CHAPTER 15

Concurrency Control

Practice Exercises

15.1 **Answer:** Suppose two-phase locking does not ensure serializability. Then there exists a set of transactions T_0, T_1 ... T_{n-1} which obey 2PL and which produce a nonserializable schedule. A non-serializable schedule implies a cycle in the precedence graph, and we shall show that 2PL cannot produce such cycles. Without loss of generality, assume the following cycle exists in the precedence graph: $T_0 \rightarrow T_1 \rightarrow T_2 \rightarrow ... \rightarrow T_{n-1} \rightarrow T_0$. Let α_i be the time at which T_i obtains its last lock (i.e. T_i's lock point). Then for all transactions such that $T_i \rightarrow T_j$, $\alpha_i < \alpha_j$. Then for the cycle we have

$$\alpha_0 < \alpha_1 < \alpha_2 < ... < \alpha_{n-1} < \alpha_0$$

Since $\alpha_0 < \alpha_0$ is a contradiction, no such cycle can exist. Hence 2PL cannot produce non-serializable schedules. Because of the property that for all transactions such that $T_i \rightarrow T_j$, $\alpha_i < \alpha_j$, the lock point ordering of the transactions is also a topological sort ordering of the precedence graph. Thus transactions can be serialized according to their lock points.

15.2 **Answer:**

a. Lock and unlock instructions:

T_{34}:
lock-S(A)
read(A)
lock-X(B)
read(B)
if $A = 0$
then $B := B + 1$
write(B)
unlock(A)
unlock(B)

T_{35}: **lock-S**(B)
read(B)
lock-X(A)
read(A)
if $B = 0$
then $A := A + 1$
write(A)
unlock(B)
unlock(A)

b. Execution of these transactions can result in deadlock. For example, consider the following partial schedule:

T_{31}	T_{32}
lock-S (A)	
	lock-S (B)
	read(B)
read(A)	
lock-X (B)	
	lock-X (A)

The transactions are now deadlocked.

15.3 **Answer:** Rigorous two-phase locking has the advantages of strict 2PL. In addition it has the property that for two conflicting transactions, their commit order is their serializability order. In some systems users might expect this behavior.

15.4 **Answer:**
Consider two nodes A and B, where A is a parent of B. Let dummy vertex D be added between A and B. Consider a case where transaction T_2 has a lock on B, and T_1, which has a lock on A wishes to lock B, and T_3 wishes to lock A. With the original tree, T_1 cannot release the lock on A until it gets the lock on B. With the modified tree, T_1 can get a lock on D, and release the lock on A, which allows T_3 to proceed while T_1 waits for T_2. Thus, the protocol allows locks on vertices to be released earlier to other transactions, instead of holding them when waiting for a lock on a child. A generalization of idea based on edge locks is described in Buckley and Silberschatz, "Concurrency Control in Graph Protocols by Using Edge Locks," Proc. ACM SIGACT-SIGMOD Symposium on the Principles of Database Systems, 1984.

15.5 **Answer:** Consider the tree-structured database graph given below.

Schedule possible under tree protocol but not under 2PL:

T_1	T_2
lock *(A)*	
lock *(B)*	
unlock *(A)*	
	lock *(A)*
lock *(C)*	
unlock *(B)*	
	lock *(B)*
	unlock *(A)*
	unlock *(B)*
unlock *(C)*	

Schedule possible under 2PL but not under tree protocol:

T_1	T_2
lock *(A)*	
	lock *(B)*
lock *(C)*	
	unlock *(B)*
unlock *(A)*	
unlock *(C)*	

15.6 **Answer:**

The proof is in Kedem and Silberschatz, "Locking Protocols: From Exclusive to Shared Locks," JACM Vol. 30, 4, 1983. (The proof of serializability and deadlock freedom of the original tree protocol may be found in Silberschatz and Kedem, "Consistency in Hierarchical Database Systems", JACM Vol. 27, 1, Jan 1980.)

