Solution for assignment 4

Exercise 16.2 Consider the following actions taken by transaction T1 on databases X and Y: R(X), W(X), R(Y), W(Y)

1. Give an example of another transaction T2 that, if run concurrently to transaction T without some form of concurrency control, could interfere with T1.
2. Explain how the use of Strict 2PL would prevent interference between the two transactions.
3. Strict 2PL is used in many database systems. Give two reasons for its popularity.

Answer:

1. If the transaction T2 performed W(Y) before T1 performed R(Y), and then T2 aborted, the value read by T1 would be invalid and the abort would be cascaded to T1 (i.e. T1 would also have to abort.).
2. Strict 2PL would require T2 to obtain an exclusive lock on Y before writing to it. This lock would have to be held until T2 committed or aborted; this would block T1 from reading Y until T2 was finished, but there would be no interference.
3. Strict 2PL is popular for many reasons. One reason is that it ensures only 'safe' interleaving of transactions so that transactions are recoverable, avoid cascading aborts, etc. Another reason is that strict 2PL is very simple and easy to implement. The lock manager only needs to provide a lookup for exclusive locks and an atomic locking mechanism (such as with a semaphore).

Exercise 16.4 We call a transaction that only reads database object a read-only transaction, other wise the transaction is called a read-write transaction. Give brief answers to the following questions:

1. What is lock thrashing and when does it occur?
2. What happens to the database system throughput if the number of read-write transactions is increased?
3. what happens to the database system throughput if the number of read-only transactions is increased?
4. Describe three ways of trning your system to increase transaction throughput

Answer:

1. Lock Thrashing is the point where system performance(throughput) decreases with increasing load (adding more active transactions). It happens due to the contention of locks. Transactions waste time on lock waits.
2. At the beginning, with the increase of transactions, throughput will increase, but the increase will stop at the thrashing point, after the point the throughput will drop with the increasing number of transactions.
3. For read-only transactions, there is no lock waiting. The throughput will increase with the increasing number of concurrent transactions.
4. Three ways of tuning your system to increase transaction throughput:
    a. By locking the smallest sized objects possible (reducing the likelihood that two transactions need the same lock).
    b. By reducing the time that transaction hold locks (so that other transactions are blocked for a shorter time).
    c. By reducing hot spots. A hot spot is a database object that is frequently accessed and modified, and causes a lot of blocking delays. Hot spots can significantly affect performance.

Exercise 17.2 Consider the following classes of schedules: *serializable*, *conflict-serializable*, *view-serializable*, *recoverable*, *avoids-cascading-aborts*, and *strict*. For each of the following schedules, state which of the preceding classes it belongs to. If you cannot decide whether a schedule belongs in a certain class based on the listed actions, explain briefly.

The actions are listed in the order they are scheduled and prefixed with the transaction name. If a commit or abort is not shown, the schedule is incomplete; assume that abort or commit must follow all the listed actions.

1. T1: R(X), T2: R(X), T1: W(X), T2: W(X)
2. T1: W(X), T2: R(Y), T1: R(Y), T2: R(X)
3. T1: R(X), T2: R(Y), T3: W(X), T2: R(X), T1: R(Y)
4. T1: R(X), T1: R(Y), T1: W(X), T2: R(Y), T3: W(Y), T1: W(X), T2: R(Y)
5. T1: R(X), T2: W(X), T1: W(X), T2: Abort, T1: Commit
6. T1: R(X), T2: W(X), T1: W(X), T2: Commit, T1: Commit
7. T1: W(X), T2: R(X), T1: W(X), T2: Abort, T1: Commit
8. T1: W(X), T2: R(X), T1: W(X), T2: Commit, T1: Commit
9. T1: W(X), T2: R(X), T1: W(X), T2: Commit, T1: Abort
10. T2: R(X), T3: W(X), T3: Commit, T1: W(Y), T1: Commit, T2: R(Y), T2: W(Z), T2: Commit
11. T1: R(X), T2: W(X), T2: Commit, T1: W(X), T1: Commit, T3: R(X), T3: Commit
12. T1: R(X), T2: W(X), T1: W(X), T3: R(X), T1: Commit, T2: Commit, T3: Commit

Answer:

