, there is a violation of our read-before-write assumption since $T_2$ writes C without reading C before. For the purposes of this assignment, I will ignore this violation and answer the question as is, while mentioning slight differences that may occur if $r_2(C)$ is present before $w_2(C)$ in $T_2$. UPDATE: Just ignore the notes about this because we can change this to assuming that $T_2$ has read C before the schedule shown.




NOTE: Depending on where $r_2(C)$ occurs (although it must occur before $w_2(C)$), it may also contribute to the edge from $T_2$ to $T_3$ because of "$T_2$ reads C before $T_3$ writes C."
Therefore, the schedule is NOT serializable because the precedence graph contains cycles.

2. Apply protocol PSC to enforce the serializability of this schedule. Use $sl_i(X)$ to denote $T_i$ requests for a shared lock on object X, $xl_i(X)$ to denote transaction $T_i$ requests for an exclusive lock or request to promote a shared lock into an exclusive lock on object X. Visualize your schedule as above.

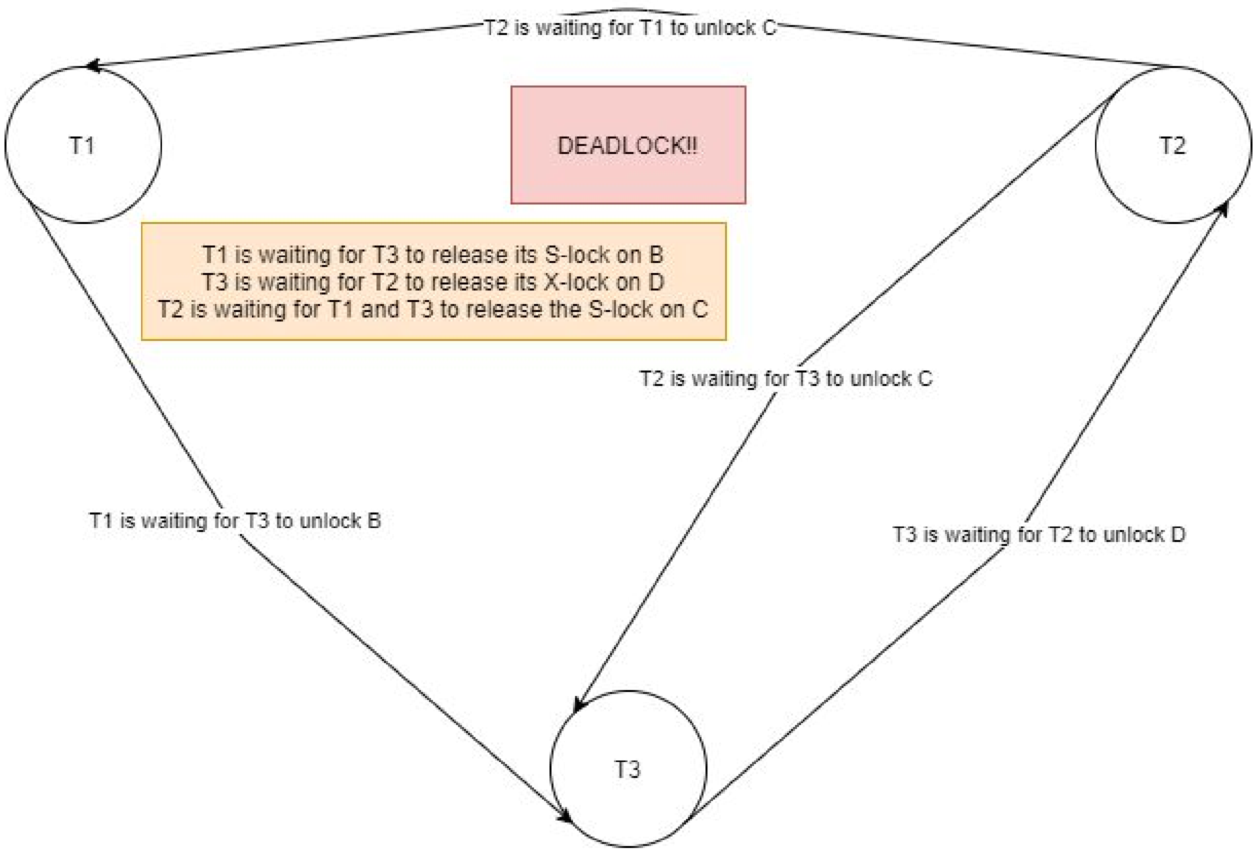
| | **T1** | **T2** | **T3** |
|---|---|---|---|

| | | | |
|---|---|---|---|
| 1 | $sl_1(A)$ | | |
| 2 | $r_1(A)$ | | |
| 3 | | $sl_2(D)$ | |
| 4 | | $r_2(D)$ | |
| 5 | $sl_1(C)$ | | |
| 6 | $r_1(C)$ | | |
| 7 | | | $sl_3(B)$ |
| 8 | | | $r_3(B)$ |
| 9 | | $xl_2(D)$ | |
| 10 | | $w_2(D)$ | |
| 11 | $xl_1(A)$ | | |
| 12 | $w_1(A)$ | | |
| 13 | | | $sl_3(C)$ |
| 14 | | | $r_3(C)$ |
| 15 | $sl_1(B)$ | | |
| 16 | $r_1(B)$ | | |
| 17 | | $xl_2(C)$ (denied) | |
| 18 | | (wait) | $sl_3(D)$ (denied) |
| 19 | $xl_1(B)$ (denied) | … | (wait) |
| 20 | (wait) | … | … |
| 21 | … | … | … |

