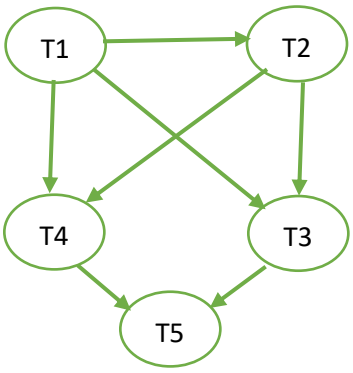
14.6 Consider the precedence graph of Figure 14.16. Is the corresponding schedule conflict serializable? Explain your answer.

Ans: There is a serializable schedule corresponding to the precedence graph below, since the graph is acyclic. A possible schedule is obtained by doing a topological sort, that is, T1, T2, T3, T4, T5.



14.8 The lost update anomaly is said to occur if a transaction Tj reads a data item, then another transaction Tk writes the data item (possibly based on a previous read), after which Tj writes the data item. The update performed by Tk has been lost, since the update done by Tj ignored the value written by Tk.

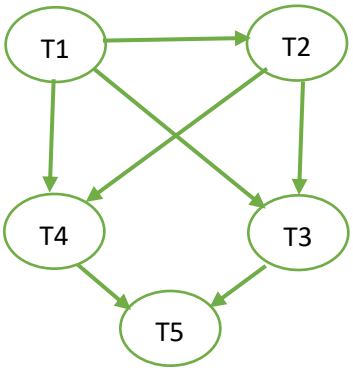


Figure 14.16 Precedence graph for Practice Exercise 14.6.

a. Give an example of a schedule showing the lost update anomaly.

Ans: A schedule showing the Lost Update Anomaly:

T1	T2
Read(A)	
	Read(A)
	Write(A)
Write(A)	

In the above schedule, the value written by the transaction T2 is lost because of the write of the transaction T1.

b. Give an example schedule to show that the lost update anomaly is possible with the read committed isolation level.

Ans: Lost Update Anomaly in Read Committed Isolation Level

T1	T2
lock-S(A)	
read(A)	
unlock(A)	
	lock-X(A)
	read(A)
	write(A)
	unlock(A)
	commit
lock-X(A)	
write(A)	
unlock(A)	
commit	

c. Explain why the lost update anomaly is not possible with the repeatable read isolation level.

Ans: Lost Update Anomaly is not possible in Repeatable Read isolation level. In repeatable read isolation level, a transaction T1 reading a data item X, holds a shared lock on X till the end. This makes it impossible for a newer transaction T2 to write the value of X (which requires X-lock) until T1 finishes. This forces the serialization order T1, T2 and thus the value written by T2 is not lost.

14.12 List the ACID properties. Explain the usefulness of each

Ans: The ACID properties, and the need for each of them are:

Consistency: Execution of a transaction in isolation (that is, with no other transaction executing concurrently) preserves the consistency of the database. This is typically the responsibility of the application programmer who codes the transactions.

Atomicity: Either all operations of the transaction are reflected properly in the database, or none are. Clearly lack of atomicity will lead to inconsistency in the database.

Isolation: When multiple transactions execute concurrently, it should be the case that, for every pair of transactions Ti and Tj , it appears to Ti that either Tj finished execution before Ti started, or Tj started execution after Ti finished. Thus, each transaction is unaware of other transactions executing concurrently with it. The user view of a transaction system requires the isolation property, and the property that concurrent schedules take the system from one consistent state to another. These requirements are satisfied by ensuring that only serializable schedules of individually consistency preserving transactions are allowed.

Durability: After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.

14.13 During its execution, a transaction passes through several states, until it finally commits or aborts. List all possible sequences of states through which a transaction may pass. Explain why each state transition may occur.

Ans: The possible sequences of states are

a. active → partially committed → committed. This is the normal sequence a successful transaction will follow. After executing all its statements, it enters the partially committed state. After enough recovery information has been written to disk, the transaction finally enters the committed state.

b. active → partially committed → aborted. After executing the last statement of the transaction, it enters the partially committed state. But before enough recovery information is written to disk, a hardware failure may occur destroying the memory contents. In this case the changes which it made to the database are undone, and the transaction enters the aborted state.

c. active → failed → aborted. After the transaction starts, if it is discovered at some point that normal execution cannot continue (either due to internal program errors or external errors), it enters the failed state. It is then rolled back, after which it enters the aborted state.

