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

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

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

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

! **Exercise 19.2.4:** 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?

!! **Exercise 19.2.5:** 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 19.2.6:** 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 19.2.7:** 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?

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.

19.3.1 Problems of Long Transactions

Roughly, a *long transaction* is one that takes too long to be allowed to hold locks that another transaction needs. Depending on the environment, "too long" could mean seconds, minutes, or hours. Three broad classes of applications that involve long transactions are:

1. *Conventional DBMS Applications.* While common database applications run mostly short transactions, many applications require occasional long transactions. For example, one transaction might examine all of a bank's accounts to verify that the total balance is correct. Another application may require that an index be reconstructed occasionally to keep performance at its peak.

2. *Design Systems.* Whether the thing being designed is mechanical like an automobile, electronic like a microprocessor, or a software system, the common element of design systems is that the design is broken into a set of components (e.g., files of a software project), and different designers work on different components simultaneously. We do not want two designers taking a copy of a file, editing it to make design changes, and then writing the new file versions back, because then one set of changes would overwrite the other. Thus, a *check-out-check-in* system allows a designer to "check out" a file and check it in when the changes are finished, perhaps hours or days later. Even if the first designer is changing the file, another designer might want to look at the file to learn something about its contents. If the check-out operation were tantamount to an exclusive lock, then some reasonable and sensible actions would be delayed, possibly for days.

3. *Workflow Systems.* These systems involve collections of processes, some executed by software alone, some involving human interaction, and perhaps some involving human action alone. We shall give shortly an example of office paperwork involving the payment of a bill. Such applications may take days to perform, and during that entire time, some database elements may be subject to change. Were the system to grant an exclusive lock on data involved in a transaction, other transactions could be locked out for days.

Example 19.13: Consider the problem of an employee vouchering travel expenses. The intent of the traveler is to be reimbursed from account A123, and the process whereby the payment is made is shown in Fig. 19.11. The process begins with action A_1, where the traveler's secretary fills out an on-line form describing the travel, the account to be charged, and the amount. We assume in this example that the account is A123, and the amount is \$1000.

The traveler's receipts are sent physically to the departmental authorization office, while the form is sent on-line to an automated action A_2. This process checks that there is enough money in the charged account (A123) and reserves the money for expenditure; i.e., it tentatively deducts \$1000 from the account

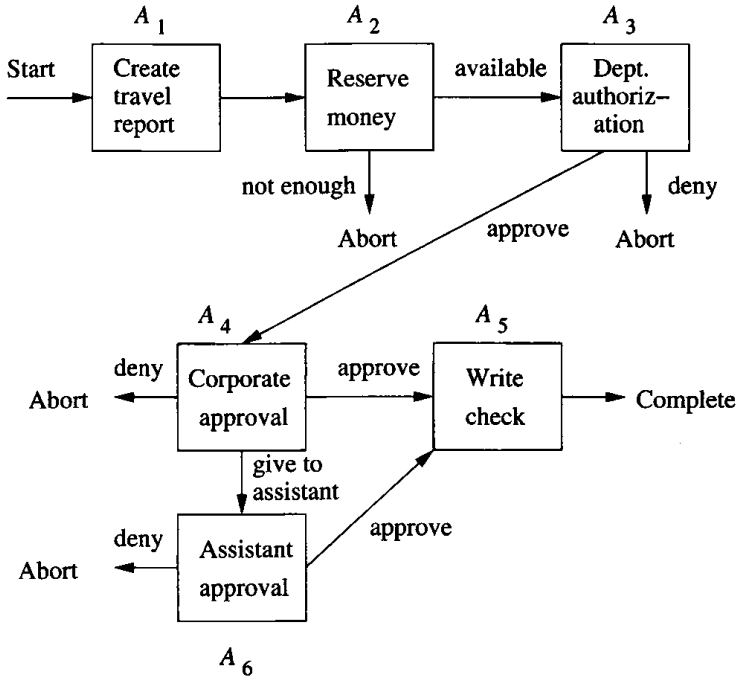

Figure 19.11: Workflow for a traveler requesting expense reimbursement

but does not issue a check for that amount. If there is not enough money in the account, the transaction aborts, and presumably it will restart when either enough money is in the account or after changing the account to be charged.

