17

CONCURRENCY CONTROL

Exercise 17.1 Answer the following questions:

1. Describe how a typical lock manager is implemented. Why must lock and unlock be atomic operations? What is the difference between a lock and a *latch*? What are *convoys* and how should a lock manager handle them?
2. Compare *lock downgrades* with upgrades. Explain why downgrades violate 2PL but are nonetheless acceptable. Discuss the use of *update* locks in conjunction with lock downgrades.
3. Contrast the timestamps assigned to restarted transactions when timestamps are used for deadlock prevention versus when timestamps are used for concurrency control.
4. State and justify the Thomas Write Rule.
5. Show that, if two schedules are conflict equivalent, then they are view equivalent.
6. Give an example of a serializable schedule that is not strict.
7. Give an example of a strict schedule that is not serializable.
8. Motivate and describe the use of locks for improved conflict resolution in Optimistic Concurrency Control.

Answer 17.1 The answer to each question is given below.

1. A typical lock manager is implemented with a hash table, also called lock table, with the data object identifier as the key. A lock table entry contains the following information: the number of transactions currently holding a lock on the object, the nature of the lock, and a pointer to a queue of lock requests.

(c) Otherwise, T writes O and WTS(O) is set to TS(T).

The justification is as follows: had $TS(T) < RTS(O)$, T would have been aborted and we would not have bothered to check the WTS(O). So to decide whether to abort T based on WTS(O), we can assume that $TS(T) >= RTS(O)$. If $TS(T) >= RTS(O)$ and $TS(T) < WTS(O)$, then $RTS(O) < WTS(O)$, which means the previous write occurred immediately before this planned-new-write of O and was never read by anyone, therefore the previous write can be safely ignored.

5. If two schedules over the same set of actions of the transactions are conflict equivalent, they must order every pair of conflicting actions of two committed transactions in the same way. Let's assume that two schedules are conflict equivalent, but they are not view equivalent, then one of the three conditions held under view equivalency must be violated. But as we can see if every pair of conflicting actions is ordered in the same way, this cannot happen. Thus we can conclude that if two schedules are conflict equivalent, they are also view equivalent.

6. The following example is a serializable schedule, but it's not strict.
T1:R(X), T2:R(X), T2:W(X), T1:W(X), T2:Commit, T1:Commit

7. The following example is a strict schedule, but it's not serializable.
T1:R(X), T2:R(X), T1:W(X), T1:Commit, T2:W(X), T2:Commit

8. In Optimistic Concurrency Control, we have no way to tell when Ti wrote the object at the time we validate Tj, since all we have is the list of objects written by Ti and the list read by Tj. To solve such conflict, we use mechanisms very similar to locking. The basic idea is that each transaction in the Read phase tells the DBMS about items it is reading, and when a transaction Ti is committed and its writes are accepted, the DBMS checks whether any of the items written by Ti are being rad by any (yet to be validated) transaction Tj. If so, we know that Tj's validation must eventually fail. Then we can pick either the die or kill policy to resolve the conflict.

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 17.2 For simplicity, we assume the listed transactions are the only ones active currently in the database and if a commit or abort is not shown for a transaction, we'll assume a commit will follow all the listed actions.

1. Not serializable, not conflict-serializable, not view-serializable;
   It is recoverable and avoid cascading aborts; not strict.
2. It is serializable, conflict-serializable, and view-serializable;
   It does NOT avoid cascading aborts, is 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 number 2 above.
4. It is NOT serializable, NOT conflict-serializable, NOT view-serializable;
   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 serializable and view-serializable, not conflict-serializable;
   It is recoverable and avoid cascading aborts;
   It is not strict.

7. It is not serializable, not view-serializable, not conflict-serializable;
   It is not recoverable, therefore not avoid cascading aborts, not strict.
8. It is not 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. (assume the 2nd T2:Commit is instead T1:Commit).
    It is serializable and view-serializable, not conflict-serializable;
    It is recoverable, avoid cascading aborts and strict.
12. It is serializable and view-serializable, not conflict-serializable;
    It is recoverable, but not avoid cascading aborts, not strict.

Exercise 17.3 Consider the following concurrency control protocols: 2PL, Strict 2PL, Conservative 2PL, Optimistic, Timestamp without the Thomas Write Rule, Timestamp with the Thomas Write Rule, and Multiversion. For each of the schedules in Exercise 17.2, state which of these protocols allows it, that is, allows the actions to occur in exactly the order shown.

For the timestamp-based protocols, assume that the timestamp for transaction Ti is i and that a version of the protocol that ensures recoverability is used. Further, if the Thomas Write Rule is used, show the equivalent serial schedule.

Answer 17.3 See the table 17.1.

Note the following abbreviations.
S-2PL: Strict 2PL; C-2PL: Conservative 2PL; Opt cc: Optimistic; TS W/O THR: Timestamp without Thomas Write Rule; TS With THR: Timestamp without Thomas Write Rule.