1. Serizability (or view) cannot be decided but NOT conflict serizability. It is recoverable and avoid cascading aborts; NOT strict
2. It is serializable, conflict-serializable, and view-serializable regardless which action (commit or abort) follows It is NOT avoid cascading aborts, NOT strict; We can not decide whether it's recoverable or not, since the abort/commit sequence of these two transactions are not specified.
3. It is the same with 2.
4. Serizability (or view) cannot be decided but NOT conflict serizability. It is NOT avoid cascading aborts, NOT strict; We can not decide whether it's recoverable or not, since the abort/commit sequence of these transactions are not specified.
5. It is serializable, conflict-serializable, and view-serializable; It is recoverable and avoid cascading aborts; it is NOT strict.
6. It is NOT serializable, NOT view-serializable, NOT conflict-serializable; it is recoverable and avoid cascading aborts; It is NOT strict.
7. It belongs to all classes
8. It is serializable, NOT view-serializable, NOT conflict-serializable; It is NOT recoverable, therefore NOT avoid cascading aborts, NOT strict.
9. It is serializable, view-serializable, and conflict-serializable; It is NOT recoverable, therefore NOT avoid cascading aborts, NOT strict.
10. It belongs to all above classes.
11. It is NOT serializable and NOT view-serializable, NOT conflict-serializable; it is recoverable, avoid cascading aborts and strict.
12. It is NOT serializable and NOT view-serializable, NOT conflict-serializable; it is recoverable, but NOT avoid cascading aborts, NOT strict.

Exercise 17.4 Consider the following sequences of actions, listed in the order in which they are submitted to the DBMS:

- Sequence S1: T1:R(X), T2:W(X), T2:W(Y), T3:W(Y), T1:W(Y), T1: Commit, T2:Commit, T3:Commit
- Sequence S2: T1:R(X), T2:W(Y), T2:W(X), T3:W(Y), T1:W(Y), T1:Commit, T2:Commit, T3:Commit

For each sequence and for each of the following concurrency control mechanisms, describe how the concurrency control mechanism handles the sequence.

Assume that the timestamp of transaction T_i is i. For lock-based concurrency control mechanism, add lock and unlock requests to the above sequence of actions as per the locking protocol. If a transaction is blocked, assume that all of its actions are queued until it is resumed; the DBMS continues with the next action (according to the listed sequence) of an unblocked transaction.

1. Strict 2PL with timestamps used for deadlock prevention.
2. Strict 2PL with deadlock detection. Show the waits-for graph if a deadlock cycle develops.

Answer:

1. Assume we use Wait-Die policy.

   Sequence S1:

   T1 acquires shared-lock on X;

   for an exclusive-lock on X, since T2 has a lower priority, it will be aborted When T2 asks;

   T3 now gets exclusive-lock on Y;

   When T1 also asks for an exclusive-lock on Y, which is still held by T3, since T1 has higher priority, T1 will be blocked waiting;

   T3 now finishes write, commits and releases all the lock;

   T1 wakes up, acquires the lock, proceeds and finishes;

   T2 now can be restarted successfully.

   Sequence S2:

   The sequence and consequence are the same with sequence S1, except T2 was able to advance a little more before it gets aborted.

2. In deadlock detection, transactions are allowed to wait, they are not aborted until a deadlock has been detected. (Compared to prevention schema, some transactions may have been aborted prematurely.)

   Sequence S1:

   T1 gets a shared-lock on X;

   T2 blocks waiting for an exclusive-lock on X;

   T3 gets an exclusive-lock on Y;

   T1 blocks waiting for an exclusive-lock on Y;

   T3 finishes, commits and releases locks;

   T1 wakes up, get an exclusive-lock on Y, finishes up and releases lock on X and Y;

   T2 now gets both an exclusive-lock on X and Y, and proceeds to finish.

   No deadlock.

   Sequence S2:

   There is a deadlock. T1 waits for T2, while T2 waits for T1.

Exercise 18.4 Consider the execution shown in the figure below.

LSN	LOG
00	update: T1 writes P2
10	update: T1 writes P1
20	update: T2 writes P5
30	update: T3 writes P3
40	T3 commit
50	update: T2 writes P5
60	update: T2 writes P3
70	T2 abort

1. Extend the figure to show prevLSN and undonextLSN values.
2. Describe the actions taken to rollback transaction T2.
3. Show the log after T2 is rolled back, including all **prevLSN** and **undonextLSN** values in log records.

Answer:

Question 1:

LSN	LOG	prevLSN	undoNextLSN
00	Update: T1 writes P2	-	-
10	Update: T1 writes P1	00	00
20	Update: T2 writes P5	-	-
30	Update: T3 writes P3	-	-
40	T3 commit	30	Not an update log record
50	Update: T2 writes P5	20	20
60	Update: T2 writes P3	50	50
70	T2 abort	60	Not an update log record

Question 2:

Step 1: Restore P3 to the before-image stored in LSN 60.

Step 2: Restore P5 to the before-image stored in LSN 50.

Step 3: Restore P5 to the before-image stored in LSN 20.

Question 3:

The log tail should look something like this:

LSN	prevLSN	transID	Type	pageID	undoNextLSN
80	70	T2	CLR	P3	50
90	80	T2	CLR	P5	20
100	90	T2	CLR	P5	-
110	100	T2	END	-	-