file systems would not be willing to pay the price (monetary, disk space, time) of supporting ACID properties.

15.5 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.
Answer: 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.

15.6 Justify the following statement: Concurrent execution of transactions is more important when data must be fetched from (slow) disk or when transactions are long, and is less important when data is in memory and transactions are very short.
Answer: If a transaction is very long or when it fetches data from a slow disk, it takes a long time to complete. In absence of concurrency, other transactions will have to wait for longer period of time. Average responce time will increase. Also when the transaction is reading data from disk, CPU is idle. So resources are not properly utilized. Hence concurrent execution becomes important in this case. However, when the transactions are short or the data is available in memory, these problems do not occur.

15.7 Explain the distinction between the terms *serial schedule* and *serializable schedule*.
Answer: A schedule in which all the instructions belonging to one single transaction appear together is called a *serial schedule*. A *serializable schedule* has a weaker restriction that it should be *equivalent* to some serial schedule. There are two definitions of schedule equivalence – conflict equivalence and view equivalence. Both of these are described in the chapter.

15.8 Consider the following two transactions:

```
$T_1$: read($A$);
     read($B$);
     if $A = 0$ then $B := B + 1$;
     write($B$).
$T_2$: read($B$);
     read($A$);
     if $B = 0$ then $A := A + 1$;
     write($A$).
```

Let the consistency requirement be $A = 0 \lor B = 0$, with $A = B = 0$ the initial values.

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

b. Show a concurrent execution of T_1 and T_2 that produces a nonserializable schedule.

c. Is there a concurrent execution of T_1 and T_2 that produces a serializable schedule?

Answer:

a. There are two possible executions: T_1 T_2 and T_2 T_1.

Case 1:

	A	B
initially	0	0
after T_1	0	1
after T_2	0	1

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

Case 2:

	A	B
initially	0	0
after T_2	1	0
after T_1	1	0

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

b. Any interleaving of T_1 and T_2 results in a non-serializable schedule.

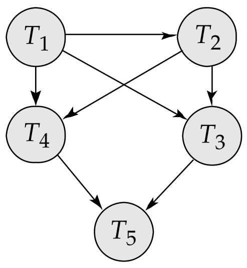


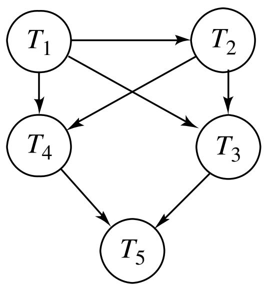
Figure 15.18. Precedence graph.

T_1	T_2
read(A)	
	read(B)
	read(A)
read(B)	
if $A = 0$ **then** $B = B + 1$	
	if $B = 0$ **then** $A = A + 1$
	write(A)
write(B)	

c. There is no parallel execution resulting in a serializable schedule. From part a. we know that a serializable schedule results in $A = 0 \vee B = 0$. Suppose we start with T_1 **read**(A). Then when the schedule ends, no matter when we run the steps of T_2, $B = 1$. Now suppose we start executing T_2 prior to completion of T_1. Then T_2 **read**(B) will give B a value of 0. So when T_2 completes, $A = 1$. Thus $B = 1 \wedge A = 1 \rightarrow \neg\, (A = 0 \vee B = 0)$. Similarly for starting with T_2 **read**(B).

15.9 Since every conflict-serializable schedule is view serializable, why do we emphasize conflict serializability rather than view serializability?
Answer: Most of the concurrency control protocols (protocols for ensuring that only serializable schedules are generated) used in practise are based on conflict serializability—they actually permit only a subset of conflict serializable schedules. The general form of view serializability is very expensive to test, and only a very restricted form of it is used for concurrency control.

15.10 Consider the precedence graph of Figure 15.18. Is the corresponding schedule conflict serializable? Explain your answer.
Answer: 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, T_1, T_2, T_3, T_4, T_5.



15.11 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.
Answer: A recoverable schedule is one where, for each pair of transactions T_i and T_j such that T_j reads data items previously written by T_i, the commit

operation of T_i appears before the commit operation of T_j. 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.

15.12 What is a cascadeless schedule? Why is cascadelessness of schedules desirable? Are there any circumstances under which it would be desirable to allow noncascadeless schedules? Explain your answer.

Answer: A cascadeless schedule is one where, for each pair of transactions T_i and T_j such that T_j reads data items previously written by T_i, the commit operation of T_i appears before the read operation of T_j. Cascadeless schedules are desirable because the failure of a transaction does not lead to the aborting of any other transaction. Of course this comes at the cost of less concurrency. If failures occur rarely, so that we can pay the price of cascading aborts for the increased concurrency, noncascadeless schedules might be desirable.