CHAPTER 18

Concurrency Control

Practice Exercises

18.1 Show that the two-phase locking protocol ensures conflict serializability and that transactions can be serialized according to their lock points.

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 nonserializable 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 nonserializable 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.

18.2 Consider the following two transactions:

```
$T_{34}$: read($A$);
       read($B$);
       if $A = 0$ then $B := B + 1$;
       write($B$).

$T_{35}$: read($B$);
       read($A$);
       if $B = 0$ then $A := A + 1$;
       write($A$).
```

Add lock and unlock instructions to transactions T_{31} and T_{32} so that they observe the two-phase locking protocol. Can the execution of these transactions result in a deadlock?

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.

18.3 What benefit does rigorous two-phase locking provide? How does it compare with other forms of two-phase locking?

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.

18.4 Consider a database organized in the form of a rooted tree. Suppose that we insert a dummy vertex between each pair of vertices. Show that, if we follow the tree protocol on the new tree, we get better concurrency than if we follow the tree protocol on the original tree.

Answer:

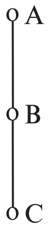
Consider two nodes A and B, where A is a parent of B. Let dummy vertex D be added between A and B. Consider a case where transaction T_2 has a lock on B, and T_1, which has a lock on A wishes to lock B, and T_3 wishes to lock A. With the original tree, T_1 cannot release the lock on A until it gets the 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 the 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* .

18.5 Show by example that there are schedules possible under the tree protocol that are not possible under the two-phase locking protocol, and vice versa.

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)	

18.6 Locking is not done explicitly in persistent programming languages. Rather, objects (or the corresponding pages) must be locked when the objects are accessed. Most modern operating systems allow the user to set access protections (no access, read, write) on pages, and memory access that violate the access protections result in a protection violation (see the Unix mprotect command, for example). Describe how the access-protection mechanism can be used for page-level locking in a persistent programming language.

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.

18.7 Consider a database system that includes an atomic **increment** operation, in addition to the read and write operations. Let V be the value of data item X. The operation

increment(X) by C

sets the value of X to $V + C$ in an atomic step. The value of X is not available to the transaction unless the latter executes a read(X).

Assume that increment operations lock the item in increment mode using the compatibility matrix in Figure 18.25.

a. Show that, if all transactions lock the data that they access in the corresponding mode, then two-phase locking ensures serializability.

b. Show that the inclusion of **increment** mode locks allows for increased concurrency.

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," Journal of the ACM Volume 30, (1983).

18.8 In timestamp ordering, **W-timestamp**(Q) denotes the largest timestamp of any transaction that executed write(Q) successfully. Suppose that, instead, we defined it to be the timestamp of the most recent transaction to execute write(Q)

successfully. Would this change in wording make any difference? Explain your answer.

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.

18.9 Use of multiple-granularity locking may require more or fewer locks than an equivalent system with a single lock granularity. Provide examples of both situations, and compare the relative amount of concurrency allowed.

Answer:

If a transaction needs to access a large 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.

18.10 For each of the following protocols, describe aspects of practical applications that would lead you to suggest using the protocol, and aspects that would suggest not using the protocol:

- Two-phase locking
- Two-phase locking with multiple-granularity locking.
- The tree protocol
- Timestamp ordering
- Validation
- Multiversion timestamp ordering
- Multiversion two-phase locking

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.

18.11 Explain why the following technique for transaction execution may provide better performance than just using strict two-phase locking: First execute the transaction without acquiring any locks and without performing any writes to the database as in the validation-based techniques, but unlike the validation techniques do not perform either validation or writes on the database. Instead, rerun the transaction using strict two-phase locking. (Hint: Consider waits for disk I/O.)

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 reexecution 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 a 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 items 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 items 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.

18.12 Consider the timestamp-ordering protocol, and two transactions, one that writes two data items *p* and *q*, and another that reads the same two data items. Give a schedule whereby the timestamp test for a write operation fails and causes the first transaction to be restarted, in turn causing a cascading abort of the other transaction. Show how this could result in starvation of both transactions. (Such a situation, where two or more processes carry out actions, but are unable to complete their task because of interaction with the other processes, is called a **livelock**.)

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 write operation 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.

18.13 Devise a timestamp-based protocol that avoids the phantom phenomenon.

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.

18.14 Suppose that we use the tree protocol of Section 18.1.5 to manage concurrent access to a B$^+$-tree. Since a split may occur on an insert that affects the root, it appears that an insert operation cannot release any locks until it has completed the entire operation. Under what circumstances is it possible to release a lock earlier?

Answer:

Note: The tree protocol of Section Section 18.1.5 which is referred to in this question is different from the multigranularity protocol of Section 18.3 and the B^+-tree concurrency protocol of Section 18.10.2.

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, then 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 upward, 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.

