CHAPTER 25

Advanced Transaction Processing

Solutions to Practice Exercises

25.1 **a.** The tasks in a workflow have dependencies based on their status. For example the starting of a task may be conditional on the outcome (such as commit or abort) of some other task. All the tasks cannot execute independently and concurrently, using 2PC just for atomic commit.

b. Once a task gets over, it will have to expose its updates, so that other tasks running on the same processing entity don't have to wait for long. 2PL is too strict a form of concurrency control, and is not appropriate for workflows.

c. Workflows have their own consistency requirements; that is, failure-atomicity. An execution of a workflow must finish in an *acceptable termination state*. Because of this, and because of early exposure of uncommitted updates, the recovery procedure will be quite different. Some form of logical logging and compensation transactions will have to be used. Also to perform forward recovery of a failed workflow, the recovery routines need to restore the state information of the scheduler and tasks, not just the updated data items. Thus simple WAL cannot be used.

25.2

- Loading the entire database into memory in advance can provide transactions which need high-speed or realtime data access the guarantee that once they start they will not have to wait for disk accesses to fetch data. However no transaction can run till the entire database is loaded.
- The advantage in loading on demand is that transaction processing can start rightaway; however transactions may see long and unpredictable delays in disk access until the entire database is loaded into memory.

25.3 A high-performance system is not necessarily a real-time system. In a high performance system, the main aim is to execute each transaction as quickly as possible, by having more resources and better utilization. Thus average speed and response time are the main things to be optimized. In a real-time system, speed is not the central issue. Here *each* transaction has a deadline, and taking care that it finishes within the deadline or takes as little extra time as possible, is the critical issue.

25.4 In the presence of long-duration transactions, trying to ensure serializability has several drawbacks:-

a. With a waiting scheme for concurrency control, long-duration transactions will force long waiting times. This means that response time will be high, concurrency will be low, so throughput will suffer. The probability of deadlocks is also increased.

b. With a time-stamp based scheme, a lot of work done by a long-running transaction will be wasted if it has to abort.

c. Long duration transactions are usually interactive in nature, and it is very difficult to enforce serializability with interactiveness.

Thus the serializability requirement is impractical. Some other notion of database consistency has to be used in order to support long duration transactions.

25.5 Each thread can be modeled as a transaction T which takes a message from the queue and delivers it. We can write transaction T as a multilevel transaction with subtransactions T_1 and T_2. Subtransaction T_1 removes a message from the queue and subtransaction T_2 delivers it. Each subtransaction releases locks once it completes, allowing other transactions to access the queue. If transaction T_2 fails to deliver the message, transaction T_1 will be undone by invoking a compensating transaction which will restore the message to the queue.

25.6 Consider the advanced recovery algorithm of Section 17.8. The redo pass, which repeats history, is the same as before. We discuss below how the undo pass is handled.

- **Recovery with nested transactions**:

  Each subtransaction needs to have a unique TID, because a failed subtransaction might have to be independently rolled back and restarted.

  If a subtransaction fails, the recovery actions depend on whether the unfinished upper-level transaction should be aborted or continued. If it should be aborted, all finished and unfinished subtransactions are undone by a backward scan of the log (this is possible because the locks on the modified data items are not released as soon as a subtransaction finishes). If the nested transaction is going to be continued, just the failed transaction is undone, and then the upper-level transaction continues.

  In the case of a system failure, depending on the application, the entire nested-transaction may need to be aborted, or, (for e.g., in the case of long duration transactions) incomplete subtransactions aborted, and the nested transaction resumed. If the nested-transaction must be aborted, the rollback can be done in the usual manner by the recovery algorithm, during the undo pass. If the nested-transaction must be restarted, any incomplete subtransactions that need to be rolled back can be rolled back as above. To restart the nested-transaction, state information about the transaction, such as locks held and execution state, must have been noted on the log, and must restored during recovery. Mini-batch transactions (discussed in Section 23.1.8) are an example of nested transactions that must be restarted.

- **Recovery with multi-level transactions**:

  In addition to what is done in the previous case, we have to handle the problems caused by exposure of updates performed by committed subtransactions of incomplete upper-level transactions. A committed subtransaction may have released locks that it held, so the compensating transaction has to reacquire the locks. This is straightforward in the case of transaction failure, but is more complicated in the case of system failure.

  The problem is, a lower level subtransaction a of a higher level transaction A may have released locks, which have to be reacquired to compensate A during recovery. Unfortunately, there may be some other lower level subtransaction b of a higher level transaction B that started and acquired the locks released by a, before the end of A. Thus undo records for b may precede the operation commit record for A. But if b had not finished at the time of the system failure, it must first be rolled back and its locks released, to allow the compensating transaction of A to reacquire the locks.

  This complicates the undo pass; it can no longer be done in one backward scan of the log. Multilevel recovery is described in detail in David Lomet, "MLR: A Recovery Method for Multi-Level Systems", ACM SIGMOD Conf. on the Management of Data 1992, San Diego.

25.7 **a.** We can have a special data item at some site on which a lock will have to be obtained before starting a global transaction. The lock should be released after the transaction completes. This ensures the single active global transaction requirement. To reduce dependency on that particular site being up, we can generalize the solution by having an election scheme to choose one of the currently up sites to be the co-ordinator, and requiring that the lock be requested on the data item which resides on the currently elected co-ordinator.

b. The following schedule involves two sites and four transactions. T_1 and T_2 are local transactions, running at site 1 and site 2 respectively. T_{G1} and T_{G2} are global transactions running at both sites. X_1, Y_1 are data items at site 1, and X_2, Y_2 are at site 2.

T_1	T_2	T_{G1}	T_{G2}
write(Y_1)			
		read(Y_1)	
		write(X_2)	
	read(X_2)		
	write(Y_2)		
			read(Y_2)
			write(X_1)
read(X_1)			

In this schedule, T_{G2} starts only after T_{G1} finishes. Within each site, there is local serializability. In site 1, $T_{G2} \rightarrow T_1 \rightarrow T_{G1}$ is a serializability order. In site 2, $T_{G1} \rightarrow T_2 \rightarrow T_{G2}$ is a serializability order. Yet the global schedule schedule is non-serializable.

25.8 **a.** The same system as in the answer to Exercise 25.7 is assumed, except that now both the global transactions are read-only. Consider the schedule given below.

T_1	T_2	T_{G1}	T_{G2}
			read(X_1)
write(X_1)			
		read(X_1)	
		read(X_2)	
	write(X_2)		
			read(X_2)

Though there is local serializability in both sites, the global schedule is not serializable.

b. Since local serializability is guaranteed, any cycle in the system wide precedence graph must involve at least two different sites, and two different global transactions. The ticket scheme ensures that whenever two global transactions access data at a site, they conflict on a data item (the ticket) at that site. The global transaction manager controls ticket access in such a manner that the global transactions execute with the same serializability order in all the sites. Thus the chance of their participating in a cycle in the system wide precedence graph is eliminated.