It is worth noting that if the protocol were modified to allow update transactions to lock any data item first, non-serializable executions can occur. Intuitively, a long running readonly transaction T_0 may precede an update transaction T_1 on node a, but continues to holds a lock on child node b. After T_1 updates a and commits, readonly transaction T_2 reads a, and thus T_1 precedes T_2. However, T_2 now overtakes T_1, share locking b and its child c (not possible with only exclusive locks), and reads c and

commits. Subsequently, transaction T_3 updates c and commits, after which T_0 share locks and reads c. Thus, there is a cycle

$$T_0 \rightarrow T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow T_0$$

In effect, two readonly transactions see two different orderings of the same two update transactions. By requiring update transactions to always start at the root, the protocol ensures that the above situation cannot occur. In the above example T_3 would be forced to start from the root, and thus T_0 cannot be serialized after T_3.
The formal proof is by induction on the number of read-only transactions (and, we must admit, not very intuitive). We present it after introducing some notation first:
Q^X : the set of update transactions.
Q^S :the set of Read-only transactions.
$L(T_i)$: the set of data items locked by transaction T_i
Let Q the new protocol. Suppose T_i and T_k are update transactions, and T_j a read only transaction; then if T_j overlaps with T_i and T_k in the set of data items it accesses, since T_i and T_k both start from the root, T_i and T_k must also overlap on the set of data items accessed. Formally,
$\forall T_i \in Q^X \forall T_j \in Q^S \forall T_k \in Q^X$
$[((L(T_i) \bigcap L(T_j)) \neq \phi \wedge (L(T_j) \bigcap L(T_k)) \neq \phi) \Rightarrow (L(T_i) \bigcap L(T_k)) \neq \phi]$
Consider an arbitrary schedule of transactions in protocol Q. Let k be the number of Read-only transactions which participate in this schedule. We show by induction that there exist no minimal cycles in $(T, \rightarrow)$, where $T_i \rightarrow T_j$ denotes that T_i precedes T_j due to a conflict on some data item.
Let the cycle without loss of generality be,
$T_0 \rightarrow T_1 ... \rightarrow T_{m-1} \rightarrow T_0$
Here, we know that $m \geq 2k$, as there must be an Update transaction on each side of a Read-only transaction, otherwise the cycle will not be formed.

- $k = 0$: In this case each T_i is in Q^X, so Q becomes the original tree protocol which ensures the serializability and deadlock freedom.
- $k = 1$: Let T_i be the unique transaction in Q^S. We can replace T_i by T_i^X. The resulting schedule follows the original tree protocol which ensures serializability and deadlock freedom.
- $k > 1$: Let $T_i \wedge Q^S$, then $\{T_{i-1}, T_{i+1}\} \in Q^X$. As, $m \geq 2k \geq 4, i-1 \neq i+1$ (modulo m). By the assumption of the given protocol's definition,
  $L(T_{i-1}) \bigcap L(T_{i+1}) \neq \phi$
  Thus, $T_{i-1} \rightarrow T_{i+1}$ or $T_{i+1} \rightarrow T_{i-1}$; in either case, the original cycle was not minimal (in the first case, T_i can be deleted from the cycle, in the second case there is a cycle involving T_{i-i}, T_i and T_{i+1}.

By contradiction, there can be no such cycle, and thus the new protocol ensures serializability.

To show deadlock freedom, consider the waits for graph. If there is any edge from an update transaction T_j to another update transaction T_i, it is easy to see that T_i locks the root before T_j. Similarly if there is an edge from an update txn T_j to any readonly transaction T_k, which in turn can only wait for an update transaction T_i, it can be seen that T_i must have locked the root before T_j. In other words, an older update transaction cannot wait (directly or indirectly) on a younger update transaction, and thus, there can be no cycle in the waits for graph.

15.7 **Answer:**
The proof is in Kedem and Silberschatz, "Controlling Concurrency Using Locking Protocols," Proc. Annual IEEE Symposium on Foundations of Computer Science, 1979. The proof is quite non-trivial, and you may skip the details if you wish.
Consider,

- $G(V, A)$: the directed acyclic graph of the data items.
- T_0, T_1, T_2,.....,T_m are the participating transactions.
- $E(T_i)$ is the first vertex locked by transaction T_i.
- $\eta : V \rightarrow \{1,2,3,.... \}$ such that for each $u, w \in V$ if $< u, w > \in A$ then $\eta(u) < \eta(w)$
- $F(T_i, T_j)$ is that $v \in L(T_i) \bigcap L(T_j)$ for which $\eta(v)$ is minimal
- For each $e \in V$, define a relation $\omega_e \subseteq T \times T$ such that $T_i \omega_e T_j$ for $e \in L(T_i) \bigcap L(T_j)$ iff T_i successfully locked e and T_j either never (successfully) locked e or locked e, only after T_i unlocked it.
- Define also, $T_i \omega T_j = \exists e[T_i \omega_e T_j]$

a. **Lemma 1** We first show that, for the given protocol,
If $L(T_i) \bigcap L(T_j) \neq \phi$ then $F(T_i, T_j) \in \{E(T_i), E(T_j)\}$
Assume by contradiction that,
$F(T_i, T_j) \notin \{E(T_i), E(T_j)\}$
But then, by the locking protocol (as both T_i and T_j) had to lock more than **half** of the predecessors of $F(T_i, T_j)$), it follows that some predecessor of $F(T_i, T_j)$ is in $L(T_i) \bigcap L(T_j)$ contradicting the definition of $F(T_i, T_j)$.

