Transactions - Concurrency

Exercise 16.3 Consider a database with objects *X* and *Y* and assume that there are two transactions *T*1 and *T*2. Transaction *T*1 reads objects *X* and *Y* and then writes object *X*. Transaction *T*2 reads objects *X* and *Y,* then reads X once more, and finally writes objects *X* and *Y* (i.e. T1: R(X), R(Y), W(X); T2: R(X), R(Y), R(X), W(X), W(Y))

1. Give an example schedule with actions of transactions *T*1 and *T*2 on objects *X* and *Y* that results in a write-read conflict.
2. Give an example schedule with actions of transactions *T*1 and *T*2 on objects *X* and *Y* that results in a read-write conflict.
3. Give an example schedule with actions of transactions *T*1 and *T*2 on objects *X* and *Y* that results in a write-write conflict.
4. For each of the three schedules, show that Strict 2PL disallows the schedule.

Exercise 16.6 Answer the following questions: SQL supports four isolation-levels and two access-modes, for a total of eight combinations of isolation-level and access-mode. Each combination implicitly defines a class of transactions; the following questions refer to these eight classes:

1. Consider the four SQL isolation levels. Describe which of the phenomena can occur at each of these isolation levels: *dirty read, unrepeatable read, phantom problem*.
2. For each of the four isolation levels, i) explain a lock-based implementation and ii) give an example of transaction that can run safely at that isolation level (i.e. doesn't expose any of the phenomena associated with that level of isolation), but doesn't run safely one level below (i.e. weaker level with more phenomena).
3. Why does the access mode of a transaction matter?

Exercise 17.4 Consider the following sequences of actions, listed in the order 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 *Ti* is *i*. For lock-based concurrency control mechanisms, add lock and unlock requests to the previous sequence of actions as per the locking protocol. The DBMS processes actions in the order shown. If a transaction is blocked, assume that all 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 in case of deadlock.)
3. Conservative (and Strict, i.e., with locks held until end-of-transaction) 2PL.

Solutions

Answer 16.3 The answer to each question is given below.

1. The following schedule results in a write-read conflict:
   T2:R(X), T2:R(Y), T2:W(X), T1:R(X) ...
   T1:R(X) is a dirty read here.
2. The following schedule results in a read-write conflict:
   T2:R(X), T1:R(X), T1:R(Y), T1:W(X), T2:R(X) ...
   Now, T2 will get an unrepeatable read on X.
3. The following schedule results in a write-write conflict:
   T2:R(X), T2:R(Y), T1:R(X), T1:R(Y), T1:W(X), T2:R(X), T2:W(X) ...
   Now, T2 has overwritten uncommitted data.
4. Strict 2PL resolves these conflicts as follows:
   (a) In S2PL, T1 could not get a shared lock on X because T2 would be holding an exclusive lock on X. Thus, T1 would have to wait until T2 was finished.
   (b) Here T1 could not get an exclusive lock on X because T2 would already be holding a shared or exclusive lock on X.
   (c) Same as above.

Answer 16.6 The answer to each question is given below.

1.

Level	Dirty Read	Unrepeatable Read	Phantom Problem
READ UNCOMMITTED	Yes	Yes	Yes
READ COMMITTED	No	Yes	Yes
REPEATABLE READ	No	No	Yes
SERIALIZABLE	No	No	No

2. (a) A SERIALIZABLE transaction achieves the highest degree of isolation from the effects of other transactions. It obtains locks before reading or writing objects, including locks on sets of objects that it requires to be unchanged and hold them until the end. Thus it is immune to all three phenomena above. We can safely run transactions like (assuming T2 inserts new objects):

T1: R(X), R(Y), W(X). W(Y) Commit
T2: W(X), W(Y) Commit

(b) A REPEATABLE READ transaction sets the same locks as a SERIALIZABLE transaction, except that it locks only objects, not sets of objects (i.e. phantom problem exists). We can safely run transactions like (assuming T1 or T2 does not insert any new objects):

T1: R(X), R(Y), W(X), W(Y), R(X) Commit
T2: R(X), W(X), R(Y), W(Y), Commit

(c) A READ COMMITTED transaction obtains exclusive locks before writing objects and holds these locks until the end. It also obtains shared lock before reading, but it releases it immediately. Thus it is only immune to dirty read. So we can safely run transactions like (assuming T1 or T2 does not insert any new objects):

T1: R(X), W(X), Commit
T2: R(X), W(X), Commit

(d) A READ UNCOMMITTED transaction can never make any lock requests, thus it is vulnerable to dirty read, unrepeatable read and phantom problem. Transactions which run safely at this level can only read objects from the database:

T1:R(X), R(Y), Commit
T2:R(Y), R(X), Commit

3. Access mode of a transaction tells what kind of lock is needed by the transaction. If the transaction is with READ ONLY access mode, only shared locks need to be obtained, thereby increasing concurrency.

Answer 17.4 The answer to each question is given below.

1. Assume we use Wait-Die policy.
   Sequence S1: T1 acquires shared-lock on X;
   When T2 asks for an exclusive lock on X, since T2 has a lower priority, it will be aborted;
   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 locks;
   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.
   T1 acquires shared-lock on X;
   T2 acquires exclusive-lock on Y;
   T2 asks for an exclusive-lock on X, but since it has lower priority than T1, it will be aborted;
   T3 acquires exclusive-lock on Y;
   T1 tries to acquire exclusive lock, but since it has higher priority than T3, it is allowed to wait
   T3 releases the lock and commits
   T1 acquire the lock on Y and proceeds

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, gets 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. Deadlocks are resolved either using a timeout or by maintaining a waits-for graph.
3. **Sequence S1:** With conservative and strict 2PL, the sequence is easy. T1 acquires lock on both X and Y, commits, releases locks; then T2; then T3.
   Sequence S2: Same as Sequence S1.