CHAPTER 15

Concurrency Control

This chapter describes how to control concurrent execution in a database, in order to ensure the isolation properties of transactions. A variety of protocols are described for this purpose. If time is short, some of the protocols may be omitted. We recommend covering, at the least, two-phase locking (Sections 15.1.1), through 15.1.3, deadlock detection and recovery (Section 15.2, omitting Section 15.2.1), and the phantom phenomenon (Section 15.8.3). The most widely used techniques would thereby be covered.

It is worthwhile pointing out how the graph-based locking protocols generalize simple protocols, such as ordered acquisition of locks, which students may have studied in an operating system course. Although the timestamp protocols by themselves are not widely used, multiversion two-phase locking (Section 15.6.2) is of increasing importance since it allows long read-only transactions to run concurrently with updates.

The phantom phenomenon is often misunderstood by students as showing that two-phase locking is incorrect. It is worth stressing that transactions that scan a relation must read some data to find out what tuples are in the relation; as long as this data is itself locked in a two-phase manner, the phantom phenomenon will not arise.

Exercises

15.20 What benefit does strict two-phase locking provide? What disadvantages result?

Answer: Because it produces only cascadeless schedules, recovery is very easy. But the set of schedules obtainable is a subset of those obtainable from plain two phase locking, thus concurrency is reduced.

15.21 Most implementations of database systems use strict two-phase locking. Suggest three reasons for the popularity of this protocol.

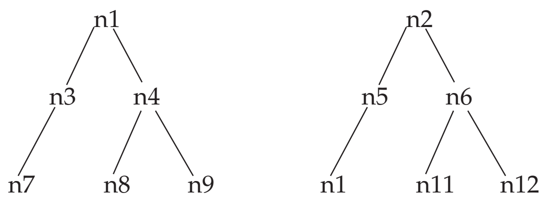
Answer: It is relatively simple to implement, imposes low rollback overhead because of cascadeless schedules, and usually allows an acceptable level of concurrency.

15.22 Consider a variant of the tree protocol called the *forest* protocol. The database is organized as a forest of rooted trees. Each transaction T_i must follow the following rules:

- The first lock in each tree may be on any data item.
- The second, and all subsequent, locks in a tree may be requested only if the parent of the requested node is currently locked.
- Data items may be unlocked at any time.
- A data item may not be relocked by T_i after it has been unlocked by T_i.

Show that the forest protocol does *not* ensure serializability.

Answer: Take a system with 2 trees:



We have 2 transactions, T_1 and T_2. Consider the following legal schedule:

T_1	T_2
lock(n1)	
lock(n3)	
write(n3)	
unlock(n3)	
	lock(n2)
	lock(n5)
	write(n5)
	unlock(n5)
lock(n5)	
read(n5)	
unlock(n5)	
unlock(n1)	
	lock(n3)
	read(n3)
	unlock(n3)
	unlock(n2)

This schedule is not serializable.

15.23 Under what conditions is it less expensive to avoid deadlock than to allow deadlocks to occur and then to detect them?
Answer: Deadlock avoidance is preferable if the consequences of abort are serious (as in interactive transactions), and if there is high contention and a resulting high probability of deadlock.

15.24 If deadlock is avoided by deadlock-avoidance schemes, is starvation still possible? Explain your answer.
Answer: A transaction may become the victim of deadlock-prevention rollback arbitrarily many times, thus creating a potential starvation situation.

15.25 In multiple-granularity locking, what is the difference between implicit and explicit locking?
Answer: When a transaction *explicitly* locks a node in shared or exclusive mode, it *implicitly* locks all the descendents of that node in the same mode. The transaction need not explicitly lock the descendent nodes. There is no difference in the functionalities of these locks, the only difference is in the way they are acquired, and their presence tested.

15.26 Although SIX mode is useful in multiple-granularity locking, an exclusive and intention-shared (XIS) mode is of no use. Why is it useless?
Answer: An exclusive lock is incompatible with any other lock kind. Once a node is locked in exclusive mode, none of the descendents can be simultaneously accessed by any other transaction in any mode. Therefore an exclusive and intend-shared declaration has no meaning.

15.27 The multiple-granularity protocol rules specify that a transaction T_i can lock a node Q in S or IS mode only if T_i currently has the parent of Q locked in either IX or IS mode. Given that SIX and S locks are stronger than IX or IS locks, why does the protocol not allow locking a node in S or IS mode if the parent is locked in either SIX or S mode?
Answer:
If T_i has locked the parent node P in S or SIX mode then it means it has implicit S locks on all the descendent nodes of the parent node including Q. So, there is no need for locking Q in S or IS mode and the protocol does not allow doing that.

15.28 When a transaction is rolled back under timestamp ordering, it is assigned a new timestamp. Why can it not simply keep its old timestamp?
Answer: A transaction is rolled back because a newer transaction has read or written the data which it was supposed to write. If the rolled back transaction is re-introduced with the same timestamp, the same reason for rollback is still valid, and the transaction will have be rolled back again. This will continue indefinitely.

15.29 Show that there are schedules that are possible under the two-phase locking protocol, but are not possible under the timestamp protocol, and vice versa.

Answer: A schedule which is allowed in the two-phase locking protocol but not in the timestamp protocol is:

step	T_0	T_1	*Precedence marks*
1	**lock-S**(A)		
2	**read**(A)		
3		**lock-X**(B)	
4		**write**(B)	
5		**unlock**(B)	
6	**lock-S**(B)		
7	**read**(B)		$T_1 \rightarrow T_0$
8	**unlock**(A)		
9	**unlock**(B)		

This schedule is not allowed in the timestamp protocol because at step 7, the W-timestamp of B is 1.