b. **Lemma 2** Now we show that,
$T_i \omega_u T_j$ for all $u \in L(T_i) \bigcap L(T_j)$ iff $T_i \omega_{F(T_i,T_j)} T_j$.

If $u = F(T_i, T_j)$ then $T_i \omega_{F(T_i,T_j)} T_j$. is trivially true. If $u \neq F(T_i, T_j)$, then $u \notin \{E(T_i), E(T_j)\}$. It thus follows that some predecessor of w of u was successfully locked by both T_i and T_j and this u was

locked by T_i and T_j when they issued the instructions to lock u. Thus, $T_i \omega_w T_j \Leftrightarrow T_i \omega_u T_j$ (and $T_j \omega_w T_i \Leftrightarrow T_j \omega_u T_i$).
Now by induction: $T_i \omega_{F(T_i,T_j)} T_j \Leftrightarrow T_i \omega_w T_j$ and the result follows.

Now, we prove that the given protocol ensures serializability and deadlock freedom by induction on the length of minimal cycle.

a. $m = 2$: The protocol ensures no minimal cycles as shown in the above Lemma 2

b. $m > 2$: Assume by contradiction that
$T_0 \omega T_1 \omega T_2 \ldots\ldots \omega T_{m-1} \omega T_0$ is a minimal cycle of length m. We will consider two cases:

i. $F(T_i, T_{i+1})$'s are not all distinct. It follows that,

$$L(T_i) \bigcap L(T_j) \bigcap L(T_k) \neq \phi \text{ for } 0 \leq i \leq j \leq k \leq m-1$$

A. Assume $< i, j, k > = < i, i+1, i+2 >$. Then it easily follows that $T_i \omega T_{i+2}$ and (*) is not a minimal cycle.

B. Assume $< i, j, k > \neq < i, i+1, i+2 >$. If say $|j - i| > 1$ then as either $T_i \omega T_j$ or $T_j \omega T_i$ and (*) is not a minimal cycle. If $|j - i| = 1$ and $|k - j| > 1$ then the proof is analogous..

ii. All $F(T_i, T_{i+1})$'s are distinct. Then for some i

$$\eta(F(T_i, T_{i+1})) < \eta(F(T_{i+1}, T_{i+2})) \qquad (**)$$

$$\text{and } \eta(F(T_{i+1}, T_{i+2})) > \eta(F(T_{i+2}, T_{i+3})) \qquad (***)$$

By (**), $E(T_{i+1}) \neq F(T_{i+1}, T_{i+2})$ and by (***), $E(T_{i+2}) \neq F(T_{i+1}, T_{i+2})$.

Thus, $F(T_{i+1}, T_{i+2}) \notin \{E(T_{i+1}), E(T_{i+2})\}$, a contradiction to Lemma 1.

We have thus shown that the given protocol ensures serializability and deadlock freedom.

15.8 Answer:
The proof is Silberschatz and Kedem, "A Family of Locking Protocols for Database Systems that Are Modeled by Directed Graphs", IEEE Trans. on Software Engg. Vol. SE-8, No. 6, Nov 1982.
The proof is rather complex; we omit details, which may be found in the above paper.

15.9 **Answer:** The access protection mechanism can be used to implement page level locking. Consider reads first. A process is allowed to read a page only after it read-locks the page. This is implemented by using `mprotect` to initially turn off read permissions to all pages, for the process. When the process tries to access an address in a page, a protection violation occurs. The handler associated with protection violation then requests a read lock on the page, and after the lock is acquired, it uses `mprotect` to allow read access to the page by the process, and finally allows the process to continue. Write access is handled similarly.

