14.1 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 roll-back of aborted transactions.

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

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

14.3 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 transac- tions, 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.

14.4 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 are inmemory 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 timewill increase.Alsowhen the transaction is readingdata 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.

14.5 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 en- suring 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.

14.6 Consider the precedence graph of Figure 14.16. 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, $T1$, $T2$, $T3$, $T4$, $T5$.

14.7 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 transac- tions *Ti* and *Tj* such that *Tj* reads data items previously written by *Ti* , the commit operation of *Ti* appears before the read operation of *Tj* . Cascade- less 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.

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), afterwhich *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* .

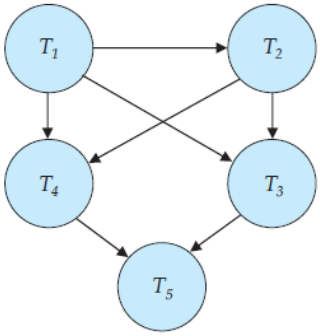

Figure 14.16 Precedence graph for Practice Exercise 14.6.

a. Give an example of a schedule showing the lost update anomaly.
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 $T2$ is lost because of the write of the transaction $T1$.

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 $T2$ is lost due to $T1$'s write.

c. 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.

14.9 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 sumof 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 $T1$ withdraws $200 from the checking account after verifying the integrity constraint by reading both the balances. Sup-pose that concurrent transaction $T2$ withdraws $200 from the checking account after verifying the integrity constraint by reading both the bal- ances. 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 non-serializable execution which results into a serious problem of withdrawal of more money.

14.10 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 $T3$ and $T4$ be their respective booking transactions, which run concurrently. Now $T3$ and $T4$ will see from their snapshots that one ticket is available and 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.

14.11 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 the above situation cause any problemfor the definition of conflict serializability? Explain your answer.

Answer:

The given situation will not cause any problem for the definition of con- flict 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 re- quirement 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 multi- processor system.

14.12 List the ACID properties. Explain the usefulness of each.

Answer:

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.

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 programerrors or external errors), it enters the failed state. It is then rolled back, after which it enters the *aborted* state.

14.14 Explain the distinction between the terms *serial schedule* and *serializable schedule*.

Answer:

A schedule inwhich 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.

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.

b. Show a concurrent execution of $T13$ and $T14$ that produces a nonserializable schedule.

c. Is there a concurrent execution of $T13$ and $T14$ that produces a serializable schedule?

Answer:

a. There are two possible executions: $T1$ $T2$ and $T2$ $T1$.

Case 1:

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

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

Case 2:

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

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

b. Any interleaving of $T1$ and $T2$ results in a non-serializable schedule.

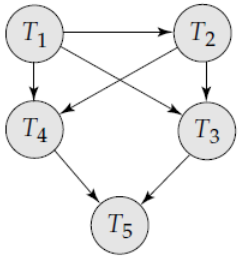

Figure 15.18. Precedence graph.

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 $T1$ **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 $B = 1 \wedge A = 1 \rightarrow \neg (A = 0 \vee B = 0)$. Similarly for starting with $T2$ **read**(B).

14.16 Give an example of a serializable schedule with two transactions such that the order in which the transactions commit is different from the serialization order.

Answer:

Let r, s and t be three relations. Consider a materialized view on these defined by (r 1 s 1 t). Suppose relation r does not have any attributes common to s or t, while s and t have foreign key relationship. Each of them have 1000 tuples and 100 tuples are added to r. Then

recomputation is better because (s 1 t) can be computed first which will have 1000 tuples. It can then be joined with t. In incremental view maintenance, the increment in t will first be joined with either s or t which will have 100000 tuples (cartesian product). This huge relation will then be joined with t which will be very expensive. However, if 100 tuples are added to s instead of r in the above situation, incremental view maintenance will obviously be better as increment in s can be joined with t to get a relation of size 100 which can then be joined with r.

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.

Answer:

A 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.