A schedule which is allowed in the timestamp protocol but not in the two-phase locking protocol is:

step	T_0	T_1	T_2
1	**write**(A)		
2		**write**(A)	
3			**write**(A)
4	**write**(B)		
5		**write**(B)	

This schedule cannot have lock instructions added to make it legal under two-phase locking protocol because T_1 must unlock (A) between steps 2 and 3, and must lock (B) between steps 4 and 5.

15.30 Under a modified version of the timestamp protocol, we require that a commit bit be tested to see whether a read request must wait. Explain how the commit bit can prevent cascading abort. Why is this test not necessary for write requests?

Answer: Using the commit bit, a read request is made to wait if the transaction which wrote the data item has not yet committed. Therefore, if the writing transaction fails before commit, we can abort that transaction alone. The waiting read will then access the earlier version in case of a multiversion system, or the restored value of the data item after abort in case of a single-version system. For writes, this commit bit checking is unnecessary. That is because either the write is a "blind" write and thus independent of the old value of the data item or there was a prior read, in which case the test was already applied.

15.31 As discussed in Exercise 15.19, snapshot isolation can be implemented using a form of timestamp validation. However, unlike the multiversion timestamp-ordering scheme, which guarantees serializability, snapshot isolation does not guarantee serializability. Explain what is the key difference between the protocols that results in this difference.

Answer:

The timestamp validation step for the snapshot isolation level checks for the presence of common written data items between the transactions. However, write skew can occur, where a transaction T_1 updates an item A whose old version is read by T_2, while T_2 updates an item B whose old version is read by T_1, resulting in a non-serializable execution. There is no validation of reads against writes in the snapshot isolation protocol.

The multiversion timestamp-ordering protocol on the other hand avoids the write skew problem by rolling back a transaction that writes a data item which has been already read by a transaction with a higher timestamp.

15.32 Outline the key similarities and differences between the timestamp based implementation of the first-committer-wins version of snapshot isolation, described in Exercise 15.19, and the optimistic-concurrency-control-without-read-validation scheme, described in Section 15.9.3.

Answer: Both the schemes do not ensure serializability. The version number check in the optimistic-concurrency- control-without-read-validation implements the first committer-wins rule used in the snapshot isolation.

Unlike the snapshot isolation, the reads performed by a transaction in optimistic-concurrency- control-without-read-validation may not correspond to the snapshot of the database. Different reads by the same transaction may return data values corresponding to different snapshots of the database.

15.33 Explain the phantom phenomenon. Why may this phenomenon lead to an incorrect concurrent execution despite the use of the two-phase locking protocol?

Answer: The phantom phenomenon arises when, due to an insertion or deletion, two transactions logically conflict despite not locking any data items in common. The insertion case is described in the book. Deletion can also lead to this phenomenon. Suppose T_i deletes a tuple from a relation while T_j scans the relation. If T_i deletes the tuple and then T_j reads the relation, T_i should be serialized before T_j. Yet there is no tuple that both T_i and T_j conflict on.

An interpretation of 2PL as just locking the accessed tuples in a relation is incorrect. There is also an index or a relation data that has information about the tuples in the relation. This information is read by any transaction that scans the relation, and modified by transactions that update, or insert into, or delete from the relation. Hence locking must also be performed on the index or relation data, and this will avoid the phantom phenomenon.

15.34 Explain the reason for the use of degree-two consistency. What disadvantages does this approach have?
Answer: The degree-two consistency avoids cascading aborts and offers increased concurrency but the disadvantage is that it does not guarantee serializability and the programmer needs to ensure it.

15.35 Give example schedules to show that with key-value locking, if any of lookup, insert, or delete do not lock the next-key value, the phantom phenomenon could go undetected.
Answer: In the next-key locking technique, every index lookup or insert or delete must not only the keys found within the range (or the single key, in case of a point lookup) but also the next-key value- that is, the key value greater than the last key value that was within the range. Thus, if a transaction attempts to insert a value that was within the range of the index lookup of another transaction, the two transactions would conflict ion the key value next to the inserted key value. The next-key value should be locked to ensure that conflicts with subsequent range lookups of other queries are detected, thereby detecting phantom phenomenon.

15.36 Many transactions update a common item (e.g., the cash balance at a branch), and private items (e.g., individual account balances). Explain how you can increase concurrency (and throughput) by ordering the operations of the transaction.
Answer: The private items can be updated by the individual transacations independently. They can acquire the exclusive locks for the private items (as no other transaction needs it) and update the data items. But the exclusive lock for the common item is shared among all the transactions. The common item should be locked before the transaction decides to update it. And when it holds the lock for the common item, all other transactions should wait till its released. But inorder that the common item is updated correctly, the transaction should follow a certain pattern. A transacation can update its private item as and when it requires, but before updating the private item again, the common item should be updated. So, essentially the private and the common items should be accessed alternately, otherwise the private item's update will not be reflected in the common item.

a. No possibility of deadlock and no starvation. The lock for the common item should be granted based on the time of requests.

b. The schedule is serializable.

15.37 Consider the following locking protocol: All items are numbered, and once an item is unlocked, only higher-numbered items may be locked.

Locks may be released at any time. Only X-locks are used. Show by an example that this protocol does not guarantee serializability.

Answer: We have 2 transactions, T_1 and T_2. Consider the following legal schedule:

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

Explanation: In the given example schedule, lets assume A is a higher numbered item then B.

a. T_i executes write(A) before T_j executes read(A). So, there's a edge T_i->T_j.

b. T_j executes write(B) before T_i executes read(A). So, there's a edge T_j->T_i.

There's a cycle in the graph which means the given schedule is not conflict serializable.