15.10 **Answer:**

a. Serializability can be shown by observing that if two transactions have an I mode lock on the same item, the increment operations can be swapped, just like read operations. However, any pair of conflicting operations must be serialized in the order of the lock points of the corresponding transactions, as shown in Exercise 15.1.

b. The **increment** lock mode being compatible with itself allows multiple incrementing transactions to take the lock simultaneously thereby improving the concurrency of the protocol. In the absence of this mode, an **exclusive** mode will have to be taken on a data item by each transaction that wants to increment the value of this data item. An exclusive lock being incompatible with itself adds to the lock waiting time and obstructs the overall progress of the concurrent schedule.
In general, increasing the **true** entries in the compatibility matrix increases the concurrency and improves the throughput.

The proof is in Korth, "Locking Primitives in a Database System," JACM Vol. 30, 1983.

15.11 **Answer:** It would make no difference. The write protocol is such that the most recent transaction to write an item is also the one with the largest timestamp to have done so.

15.12 **Answer:** If a transaction needs to access a large a set of items, multiple granularity locking requires fewer locks, whereas if only one item needs to be accessed, the single lock granularity system allows this with just one lock. Because all the desired data items are locked and unlocked together in the multiple granularity scheme, the locking overhead is low, but concurrency is also reduced.

15.13 **Answer:** In the concurrency control scheme of Section 15.5 choosing **Start**(T_i) as the timestamp of T_i gives a subset of the schedules allowed by choosing **Validation**(T_i) as the timestamp. Using **Start**(T_i) means that whoever started first must finish first. Clearly transactions could enter the validation phase in the same order in which they began executing,

but this is overly restrictive. Since choosing **Validation**(T_i) causes fewer nonconflicting transactions to restart, it gives the better response times.

15.14 **Answer:**

- Two-phase locking: Use for simple applications where a single granularity is acceptable. If there are large read-only transactions, multiversion protocols would do better. Also, if deadlocks must be avoided at all costs, the tree protocol would be preferable.
- Two-phase locking with multiple granularity locking: Use for an application mix where some applications access individual records and others access whole relations or substantial parts thereof. The drawbacks of 2PL mentioned above also apply to this one.
- The tree protocol: Use if all applications tend to access data items in an order consistent with a particular partial order. This protocol is free of deadlocks, but transactions will often have to lock unwanted nodes in order to access the desired nodes.
- Timestamp ordering: Use if the application demands a concurrent execution that is equivalent to a particular serial ordering (say, the order of arrival), rather than *any* serial ordering. But conflicts are handled by roll-back of transactions rather than waiting, and schedules are not recoverable. To make them recoverable, additional overheads and increased response time have to be tolerated. Not suitable if there are long read-only transactions, since they will starve. Deadlocks are absent.
- Validation: If the probability that two concurrently executing transactions conflict is low, this protocol can be used advantageously to get better concurrency and good response times with low overheads. Not suitable under high contention, when a lot of wasted work will be done.
- Multiversion timestamp ordering: Use if timestamp ordering is appropriate but it is desirable for read requests to never wait. Shares the other disadvantages of the timestamp ordering protocol.
- Multiversion two-phase locking: This protocol allows read-only transactions to always commit without ever waiting. Update transactions follow 2PL, thus allowing recoverable schedules with conflicts solved by waiting rather than roll-back. But the problem of deadlocks comes back, though read-only transactions cannot get involved in them. Keeping multiple versions adds space and time overheads though, therefore plain 2PL may be preferable in low conflict situations.

15.15 **Answer:** A transaction waits on *a.* disk I/O and *b.* lock acquisition. Transactions generally wait on disk reads and not on disk writes as disk

writes are handled by the buffering mechanism in asynchronous fashion and transactions update only the in-memory copy of the disk blocks.

The technique proposed essentially separates the waiting times into two phases. The first phase – where transaction is executed without acquiring any locks and without performing any writes to the database – accounts for almost all the waiting time on disk I/O as it reads all the data blocks it needs from disk if they are not already in memory. The second phase—the transaction re-execution with strict two-phase locking—accounts for all the waiting time on acquiring locks. The second phase may, though rarely, involve a small waiting time on disk I/O if a disk block that the transaction needs is flushed to memory (by buffer manager) before the second phase starts.

The technique may increase concurrency as transactions spend almost no time on disk I/O with locks held and hence locks are held for shorter time. In the first phase the transaction reads all the data items required —and not already in memory—from disk. The locks are acquired in the second phase and the transaction does almost no disk I/O in this phase. Thus the transaction avoids spending time in disk I/O with locks held.