Action A_3 is performed by the departmental administrator, who examines the receipts and the on-line form. This action might take place the next day. If everything is in order, the form is approved and sent to the corporate administrator, along with the physical receipts. If not, the transaction is aborted. Presumably the traveler will be required to modify the request in some way and resubmit the form.

In action A_4, which may take place several days later, the corporate administrator either approves or denies the request, or passes the form to an assistant, who will then make the decision in action A_5. If the form is denied, the transaction again aborts and the form must be resubmitted. If the form is approved, then at action A_6 the check is written, and the deduction of \$1000 from account A123 is finalized.

However, suppose that the only way we could implement this workflow is by conventional locking. In particular, since the balance of account A123 may be changed by the complete transaction, it has to be locked exclusively at action A_2 and not unlocked until either the transaction aborts or action A_6 completes. This lock may have to be held for days, while the people charged with authorizing the payment get a chance to look at the matter. If so, then there can be no other charges made to account A123, even tentatively. On

the other hand, if there are no controls at all over how account A123 can be accessed, then it is possible that several transactions will reserve or deduct money from the account simultaneously, leading to an overdraft. Thus, some compromise between rigid, long-term locks on one hand, and anarchy on the other, is required. □

19.3.2 Sagas

A *saga* is a collection of actions, such as those of Example 19.13, that together form a long-duration "transaction." That is, a saga consists of:

1. A collection of actions.
2. A directed graph whose nodes are either actions or the *terminal* nodes *Abort* and *Complete*. No arcs leave the terminal nodes.
3. An indication of the node at which the action starts, called the *start node*.

The paths through the graph, from the start node to either of the terminal nodes, represent possible sequences of actions. Those paths that lead to the *Abort* node represent sequences of actions that cause the overall transaction to be rolled back, and these sequences of actions should leave the database unchanged. Paths to the *Complete* node represent successful sequences of actions, and all the changes to the database system that these actions perform will remain in the database.

Example 19.14: The paths in the graph of Fig. 19.11 that lead to the *Abort* node are: A_1A_2, $A_1A_2A_3$, $A_1A_2A_3A_4$, and $A_1A_2A_3A_4A_5$. The paths that lead to the *Complete* node are $A_1A_2A_3A_4A_6$, and $A_1A_2A_3A_4A_5A_6$. Notice that in this case the graph has no cycles, so there are a finite number of paths leading to a terminal node. However, in general, a graph can have cycles and an infinite number of paths. □

Concurrency control for sagas is managed by two facilities:

1. Each action may be considered itself a (short) transaction, that when executed uses a conventional concurrency-control mechanism, such as locking. For instance, A_2 may be implemented to (briefly) obtain a lock on account A123, decrement the amount indicated on the travel voucher, and release the lock. This locking prevents two transactions from trying to write new values of the account balance at the same time, thereby losing the effect of the first to write and making money "appear by magic."
2. The overall transaction, which can be any of the paths to a terminal node, is managed through the mechanism of "compensating transactions," which are inverses to the transactions at the nodes of the saga. Their job is to roll back the effect of a committed action in a way that does not depend on what has happened to the database between the time the action was executed and the time the compensating transaction is executed.

When are Database States "The Same"?

When discussing compensating transactions, we should be careful about what it means to return the database to "the same" state that it had before. We had a taste of the problem when we discussed logical logging for B-trees in Example 19.8. There we saw that if we "undid" an operation, the state of the B-tree might not be identical to the state before the operation, but would be equivalent to it as far as access operations on the B-tree were concerned. More generally, executing an action and its compensating transaction might not restore the database to a state literally identical to what existed before, but the differences must not be detectable by whatever application programs the database supports.

19.3.3 Compensating Transactions

In a saga, each action A has a *compensating transaction*, which we denote A^{-1}. Intuitively, if we execute A, and later execute A^{-1}, then the resulting database state is the same as if neither A nor A^{-1} had executed. More formally:

- If D is any database state, and $B_1B_2\cdots B_n$ is any sequence of actions and compensating transactions (whether from the saga in question or any other saga or transaction that may legally execute on the database) then the same database states result from running the sequences $B_1B_2\cdots B_n$ and $AB_1B_2\cdots B_nA^{-1}$ starting in database state D.

