1. Our first step is to reconstruct the state of the database at the time of the crash, including blocks whose current values were in buffers and therefore got lost. To do so:

   (a) Find the most recent checkpoint on the log, and determine from it the set of transactions that were active at that time.

   (b) For each log entry $<L, T, A, B>$, compare the log sequence number N on block B with the log sequence number L for this log record. If $N < L$, then redo action A; that action was never performed on block B. However, if $N \geq L$, then do nothing; the effect of A was already felt by B.

   (c) For each log entry that informs us that a transaction T started, committed, or aborted, adjust the set of active transactions accordingly.

2. The set of transactions that remain active when we reach the end of the log must be aborted. To do so:

   (a) Scan the log again, this time from the end back to the previous checkpoint. Each time we encounter a record $<L, T, A, B>$ for a transaction T that must be aborted, perform the compensating action for A on block B and record in the log the fact that that compensating action was performed.

   (b) If we must abort a transaction that began prior to the most recent checkpoint (i.e., that transaction was on the active list for the checkpoint), then continue back in the log until the start-records for all such transactions have been found.

   (c) Write abort-records in the log for each of the transactions we had to abort.

19.1.9 Exercises for Section 19.1

Exercise 19.1.1: What are all the ways to insert locks (of a single type only, as in Section 18.3) into the sequence of actions

$$r_1(A);\ r_1(B);\ w_1(A);\ w_1(B);$$

so that the transaction T_1 is:

a) Two-phase locked, but not strict.

b) Two-phase locked, and strict.

Exercise 19.1.2: Suppose that each of the sequences of actions below is followed by an abort action for transaction T_1. Tell which transactions need to be rolled back.

a) $r_1(A);\ w_1(B);\ r_3(B);\ w_3(C);\ r_2(C);\ w_2(D);$

b) $r_3(A)$; $r_2(A)$; $r_1(A)$; $w_1(B)$; $r_3(B)$; $r_2(B)$; $w_3(C)$; $r_2(C)$;

c) $r_3(A)$; $r_2(A)$; $r_1(A)$; $w_1(B)$; $r_2(B)$; $w_3(C)$; $r_2(C)$;

d) $r_1(A)$; $r_3(B)$; $w_1(B)$; $w_3(C)$; $r_2(B)$; $r_2(C)$; $w_2(D)$;

Exercise 19.1.3: Give an example of an ACR schedule with shared and exclusive locks that is not strict.

Exercise 19.1.4: Consider each of the sequences of actions in Exercise 19.1.2, but now suppose that all three transactions commit and write their commit record on the log immediately after their last action. However, a crash occurs, and a tail of the log was not written to disk before the crash and is therefore lost. Tell, depending on where the lost tail of the log begins:

i. What transactions could be considered uncommitted?

ii. Are any dirty reads created during the recovery process? If so, what transactions need to be rolled back?

iii. What additional dirty reads could have been created if the portion of the log lost was not a tail, but rather some portions in the middle?

! **Exercise 19.1.5:** Consider the following two transactions:

$$T_1\colon\ w_1(A);\ w_1(B);\ r_1(C);\ c_1;$$
$$T_2\colon\ w_2(A);\ r_2(B);\ w_2(C);\ c_2;$$

a) How many schedules of T_1 and T_2 are recoverable?

b) Of these, how many are ACR schedules?

c) How many are both ACR and serializable?

d) How many are both recoverable and serializable?

19.2 Deadlocks

Several times we have observed that concurrently executing transactions can compete for resources and thereby reach a state where there is a *deadlock*: each of several transactions is waiting for a resource held by one of the others, and none can make progress.

- In Section 18.3.4 we saw how ordinary operation of two-phase-locked transactions can still lead to a deadlock, because each has locked something that another transaction also needs to lock.

- In Section 18.4.3 we saw how the ability to upgrade locks from shared to exclusive can cause a deadlock because each transaction holds a shared lock on the same element and wants to upgrade the lock.

Why Timestamp-Based Deadlock Detection Works

We claim that in either the wait-die or wound-wait scheme, there can be no cycle in the waits-for graph, and hence no deadlock. Suppose there is a cycle such as $T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow T_1$. One of the transactions is the oldest, say T_2.

