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.

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_{34} and T_{35} so that they observe the two-phase locking protocol. Can the execution of these transactions result in a deadlock?

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

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.

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.

18.6 Locking is not done explicitly in persistent programming languages. Rather, objects (or the corresponding pages) must be locked when the objects are ac-

cessed. 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.

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.

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.

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.

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

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.)

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**.)

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

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?

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.

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.