If a saga execution leads to the *Abort* node, then we roll back the saga by executing the compensating transactions for each executed action, in the reverse order of those actions. By the property of compensating transactions stated above, the effect of the saga is negated, and the database state is the same as if it had never happened. An explanation of why the effect is guaranteed to be negated is given in Section 19.3.4

Example 19.15: Let us consider the actions in Fig. 19.11 and see what the compensating transactions for A_1 through A_6 might be. First, A_1 creates an online document. If the document is stored in the database, then A_1^{-1} must remove it from the database. Notice that this compensation obeys the fundamental property for compensating transactions: If we create the document, do any sequence of actions α (including deletion of the document if we wish), then the effect of $A_1\alpha A_1^{-1}$ is the same as the effect of α.

A_2 must be implemented carefully. We "reserve" the money by deducting it from the account. The money will stay removed unless restored by the compensating transaction A_2^{-1}. We claim that this A_2^{-1} is a correct compensating transaction if the usual rules for how accounts may be managed are followed. To appreciate the point, it is useful to consider a similar transaction where the

obvious compensation will not work; we consider such a case in Example 19.16, next.

The actions A_3, A_4, and A_6 each involve adding an approval to a form. Thus, their compensating transactions can remove that approval.[3]

Finally, A_5, which writes the check, does not have an obvious compensating transaction. In practice none is needed, because once A_5 is executed, this saga cannot be rolled back. However, technically A_5 does not affect the database anyway, since the money for the check was deducted by A_2. Should we need to consider the "database" as the larger world, where effects such as cashing a check affected the database, then we would have to design A_5^{-1} to first try to cancel the check, next write a letter to the payee demanding the money back, and if all remedies failed, restoring the money to the account by declaring a loss due to a bad debt. □

Next, let us take up the example, alluded to in Example 19.15, where a change to an account cannot be compensated by an inverse change. The problem is that accounts normally are not allowed to go negative.

Example 19.16: Suppose B is a transaction that adds \$1000 to an account that has \$2000 in it initially, and B^{-1} is the compensating transaction that removes the same amount of money. Also, it is reasonable to assume that transactions may fail if they try to delete money from an account and the balance would thereby become negative. Let C be a transaction that deletes \$2500 from the same account. Then $BCB^{-1} \not\equiv C$. The reason is that C by itself fails, and leaves the account with \$2000, while if we execute B then C, the account is left with \$500, whereupon B^{-1} fails.

Our conclusion that a saga with arbitrary transfers among accounts and a rule about accounts never being allowed to go negative cannot be supported simply by compensating transactions. Some modification to the system must be done, e.g., allowing negative balances in accounts. □

19.3.4 Why Compensating Transactions Work

Let us say that two sequences of actions are *equivalent* ($\equiv$) if they take any database state D to the same state. The fundamental assumption about compensating transactions can be stated:

- If A is any action and α is any sequence of legal actions and compensating transactions, then $A\alpha A^{-1} \equiv \alpha$.

Now, we need to show that if a saga execution $A_1A_2\cdots A_n$ is followed by its compensating transactions in reverse order, $A_n^{-1}\cdots A_2^{-1}A_1^{-1}$, with any intervening actions whatsoever, then the effect is as if neither the actions nor the compensating transactions executed. The proof is an induction on n.

[3]In the saga of Fig. 19.11, the only time these actions are compensated is when we are going to delete the form anyway, but the definition of compensating transactions require that they work in isolation, regardless of whether some other compensating transaction was going to make their changes irrelevant.

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

$$A_1\alpha_1 A_2\alpha_2 \cdots \alpha_{n-1}A_n\beta A_n^{-1}\gamma_{n-1}\cdots\gamma_2 A_2^{-1}\gamma_1 A_1^{-1} \tag{19.1}$$

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

$$A_1\alpha_1 A_2\alpha_2 \cdots A_{n-1}\alpha_{n-1}\beta\gamma_{n-1}A_{n-1}^{-1}\gamma_{n-2}\cdots\gamma_2 A_2^{-1}\gamma_1 A_1^{-1} \tag{19.2}$$

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.