In the wait-die scheme, you can only wait for younger transactions. Thus, it is not possible that T_1 is waiting for T_2, since T_2 is surely older than T_1. In the wound-wait scheme, you can only wait for older transactions. Thus, there is no way T_2 could be waiting for the younger T_3. We conclude that the cycle cannot exist, and therefore there is no deadlock.

On the other hand, when a rollback does occur, wait-die rolls back a transaction that is still in the stage of gathering locks, presumably the earliest phase of the transaction. Thus, although wait-die may roll back more transactions than wound-wait, these transactions tend to have done little work. In contrast, when wound-wait does roll back a transaction, it is likely to have acquired its locks and for substantial processor time to have been invested in its activity. Thus, either scheme may turn out to cause more wasted work, depending on the population of transactions processed.

We should also consider the advantages and disadvantages of both wound-wait and wait-die when compared with a straightforward construction and use of the waits-for graph. The important points are:

- Both wound-wait and wait-die are easier to implement than a system that maintains or periodically constructs the waits-for graph.

- Using the waits-for graph minimizes the number of times we must abort a transaction because of deadlock. If we abort a transaction, there really is a deadlock. On the other hand, either wound-wait or wait-die will sometimes roll back a transaction when there really is no deadlock.

19.2.6 Exercises for Section 19.2

Exercise 19.2.1: For each of the sequences of actions below, assume that shared locks are requested immediately before each read action, and exclusive locks are requested immediately before every write action. Also, unlocks occur immediately after the final action that a transaction executes. Tell what actions are denied, and whether deadlock occurs. Also tell how the waits-for graph evolves during the execution of the actions. If there are deadlocks, pick a transaction to abort, and show how the sequence of actions continues.

a) $r_1(A)$; $r_3(B)$; $r_2(C)$; $w_1(B)$; $w_3(C)$; $w_2(D)$;

b) $r_1(A)$; $r_3(B)$; r

c) $r_1(A)$; $r_3(B)$;

d) $r_1(A)$; $r_3(B)$;

Exercise 19.2.2: F
happens under the
deadlock-timestamp
that is, T_1, T_2, T_3, T_4
in the order that th

Exercise 19.2.3:
what happens unde
same assumptions a

!! **Exercise 19.2.4:** (
action to announce
all those locks or de
proach avoid deadlo
of a deadlock that c

! **Exercise 19.2.5:**
scribe how to constr
cially, consider the
transactions in mod
has to wait, what a

! **Exercise 19.2.6:**
methods other than
vation, where a tra
Give an example of
would cause a cycl
request locks on ele
about timeouts as a

! **Exercise 19.2.7:**
no smaller cycle, fo
node?

19.3 Long-

There is a family o
maintaining data, b
concurrency-contro
tion we shall exam
that arise. We the
that negate the effe
been.

b) $r_1(A)$; $r_3(B)$; $r_2(C)$; $w_1(B)$; $w_3(C)$; $w_2(A)$;

c) $r_1(A)$; $r_3(B)$; $w_1(C)$; $w_3(D)$; $r_2(C)$; $w_1(B)$; $w_4(D)$; $w_3(A)$;

d) $r_1(A)$; $r_3(B)$; $w_1(C)$; $r_2(D)$; $r_4(E)$; $w_2(B)$; $w_3(C)$; $w_4(A)$; $w_1(D)$;

Exercise 19.2.2: For each of the action sequences in Exercise 19.2.1, tell what happens under the wait-die deadlock avoidance system. Assume the order of deadlock-timestamps is the same as the order of subscripts for the transactions, that is, T_1, T_2, T_3, T_4. Also assume that transactions that need to restart do so in the order that they were rolled back.

Exercise 19.2.3: For each of the action sequences in Exercise 19.2.1, tell what happens under the wound-wait deadlock avoidance system. Make the same assumptions as in Exercise 19.2.2.