18.15 The snapshot isolation protocol uses a validation step which, before performing a write of a data item by transaction T, checks if a transaction concurrent with T has already written the data item.

a. A straightforward implementation uses a start timestamp and a commit timestamp for each transaction, in addition to an *update set*, that, is the set of data items updated by the transaction. Explain how to perform validation for the first-committer-wins scheme by using the transaction timestamps along with the update sets. You may assume that validation and other commit processing steps are executed serially, that is, for one transaction at a time,

b. Explain how the validation step can be implemented as part of commit processing for the first-committer-wins scheme, using a modification of the above scheme, where instead of using update sets, each data item has a write timestamp associated with it. Again, you may assume that validation and other commit processing steps are executed serially.

c. The first-updater-wins scheme can be implemented using timestamps as described above, except that validation is done immediately after acquiring an exclusive lock, instead of being done at commit time.

   i. Explain how to assign write timestamps to data items to implement the first-updater-wins scheme.

   ii. Show that as a result of locking, if the validation is repeated at commit time the result would not change.

   iii. Explain why there is no need to perform validation and other commit processing steps serially in this case.

Answer:

a. Validation test for first-committer-wins scheme: Let StartTS(T_i), CommitTS(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 CommitTS(T_k) < CommitTS(T_i), one of the following two conditions must hold:

- If CommitTS(T_k) < StartTS(T_k), T_k completes its execution before T_i started, the serializability is maintained.
- StartTS(T_i) < CommitTS(T_k) < CommitTS(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 CommitTS(T_i). For the validation test of a transaction T_i to pass, the following condition must hold:

- For each data item Q written by T_i, W-timestamp(Q) < StartTS(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 in between the lock acquisition and commit time. So, the result of the 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 most one transaction can acquire the lock on a data item at a time and do the validation testing. Thus, two or more transactions cannot do validation testing for the same data item simultaneously.

18.16 Consider functions *insert_latchfree*() and *delete_latchfree*(), shown in Figure 18.23.

a. Explain how the ABA problem can occur if a deleted node is reinserted.

b. Suppose that adjacent to *head* we store a counter *cnt*. Also suppose that DCAS((*head,cnt*), (*oldhead, oldcnt*), (*newhead, newcnt*)) atomically performs a compare-and-swap on the 128 bit value (*head,cnt*). Modify the *insert_latchfree*() and *delete_latchfree*() to use the DCAS operation to avoid the ABA problem.

c. Since most processors use only 48 bits of a 64 bit address to actually address memory, explain how the other 16 bits can be used to implement a counter, in case the DCAS operation is not supported.

Answer:

a. Let the head of the list be pointer $n1$, and the next three elements be $n2$ and $n3$. Suppose process $P1$ which is performing a delete, reads pointer

$n1$ as head and $n2$ as *newhead*, but before it executes CAS(*head*, $n1$, $n2$), process $P2$ deletes $n1$, then deletes $n2$ and then inserts $n1$ back at the head.

The CAS would replace $n1$ by a pointer to $n2$, since the head is still $n1$. However, node $n2$ has meanwhile been deleted and is garbage. Thus, the list is now inconsistent.

b. The following code

```
atomic_read(head, cnt) {
        repeat
                oldhead = head
                oldcnt = cnt
                result = DCAS((head, cnt), (oldhead, oldcnt), (oldhead, oldcnt))
        until (result == success)
        return (oldhead, oldcnt)
}

insert_latchfree(head, value) {
        node = new node
        node−>value = value
        repeat
                (oldhead, oldcnt) = atomic_read(head, cnt)
                node−>next = oldhead
                newcnt = oldcnt+1
                result = DCAS(head, (oldhead, oldcnt), (node, newcnt))
        until (result == success)
}

delete_latchfree(head) {
        /* This function is not quite safe; see explanation in text. */
        repeat
                (oldhead, oldcnt) = atomic_read(head, cnt)
                newhead = oldhead−>next
                newcnt = oldcnt+1
                result = DCAS(head, (oldhead, oldcnt), (newhead, newcnt))
        until (result == success)
}
```

The *atomic_read* function ensures that the 128 bit address, counter pair is read atomically, by using the DCAS instruction to ensure that the values are still same (the DCAS instruction stores the same values back if it succeeds, so there is no change in the value). If the DCAS fails, we may have read an old pointer and a new value, or vice versa, requiring the values to be read again.

The ABA problem would be avoided by the modified code for *insert_latchfree*() and *delete_latchfree*(), since although the reinsert of the $n1$ by $P2$ would result in the head having the same pointer $n1$ as earlier, counter *cnt* would be different from *oldcnt*, resulting in the CAS operation of $P1$ failing.

c. Most processors use only the last 48 bits of a 64 bit address to access memory (which can support 256 Terabytes of memory). The first 16 bits of a 64 bit value can then be used as a counter, and the last 48 bits as the address, with the counter and the address extracted using bit-and operations before being used, and using bit-and and bit-or operations to reconstruct the 64 bit value from a pointer and a counter. If a hardware implementation does not support DCAS, this could be used as an alternative to a DCAS, although it still runs a the small risk of the counter wrapping around if there are exactly 64K other operations on the list between the read of the head and the CAS operation.