Thomas Write Rule is used in the following schedules, and the equivalent serial schedules are shown below:
5. T1:R(X), T1:W(X), T2:Abort, T1:Commit

6. T1:R(X), T1:W(X), T2:Commit, T1:Commit
11. T1:R(X), T2:Commit, T1:W(X), T2:Commit, T3:R(X), T3:Commit

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

	2PL	S-2PL	C-2PL	Opt CC	TS w/o TWR	TS w/ TWR	Multiv.
1	N	N	N	N	N	N	N
2	Y	N	N	Y	Y	Y	Y
3	N	N	N	Y	N	N	Y
4	N	N	N	Y	N	N	Y
5	N	N	N	Y	N	Y	Y
6	N	N	N	N	N	Y	Y
7	N	N	N	Y	N	N	N
8	N	N	N	N	N	N	N
9	N	N	N	Y	N	N	N
10	N	N	N	N	Y	Y	Y
11	N	N	N	N	N	Y	N
12	N	N	N	N	N	Y	Y

Table 17.1

- **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.
4. Optimistic concurrency control.
5. Timestamp concurrency control with buffering of reads and writes (to ensure recoverability) and the Thomas Write Rule.
6. Multiversion concurrency control.

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 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, 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.

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.

4. Optimistic concurrency control:
   For both S1 and S2: each transaction will execute, read values from the database and write to a private workspace; they then acquire a timestamp to enter the validation phase. The timestamp of transaction Ti is i.

   Sequence S1: Since T1 gets the earliest timestamp, it will commit without problem; but when validating T2 against T1, none of the three conditions hold, so T2 will be aborted and restarted later; so is T3 (same as T2).

   Sequence S2: The fate is the same as in Sequence S1.

	Serializable	Conflict-serializable	Recoverable	Avoid cascading aborts
1	No	No	No	No
2	No	No	Yes	Yes
3	Yes	Yes	Yes	Yes
4	Yes	Yes	Yes	Yes

Table 17.2

5. Timestamp concurrency control with buffering of reads and writes and TWR.

   Sequence S1: This sequence will be allowed the way it is.

   Sequence S2: Same as above.

6. Multiversion concurrency control

   Sequence S1: T1 reads X, so RTS(X) = 1;
   T2 is able to write X, since TS(T2) ¿ RTS(X); and RTS(X) and WTS(X) are set to 2;
   T2 writes Y, RTS(Y) and WTS(Y) are set to 2;
   T3 is able to write Y as well, so RTS(Y) and WTS(Y) are set to 3;
   Now when T1 tries to write Y, since TS(T1) ¡ RTS(Y), T1 needs to be aborted and restarted later.

   Sequence S2: The fate is similar to the one in Sequence S1.

Exercise 17.5 For each of the following locking protocols, assuming that every transaction follows that locking protocol, state which of these desirable properties are ensured: serializability, conflict-serializability, recoverability, avoidance of cascading aborts.

1. Always obtain an exclusive lock before writing; hold exclusive locks until end-of-transaction. No shared locks are ever obtained.
2. In addition to (1), obtain a shared lock before reading; shared locks can be released at any time.
3. As in (2), and in addition, locking is two-phase.
4. As in (2), and in addition, all locks held until end-of-transaction.

Answer 17.5 See the table 17.2.

Exercise 17.6 The Venn diagram (from [76]) in Figure 17.1 shows the inclusions between several classes of schedules. Give one example schedule for each of the regions $S1$ through $S12$ in the diagram.

18

CRASH RECOVERY

Exercise 18.1 Briefly answer the following questions:

1. How does the recovery manager ensure atomicity of transactions? How does it ensure durability?
2. What is the difference between stable storage and disk?
3. What is the difference between a system crash and a media failure?
4. Explain the WAL protocol.
5. Describe the steal and no-force policies.

Answer 18.1 The answer to each question is given below.

1. The Recovery Manager ensures atomicity of transactions by undoing the actions of transactions that do not commit. It ensures durability by making sure that all actions of committed transactions survive system crashes and media failures.
2. Stable storage is guaranteed (with very high probability) to survive crashes and media failures. A disk might get corrupted or fail but the stable storage is still expected to retain whatever is stored in it. One of the ways of achieving stable storage is to store the information in a set of disks rather than in a single disk with some information duplicated so that the information is available even if one or two of the disks fail.
3. A system crash happens when the system stops functioning in a normal way or stops altogether. The Recovery Manager and other parts of the DBMS stop functioning (e.g. a core dump caused by a bus error) as opposed to media failure.

   In a media failure, the system is up and running but a particular entity of the system is not functioning. In this case, the Recovery Manager is still functioning and can start recovering from the failure while the system is still running (e.g., a disk is corrupted).