!! **Exercise 19.2.4:** One approach to avoiding deadlocks is to require each transaction to announce all the locks it wants at the beginning, and to either grant all those locks or deny them all and make the transaction wait. Does this approach avoid deadlocks due to locking? Either explain why, or give an example of a deadlock that can arise.

! **Exercise 19.2.5:** Consider the intention-locking system of Section 18.6. Describe how to construct the waits-for graph for this system of lock modes. Especially, consider the possibility that a database element A is locked by different transactions in modes IS and also either S or IX. If a request for a lock on A has to wait, what arcs do we draw?

! **Exercise 19.2.6:** In Section 19.2.5 we pointed out that deadlock-detection methods other than wound-wait and wait-die do not necessarily prevent starvation, where a transaction is repeatedly rolled back and never gets to finish. Give an example of how using the policy of rolling back any transaction that would cause a cycle can lead to starvation. Does requiring that transactions request locks on elements in a fixed order necessarily prevent starvation? What about timeouts as a deadlock-resolution mechanism?

! **Exercise 19.2.7:** Can one have a waits-for graph with a cycle of length n, but no smaller cycle, for any integer $n > 1$? What about $n = 1$, i.e., a loop on a node?

19.3 Long-Duration Transactions

There is a family of applications for which a database system is suitable for maintaining data, but the model of many short transactions on which database concurrency-control mechanisms are predicated, is inappropriate. In this section we shall examine some examples of these applications and the problems that arise. We then discuss a solution based on "compensating transactions" that negate the effects of transactions that were committed, but shouldn't have been.

BASIS: If $n = 1$, then the sequence of all actions between A_1 and its compensating transaction A_1^{-1} looks like $A_1\alpha A_1^{-1}$. By the fundamental assumption about compensating transactions, $A_1\alpha A_1^{-1} \equiv \alpha$.

INDUCTION: Assume the statement for paths of up to $n-1$ actions, and consider a path of n actions, followed by its compensating transactions in reverse order, with any other transactions intervening. The sequence looks like

$$A_1\alpha_1A_2\alpha_2\cdots\alpha_{n-1}A_n\beta A_n^{-1}\gamma_{n-1}\cdots\gamma_2A_2^{-1}\gamma_1A_1^{-1} \qquad (19.1)$$

where all Greek letters represent sequences of zero or more actions. By the definition of compensating transaction, $A_n\beta A_n^{-1} \equiv \beta$. Thus, (19.1) is equivalent to

$$A_1\alpha_1A_2\alpha_2\cdots A_{n-1}\alpha_{n-1}\beta\gamma_{n-1}A_{n-1}^{-1}\gamma_{n-2}\cdots\gamma_2A_2^{-1}\gamma_1A_1^{-1} \qquad (19.2)$$

By the inductive hypothesis, expression (19.2) is equivalent to

$$\alpha_1\alpha_2\cdots\alpha_{n-1}\beta\gamma_{n-1}\cdots\gamma_2\gamma_1$$

since there are only $n-1$ actions in (19.2). That is, the saga and its compensation leave the database state the same as if the saga had never occurred.

19.3.5 Exercises for Section 19.3

! **Exercise 19.3.1:** The process of "uninstalling" software can be thought of as a compensating transaction for the action of installing the same software. In a simple model of installing and uninstalling, suppose that an action consists of *loading* one or more files from the source (e.g., a CD-ROM) onto the hard disk of the machine. To load a file f, we copy f from CD-ROM. If there was a file f' with the same path name, we back up f' before replacement. To distinguish files with the same path name, we may assume each file has a timestamp.

a) What is the compensating transaction for the action that loads file f? Consider both the case where no file with that path name existed, and where there was a file f' with the same path name.

b) Explain why your answer to (a) is guaranteed to compensate. *Hint*: Consider carefully the case where after replacing f' by f, a later action replaces f by another file with the same path name.

! **Exercise 19.3.2:** Describe the process of booking an airline seat as a saga. Consider the possibility that the customer will query about a seat but not book it. The customer may book the seat, but cancel it, or not pay for the seat within the required time limit. The customer may or may not show up for the flight. For each action, describe the corresponding compensating transaction.