14.15 Consider the following two transactions:

```
T13: read(A);
     read(B);
     if A = 0 then B= B + 1;
     write(B).
T14: 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.

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

Ans: There are two possible executions: T1 T2 and T2 T1.

Case 1:

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

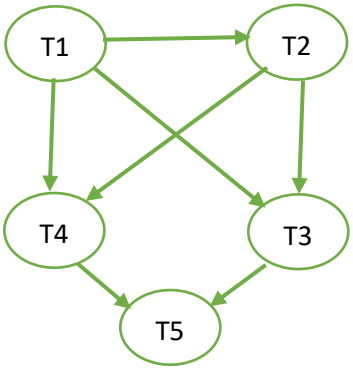
Consistency met: $A = 0 \vee B = 0 \equiv T \vee F = T$

Case 2:

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

Consistency met: $A = 0 \vee B = 0 \equiv F \vee T = T$

b. Show a concurrent execution of *T*13 and *T*14 that produces a non-serializable schedule.
Ans: 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 & B= B+1	
	if B=0 & A= A+1
	write(A)
write(B)	

c. Is there a concurrent execution of *T*13 and *T*14 that produces a serializable schedule?
Ans: There is no parallel execution resulting in a serializable schedule. From part a. we know that a serializable schedule results in A = 0 ∨ 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 ∧ A = 1 → ¬ (A = 0 ∨ B = 0). Similarly, for starting with T2 read(B).

14.17 What is a recoverable schedule? Why is recoverability of schedules desirable? Are there any circumstances under which it would be desirable to allow nonrecoverable schedules? Explain your answer.
Ans: recoverable schedule is one where, for each pair of transactions Ti and Tj such that Tj reads data items previously written by Ti, the commit operation of Ti appears before the commit operation of Tj. Recoverable schedules are desirable because failure of a transaction might otherwise bring the system into an irreversibly inconsistent state. Nonrecoverable schedules may sometimes be needed when updates must be made visible early due to time constraints, even if they have not yet been committed, which may be required for very long duration transactions.

14.18 Why do database systems support concurrent execution of transactions, in spite of the extra programming effort needed to ensure that concurrent execution does not cause any problems?
Ans: Transaction-processing systems usually allow multiple transactions to run concurrently. It is far easier to insist that transactions run serially. However, there are two good reasons for allowing concurrency:
Improved throughput and resource utilization. A transaction my involve I/O activity, CPU activity. The CPU and the disk in a computer system can operate in parallel. This can be exploited to run multiple transactions in parallel. For example, while a read or write on behalf of one transaction is in progress on one disk, another transaction can be running in the CPU. This increases the throughput of the system.
Reduced waiting time. If transactions run serially, a short transaction may have to wait for a preceding long transaction to complete. If the transactions are operating on different parts of the database, it is better to let them run concurrently, sharing the CPU cycles and disk accesses among them. It reduces the unpredictable delays and the average response time

14.20 For each of the following isolation levels, give an example of a schedule that respects the specified level of isolation, but is not serializable:
a. Read uncommitted
Ans:

T1	T2
Read(A)	
Write(A)	
	Read(A)
	Write(A)
Read(A)	

In the above schedule, T2 reads the value of A written by T1 even before T1 commits. This schedule is not serializable since T1 also reads a value written by T2, resulting in a cycle in the precedence graph.

b. Read committed
Ans:

T1	T2
lock-S(A)	
read(A)	
unlock(A)	
	lock-X(A)
	write(A)
	unlock(A)
	commit
lock-X(A)	
read(A)	
unlock-S(A)	
commit	

In the above schedule, the first time T1 reads A, it sees a value of A before it was written by T2, while the second read(A) by T1 sees the value written by T2 (which has already committed). The first read results in T1 preceding T2, while the second read results in T2 preceding T1, and thus the schedule is not serializable.

c. Repeatable read

Ans: Consider the following schedule, where T1 reads all tuples in r satisfying predicate P; to satisfy repeatable read, it must also share-lock these tuples in a two-phase manner.

T1	T2
pred read(r, P)	
	insert(t)
	Write(A)
	commit
Read(A)	
commit	

Suppose that the tuple t inserted by T2 satisfies P; then the insert by T2 causes T2 to serialized after T1, since T1 does not see t. However, the final read(A) operation of T1 forces T2 to precede T1, causing a cycle in the precedence graph