Notice that a deadlock occurs since the transactions wait indefinitely for each other to release resources/locks. $T_1$ is waiting on $T_3$ to release its shared lock on B, but $T_3$ is waiting on $T_2$ to release its exclusive lock on D, but $T_2$ is waiting on both $T_1$ and $T_3$ to release their shared locks on C.

3. Create a wait-for graph from the result of question 2.2 for detecting deadlocks. If a deadlock is detected, how to solve it?

From the theorem, we know the schedule is deadlock-free if its wait-for graph contains no cycles, so let's create this using what we know above to detect any deadlocks, where the edges are drawn between concurrently executing transaction vertices if $T_i$ is waiting for $T_j$ to unlock an object (i.e. $T_i \rightarrow T_j$).




NOTE: Regardless of where $r_2(C)$ occurs (although it must occur before $w_2(C)$), the wait-for graph will not change since $T_2$ will be able to acquire a S-lock on C and will also be able promote it to an X-lock in order to write to C (but of course still need to wait for $T_1$ and $T_3$ to unlock C as well).
Therefore, a deadlock is detected because there are cycles in the wait-for graph (T1 → T3 → T2 → T1, & T2 ←→ T3).

Since a deadlock has been detected, we can break this by rolling back one of the transactions in the cycle, undoing its changes, and then restarting it again.

Let's rollback the youngest transaction, $T_3$, and undo its changes, since this transaction is involved with both cycles, and will remove both cycles in the wait-for graph. Doing this will allow $T_1$ to acquire an X-lock on B and finish its transaction while $T_2$ still waits on $T_1$ to unlock C. After, $T_3$ will be able to acquire an X-lock on C and finish its transaction. Finally, we can restart $T_3$, which can carry out its transaction completely.

Here is what the next steps could look like if we continue visualization of the schedule:

| | T1 | T2 | T3 |
|---|---|---|---|
| 21 | … | … | … |

| | | | |
|---|---|---|---|
| 22 | … | … | $u_3(B)$, $u_3(C)$ |
| 23 | … | … | $ROLLBACK_3$ |
| 24 | … | $xl_2(C)$ (still denied) | |
| 25 | $xl_1(B)$ (granted) | (still wait) | |
| 26 | $w_1(B)$ | … | |
| 27 | $u_1(A)$, $u_1(B)$, $u_1(C)$ | … | |
| 28 | $COMMIT_1$ | … | |
| 29 | | $xl_2(C)$ (granted) | |
| 30 | | $w_2(C)$ | |
| 31 | | $u_2(C)$, $u_2(D)$ | |
| 32 | | $COMMIT_2$ | (restart now) |
| 33 | | | $sl_3(B)$ |
| 34 | | | $r_3(B)$ |
| 35 | | | $sl_3(C)$ |
| 36 | | | $r_3(C)$ |
| 37 | | | $sl_3(D)$ |
| 38 | | | $r_3(D)$ |
| 39 | | | $xl_3(C)$ |
| 40 | | | $w_3(C)$ |
| 41 | | | $u_3(B)$, $u_3(C)$, $u_3(D)$ |
| 42 | | | $COMMIT_3$ |

**Question 3:** (5%)

A database crashed, but not damaged. The log shows that transactions A and B were in progress at the latest checkpointing. The log from the checkpointing up to the time of the failure is given below. It is assumed that the shared lock protocol (PSC) is used in this database for concurrency control.

| | | | | | |
|---|---|---|---|---|---|
| 1. | C | Begin trans | 11. | F | Begin trans |
| 2. | A | Write R1, old: 10, new: 20 | 12. | D | Write R1, old: 5, new: 10 |
| 3. | D | Begin trans | 13. | B | Write R2, old: 1030, new: 1050 |
| 4. | A | Commit | 14. | F | Write R3, old: 40, new: 50 |
| 5. | C | Write R1, old: 20, new: 5 | 15. | F | Rollback |
| 6. | C | Write R2, old: 1000, new: 1015 | 16. | D | Write R3, old: 40, new: 70 |
| 7. | E | Begin trans | | | |
| 8. | C | Commit | | | |
| 9. | E | Write R2, old: 1015, new: 1030 | | | |
| 10. | E | Commit | | | |

1. If an "immediate updating" strategy is used, which transactions should be undone, and which transactions should be redone, in recovering the database? Show how data objects R1, R2, R3 are recovered.