The technique may even increase disk throughput as the disk I/O is not stalled for want of a lock. Consider the following scenario with strict two-phase locking protocol: A transaction is waiting for a lock, the disk is idle and there are some item to be read from disk. In such a situation disk bandwidth is wasted. But in the proposed technique, the transaction will read all the required item from the disk without acquiring any lock and the disk bandwidth may be properly utilized.

Note that the proposed technique is most useful if the computation involved in the transactions is less and most of the time is spent in disk I/O and waiting on locks, as is usually the case in disk-resident databases. If the transaction is computation intensive, there may be wasted work. An optimization is to save the updates of transactions in a temporary buffer, and instead of reexecuting the transaction, to compare the data values of items when they are locked with the values used earlier. If the two values are the same for all items, then the buffered updates of the transaction are executed, instead of reexecuting the entire transaction.

15.16 **Answer:** Consider two transactions T_1 and T_2 shown below.

T_1	T_2
write(p)	
	read(p)
	read(q)
write(q)	

Let TS(T_1) < TS(T_2) and let the timestamp test at each operation except write(q) be successful. When transaction T_1 does the timestamp test for write(q) it finds that TS(T_1) < R-timestamp(q), since TS(T_1) < TS(T_2) and R-timestamp(q) = TS(T_2). Hence the writeoperation fails and transaction T_1 rolls back. The cascading results in transaction T_2 also being rolled back as it uses the value for item p that is written by transaction T_1.
If this scenario is exactly repeated every time the transactions are restarted, this could result in starvation of both transactions.

15.17 **Answer:** In the text, we considered two approaches to dealing with the phantom phenomenon by means of locking. The coarser granularity approach obviously works for timestamps as well. The B^+-tree index based approach can be adapted to timestamping by treating index buckets as data items with timestamps associated with them, and requiring that all read accesses use an index. We now show that this simple method works. Suppose a transaction T_i wants to access all tuples with a particular range of search-key values, using a B^+-tree index on that search-key. T_i will need to read all the buckets in that index which have key values in that range. It can be seen that any delete or insert of a tuple with a key-value in the same range will need to write one of the index buckets read by T_i. Thus the logical conflict is converted to a conflict on an index bucket, and the phantom phenomenon is avoided.

15.18 **Answer:** Note: The tree-protocol of Section 15.1.5 which is referred to in this question, is different from the multigranularity protocol of Section 15.3 and the B^+-tree concurrency protocol of Section 15.10.

One strategy for early lock releasing is given here. Going down the tree from the root, if the currently visited node's child is not full, release locks held on all nodes except the current node, request an X-lock on the child node, after getting it release the lock on the current node, and then descend to the child. On the other hand, if the child is full, retain all locks held, request an X-lock on the child, and descend to it after getting the lock. On reaching the leaf node, start the insertion procedure. This strategy results in holding locks only on the full index tree nodes from the leaf upwards, until and including the first non-full node.

An optimization to the above strategy is possible. Even if the current node's child is full, we can still release the locks on all nodes but the current one. But after getting the X-lock on the child node, we split it right away. Releasing the lock on the current node and retaining just the lock on the appropriate split child, we descend into it making it the current node. With this optimization, at any time at most two locks are held, of a parent and a child node.

15.19 **Answer:**

a. validation test for first-committer-wins scheme: Let Start(T_i), Commit(T_i) and be the timestamps associated with a transaction T_i and the update set for T_i be update_set(T_i). Then for all transactions T_k with

Commit(T_k) < Commit(T_i), one of the following two conditions must hold:

- If Commit(T_k) < Start(T_k), T_k completes its execution before T_i started, the serializability is maintained.
- If Start(T_i) < Commit(T_k) < Commit(T_i) and update_set(T_i) and update_set(T_k) do not intersect

b. Validation test for first-committer-wins scheme with W-timestamps for data items: If a transaction T_i writes a data item Q, then the W-timestamp(Q) is set to Commit(T_i). For the validation test of a transaction T_i to pass, following condition must hold:

- For each data item Q written by T_i, W-timestamp(Q) < Start(T_i)

c. First-updater-wins scheme:

i. For a data item Q written by T_i, the W-timestamp is assigned the timestamp when the write occurred in T_i

ii. Since the validation is done after acquiring the exclusive locks and the exclusive locks are held till the end of the transaction, the data item cannot be modified inbetween the lock acquisition and commit time. So, the result of validation test for a transaction would be the same at the commit time as that at the update time.

iii. Because of the exclusive locking, at the most one transaction can acquire the lock on a data item at a time and do the validation testing. Thus, two or more transactions can not do validation testing for the same data item simultaneously.