CHAPTER 14

Transactions

This chapter provides an overview of transaction processing. It first motivates the problems of atomicity, consistency, isolation and durability, and introduces the notion of ACID transactions. It then presents some naive schemes, and their drawbacks, thereby motivating the techniques described in Chapters 15 and 16. The rest of the chapter describes the notion of schedules and the concept of serializability.

We strongly recommend covering this chapter in a first course on databases, since it introduces concepts that every database student should be aware of. Details on how to implement the transaction properties are covered in Chapters 15 and 16.

In the initial presentation to the ACID requirements, the isolation requirement on concurrent transactions does not insist on serializability. Following Haerder and Reuter [1983], isolation just requires that the events within a transaction must be hidden from other transactions running concurrently, in order to allow rollback. However, later in the chapter, and in most of the book (except in Chapter 26), we use the stronger condition of serializability as a requirement on concurrent transactions.

Exercises

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 T_i and T_j, it appears to T_i that either T_j finished execution before T_i started, or T_j started execution after T_i 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 program errors 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 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.

14.15 Consider the following two transactions:

T_{13}: read(A);
read(B);
if $A = 0$ **then** $B := B + 1$;
write(B).
T_{14}: 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 T_{13} and T_{14} that produces a nonserializable schedule.

c. Is there a concurrent execution of T_{13} and T_{14} that produces a serializable schedule?

Answer:

a. There are two possible executions: T_{13} T_{14} and T_{14} T_{13}.

Case 1:

	A	B
initially	0	0
after T_{13}	0	1
after T_{14}	0	1

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

Case 2:

	A	B
initially	0	0
after T_{14}	1	0
after T_{13}	1	0

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

b. Any interleaving of T_{13} and T_{14} results in a non-serializable schedule.

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)	

T_{13}	T_{14}
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_{13} **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_{14} prior to completion of T_{13}. 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_{14} **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:

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

As we can see, the above schedule is serializable with an equivalent serial schedule T_1, T_2. In the above schedule T_2 commits before T_1. Note that the unlock instruction is added to show how this schedule can occur even

with strict two-phase locking, where exclusive locks are held to commit, but shared locks can be released early in two-phase manner.

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 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.

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?

Answer: 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.

14.19 Explain why the read-committed isolation level ensures that schedules are cascade-free.

Answer:
The read-committed isolation level ensures that a transaction reads only the committed data. A transaction T_i can not read a data item X which has been modified by a yet uncommitted concurrent transaction T_j. This makes T_i independent of the success or failure of T_j. Hence, the schedules which follow read committed isolation level become cascade-free.

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

b. Read committed

c. Repeatable read

Answer:

a. Read Uncommitted:

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

In the above schedule, T_2 reads the value of A written by T_1 even before T_1 commits. This schedule is not serializable since T_1 also reads a value written by T_2, resulting in a cycle in the precedence graph.

b. Read Committed:

T_1	T_2
lock-S(A)	
read(A)	
unlock(A)	
	lock-X(A)
	write(A)
	unlock(A)
	commit
lock-S(A)	
read(A)	
unlock-S(A)	
commit	

In the above schedule, the first time T_1 reads A, it sees a value of A before it was written by T_2, while the second **read**(A) by T_1 sees the value written by T_2 (which has already committed). The first read results in T_1 preceding T_2, while the second read results in T_2 preceding T_1, and thus the schedule is not serializable.

c. Repeatable Read :
Consider the following schedule, where T_1 reads all tuples in r satisfying predicate P; to satisfy repeatable read, it must also share-lock these tuples in a two-phase manner.

T_1	T_2
pred_read(r, P)	
	insert(t)
	write(A)
	commit
read(A)	
commit	

Suppose that the tuple t inserted by T_2 satisfies P; then the insert by T_2 causes T_2 to serialized after T_1, since T_1 does not see t. However, the final **read**(A) operation of T_1 forces T_2 to precede T_1, causing a cycle in the precedence graph.

14.21 Suppose that in addition to the operations read and write, we allow an operation pred_read(r, P), which reads all tuples in relation r that satisfy predicate P.

a. Give an example of a schedule using the pred_read operation that exhibits the phantom phenomenon, and is non-serializable as a result.

b. Give an example of a schedule where one transaction uses the pred_read operation on relation r and another concurrent transactions deletes a tuple from r, but the schedule does not exhibit a phantom conflict. (To do so, you have to give the schema of relation r, and show the attribute values of the deleted tuple.)

Answer:

a. The repeatable read schedule in the preceding question is an example of a schedule exhibiting the phantom phenomenon and is non-serializable.

b. Consider the schedule

T_1	T_2
pred_read(r, $r.A$=5)	
	delete(t)
	write(B)
	commit
read(B)	
commit	

Suppose that tuple t deleted by T_2 is from relation r, but does not satisfy predicate P, for example because its A value is 3. Then, there is no phantom conflict between T_1 and T_2, and T_2 can be serialized before T_1.