Step 1: Transactions that should be undone: all transactions that had no end-transaction record
- B, D → After step 1: R1 = 5, R2 = 1030, R3 = 40

Step 2: Transactions that should be redone: all committed transactions that had an end-transaction record (ignore transactions that were rolled back, i.e F.)
- A, C, E→ After step 2: R1 = 5, R2 = 1030, R3 = 40

Step 3: Resume transaction processing, possibly restarting transactions that had no end-transaction record
- B, D → After step 3: R1 = 10, R2 = 1050, R3 = 70

2. If a "deferred updating" strategy is used, which transactions should be undone, and which transactions should be redone, on recovering the database? Show how data objects R1, R2, R3 are recovered.

Step 1: Transactions that should be undone: no transactions need to be undone (SKIP step)

Step 2: Transactions that should be redone: all transactions that have committed
- A, C, E→ After step 2: R1 = 5, R2 = 1030

Step 3: Resume transaction processing, restarting transactions that have not committed
- B, D → After step 3: R1 = 10, R2 = 1050, R3 = 70

**Question 4:** (5%)

Relations t and c are instances of relation schemas Transaction(Tno, Vno, Account, T_Date, Amount) and Customer(Account, Cname, Province, Cbalance, Crlimit), where Tno is the primary key of Transaction, and Account is the primary key of Customer. Relation c has a primary index on Account, and relation t has a secondary index on Account. The two relations have the following statistics:

- The number of tuples in c is 30,000 and the number of tuples in t is 480,000
- The number of blocks containing c is 1,500 and the number of blocks containing t is 16,000

- In c, there are three Province values: ONT, BC, and QC
- The height of the index of c on Account is 5 and the height of the index of t on Account is 7. One index node is stored in a block in an index file on disk

The following is a user query:

```
select  Tno, Amount
from    t, c
where   t.Account = c.Account and
        Province = 'ONT';
```

1. Write an optimal strategy for this query in relational algebra.

<u>Relational Algebra</u>: $\Pi_{\text{Tno,Amount}}(\ \sigma_{\text{Province="ONT"}}(c) \bowtie t\ )$

2. Estimate the number of disk reads required for executing the strategy using indexed nested loop join. It is assumed that the RAM space available can contain at most 2,000 blocks. It is assumed that the selection result is not written to disk.

<u>Step 1</u>: Estimate the number of disk reads for the selection operation Province = 'ONT'

- Since attribute Province on c is a non-key attribute without a given index on Province and c is NOT sorted by Province because the primary index is on Account, we use a linear search:
    - Therefore, the number of disk reads is $b_c$ (the number of blocks containing tuples of c), which is **1500**

<u>Step 2</u>: Estimate the number of tuples in the result of the selection from Step 1

- sel(Province=ONT) = 1 / dist(Province) = ⅓
    - Since there are only 3 province values in c
- <#of output tuples> = selectivity * <#of input tuples> = ⅓ * 30,000 = **10,000**
    - Since $n_r$ = 30,000

<u>Step 3</u>: Estimate the cost of the indexed nested-loop join (for c' ⋈ t)

- Since the join condition is t.Account = c.Account, we know attribute t.Account is a foreign key referencing c.Account
    - According to the algorithm from class, join size = $n_t$ = 480,000
- $b_{c'} + n_{c'} * C = b_{c'} + n_{c'} * (h_{tt} + sc(Account,t))$

  $= b_{c'} + n_{c'} * (h_{tt} + sel(Account=const))$

  $= b_{c'} + n_{c'} * (h_{tt} + n_t / dist(Account))$

  $= b_{c'} + n_{c'} * (h_{tt} + n_t / n_c)$

  = 0 + 10,000 * (7 + 480,000 / 30,000)

  = 10,000 * (7 + 16) = 10,000 * 23 = **230,000**
    - Since the number of block containing tuples of c' is 0 (because the selection result from earlier is not written to disk, so no disk reads are needed to bring it to memory) and the number of tuples in c' is 10,000 (from Step 2)

- Since t has a secondary index on Account (a non-key attribute), the height of the index of t on Account is 7, and sc(Account,t) actually just works out to be the number of tuples in t divided by the number of distinct customer Accounts total (i.e. $n_c$)
- Notice that we never surpassed the assumption that the RAM space available can contain at most 2,000 blocks.

Step 4: Estimate the total cost by calculating the sum of the estimated cost of each component
Total cost = selection cost (Step 1) + join cost (Step 3, which used Step 2 as a helper)
= 1500 + 230,000 = **231,500**

*Therefore, the estimate of the number of disk reads required for executing the strategy using indexed nested loop join is 231,500.*

3. Estimate the number of tuples in the query result.

For c' ⋈ t that the join condition is c.Account = t.Account, so we know that attribute t.Account is a foreign key referencing c.Account. Using the algorithm from class, we know the number of tuples in the join result (i.e. join size) is $n_t$ (the number of tuples in t), which is 480,000. Thus, the number of tuples in the result of c' ⋈ t (i.e. output tuples) is:

<#of output tuples> = selectivity * <#of input tuples>
= sel(Province=ONT) * $n_t$
= ⅓ * 480,000 = **160,000**

- Since ⅓ is the selectivity of the selection condition (see Step 2)

*Therefore, the estimate of the number of tuples in the query result is 160,000.*