CHAPTER 17

Transactions

Practice Exercises

17.1 Suppose that there is a database system that never fails. Is a recovery manager required for this system?

Answer:
Even in this case the recovery manager is needed to perform rollback of aborted transactions for cases where the transaction itself fails.

17.2 Consider a file system such as the one on your favorite operating system.

a. What are the steps involved in the creation and deletion of files and in writing data to a file?

b. Explain how the issues of atomicity and durability are relevant to the creation and deletion of files and to writing data to files.

Answer:
There are several steps in the creation of a file. A storage area is assigned to the file in the file system. (In UNIX, a unique i-number is given to the file and an i-node entry is inserted into the i-list.) Deletion of file involves exactly opposite steps.

For the file system user, durability is important for obvious reasons, but atomicity is not relevant generally as the file system doesn't support transactions. To the file system implementor, though, many of the internal file system actions need to have transaction semantics. All steps involved in creation/deletion of the file must be atomic, otherwise there will be unreferenceable files or unusable areas in the file system.

17.3 Database-system implementers have paid much more attention to the ACID properties than have file-system implementers. Why might this be the case?

Answer:

Database systems usually perform crucial tasks whose effects need to be atomic and durable, and whose outcome affects the real world in a permanent manner. Examples of such tasks are monetary transactions, seat bookings etc. Hence the ACID properties have to be ensured. In contrast, most users of file systems would not be willing to pay the price (monetary, disk space, time) of supporting ACID properties.

17.4 What class or classes of storage can be used to ensure durability? Why?

Answer:
Only stable storage ensures true durability. Even nonvolatile storage is susceptible to data loss, albeit less so than volatile storage. Stable storage is only an abstraction. It is approximated by redundant use of nonvolatile storage in which data are not only replicated but distributed phyically to reduce to near zero the chance of a single event casuing data loss.

17.5 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 practice 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.

17.6 Consider the precedence graph of Figure 17.16. Is the corresponding schedule conflict serializable? Explain your answer.

Answer:

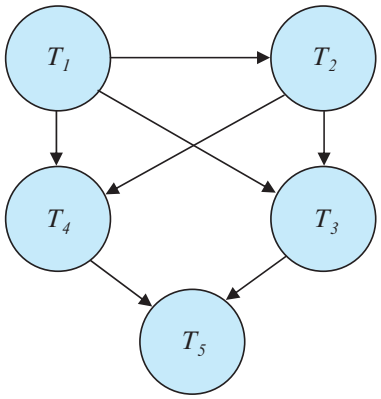

Figure 17.16 Precedence graph for Practice Exercise 17.6.

There is a serializable schedule corresponding to the precedence graph 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.

17.7 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.

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

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

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

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

Answer:

a. A schedule showing the lost update anomaly:

T_1	T_2
read(A)	
	read(A)
	write(A)
write(A)	

In the above schedule, the value written by the transaction T_2 is lost because of the write of the transaction T_1.

b. Lost update anomaly in read-committed isolation level:

T_1	T_2
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	

The locking in the above schedule ensures the read-committed isolation level. The value written by transaction T_2 is lost due to T_1's write.

c. Lost update anomaly is not possible in repeatable read isolation level. In repeatable read isolation level, a transaction T_1 reading a data item X holds a shared lock on X till the end. This makes it impossible for a newer transaction T_2 to write the value of X (which requires X-lock) until T_1 finishes. This forces the serialization order T_1, T_2, and thus the value written by T_2 is not lost.

17.9 Consider a database for a bank where the database system uses snapshot isolation. Describe a particular scenario in which a nonserializable execution occurs that would present a problem for the bank.

Answer:

Suppose that the bank enforces the integrity constraint that the sum of the balances in the checking and the savings account of a customer must not be negative. Suppose the checking and savings balances for a customer are \$100 and \$200 respectively.

Suppose that transaction T_1 withdraws \$200 from the checking account after verifying the integrity constraint by reading both the balances. Suppose that concurrent transaction T_2 withdraws \$200 from the checking account after verifying the integrity constraint by reading both the balances.

Since each of the transactions checks the integrity constraints on its own snapshot, if they run concurrently, each will believe that the sum of the balances after the withdrawal is \$100, and therefore its withdrawal does not violate the integrity constraint. Since the two transactions update different data items, they do not have any update conflict, and under snapshot isolation both

of them can commit. This is a nonserializable execution which results into a serious problem.

17.10 Consider a database for an airline where the database system uses snapshot isolation. Describe a particular scenario in which a nonserializable execution occurs, but the airline may be willing to accept it in order to gain better overall performance.

Answer:

Consider a web-based airline reservation system. There could be many concurrent requests to see the list of available flights and available seats in each flight and to book tickets. Suppose there are two users A and B concurrently accessing this web application, and only one seat is left on a flight.

Suppose that both user A and user B execute transactions to book a seat on the flight and suppose that each transaction checks the total number of seats booked on the flight, and inserts a new booking record if there are enough seats left. Let T_3 and T_4 be their respective booking transactions, which run concurrently. Now T_3 and T_4 will see from their snapshots that one ticket is available and will insert new booking records. Since the two transactions do not update any common data item (tuple), snapshot isolation allows both transactions to commit. This results in an extra booking, beyond the number of seats available on the flight.

However, this situation is usually not very serious since cancellations often resolve the conflict; even if the conflict is present at the time the flight is to leave, the airline can arrange a different flight for one of the passengers on the flight, giving incentives to accept the change. Using snapshot isolation improves the overall performance in this case since the booking transactions read the data from their snapshots only and do not block other concurrent transactions.

17.11 The definition of a schedule assumes that operations can be totally ordered by time. Consider a database system that runs on a system with multiple processors, where it is not always possible to establish an exact ordering between operations that executed on different processors. However, operations on a data item can be totally ordered.

Does this situation cause any problem for the definition of conflict serializability? Explain your answer.

Answer:

The given situation will not cause any problem for the definition of conflict serializability since the ordering of operations on each data item is necessary for conflict serializability, whereas the ordering of operations on different data items is not important.

T_1	T_2
read(A)	
	read(B)
write(B)	

For the above schedule to be conflict serializable, the only ordering requirement is **read**(B) -> **write**(B). **read**(A) and **read**(B) can be in any order.

Therefore, as long as the operations on a data item can be totally ordered, the definition of conflict serializability should hold on the given multiprocessor system.