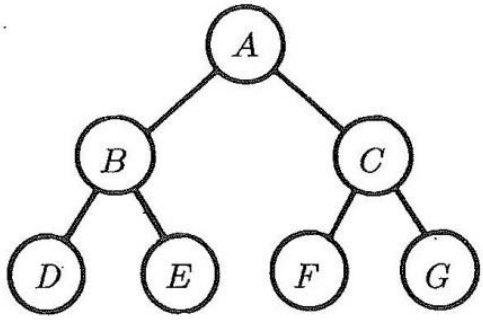

Fig. 11.17. A hierarchy.

Example 11.12: Figure 11.17 shows a hierarchy, and Fig. 11.18 is a schedule of three transactions obeying the warning protocol. Notice, for example that at step (4) T_1 places a warning on B. Therefore T_3 was not able to lock B until T_1 unlocked its warning on B at step (10). However, at steps (1)–(3) all three transactions place warnings on A, which is legal.

The lock of C by T_2 at step (5) implicitly locks C, F, and G. We assume that any or all of these items are changed by T_2 before the lock is removed at step (7). □

Theorem 11.6: Schedules of transactions obeying the warning protocol are serializable.

Proof: Parts (1)–(3) of the warning protocol guarantee that no transaction can place a lock on an item unless it holds warnings on all its ancestors. It follows that at no time can two transactions hold locks on two ancestors of the same item. We can now show that a schedule obeying the warning protocol is equivalent to a schedule using the model of Section 11.2, in which all items are locked explicitly (not implicitly, by locking an ancestor). Given a schedule S satisfying the warning protocol, construct a schedule S' in the model of Section 11.2 as follows.

1. Remove all warning steps, and their matching unlock steps.
2. Replace all locks by locks on the item and all its descendants. Do the same for the corresponding unlocks.

The resulting schedule S' is legal because of parts (1)–(3) of the warning protocol, and its transactions are two-phase because those of S are two-phase, by part (4) of the warning protocol. □

11.6 PROTECTING AGAINST CRASHES

Until now we have assumed that each transaction runs happily to completion. In practice, several things might happen to prevent a transaction from completing.

1. The system could fail from a variety of hardware or software causes. In this case, all active transactions are prevented from completing, and it is even possible that some completed transactions must be "cancelled," because they read values written by transactions that have not yet completed.

	T_1	T_2	T_3
(1)	WARN A		
(2)		WARN A	
(3)			WARN A
(4)	WARN B		
(5)		LOCK C	
(6)	LOCK D		
(7)		UNLOCK C	
(8)	UNLOCK D		
(9)		UNLOCK A	
(10)	UNLOCK B		
(11)			LOCK B
(12)			WARN C
(13)			LOCK F
(14)	UNLOCK A		
(15)			UNLOCK B
(16)			UNLOCK F
(17)			UNLOCK C
(18)			UNLOCK A

Fig. 11.18. A schedule of transactions satisfying the warning protocol.

System crashes cause serious problems, since we must not only find a set of transactions to "cancel" that will bring us back to a consistent state, but we must make sure that some way of reconstructing that state exists.

2. A single transaction could be forced to stop before completion for a variety of reasons. If deadlock detection is done by the system, the transaction could be found to contribute to a deadlock and be selected for cancellation by the system. A bug in the transaction, e.g., a division by zero, could cause an interrupt and cancellation of the transaction. Similarly, the user could cause an interrupt at his terminal for the express purpose of cancelling a transaction.
3. In the next section, we shall discuss "optimistic" concurrency control, where transactions are allowed to run without locking items, and violations of the serializability condition are detected and the offending transaction cancelled.

Backup Copies

It should be evident that we cannot rely on the indefinite preservation of the data in a database. Data in the machine's registers or solid state memory cannot be presumed to survive a power outage, for example. Magnetic devices

such as tapes, disks, or magnetic core memory will usually be preserved even if the machine has to be shut down, but even this data is vulnerable to physical problems such as disk "head crashes" or a small child with a large magnet running amok in the computer room. Further, no data is completely safe from being obliterated by system software errors.

For these reasons, it is essential that backup copies of the database be made periodically, at least once a day if possible, although enormous databases, for which the copying process could take hours, must be copied less frequently. The copy, once made on tape on disk, should be removed from the vicinity of the computer (in case of fire, for example), and stored in a safe place. For extra security, several of the most recent copies could be stored in different places.

When making a copy, it is important that the copied data represent a consistent state. Therefore, the copying utility routine must itself be a transaction that read-locks all items in the database.

The Journal

We must also be prepared to restore the database to a consistent state that reflects the situation after some number, perhaps a large number, of transactions were completed following the creation of the last backup copy.† For this reason, we need to preserve in a relatively safe place, e.g., on a tape or disk, a history, called a *journal* or *log,* of all changes to the database since the last backup copy was made. In the most general case, journal entries consist of

1. A unique identifier for the transaction causing the change,
2. The old value of the item, and
3. The new value of the item.

We also expect the journal to record key times in the progress of a transaction, such as its beginning, end, and what we shall later call its "commit point."

The need for old and new values will become evident when we consider that it may not only be necessary to redo transactions, but to undo them, that is to erase completely the effect of certain transactions. If items are large, if they are relations for example, then it is wise to represent the changes only, rather than listing the complete old and new values. For example, we could list the inserted and deleted tuples and give the old and new values for the modified tuples.

Committed Transactions

When dealing with transactions that may have to be redone or undone, it helps to think in terms of "committed" and "uncommitted" transactions. There

† This consistent state may never have existed during the history of the database, since other uncompleted, or even completed, transactions may have been running concurrently with the selected set of transactions on which the state is based.

is a point during the execution of any transaction at which we regard it as completed. All calculations done by the transaction in its workspace must have been finished, and a copy of the results of the transaction must have been written in a secure place, presumably in the journal. At this time we may regard the transaction as *committed;* if a system crash occurs subsequently, its effects will survive the crash, even though the values produced by the transaction may not yet have appeared in the database itself. The action of committing the transaction must itself be written in the journal, so if we have to recover from a crash by examining the journal, we know which transactions are committed. We define the *two-phase commit*† policy as follows.

1. A transaction cannot write into the database until it has committed.
2. A transaction cannot commit until it has recorded all its changes to items in the journal.

Note that phase one is the writing of data in the journal and phase two is writing the same data in the database.

If in addition, transactions follow the two-phase locking protocol, and unlocking occurs after commitment, then we know that no transaction can read from the database a value written by an uncommitted transaction. In the case of a system crash, it is then possible to examine the log and redo all the committed transactions that did not have a chance to write their values in the database. If the crash is of a nature that destroys data in the database, we shall have to redo all committed transactions since the last backup copy was made, which is generally far more time consuming. It is not necessary to undo any transactions that did not reach their commit point before the system crash, since these have no effect on the database. It would be a good idea to print a message to the user warning him that his transaction did not complete. To be able to do so after a crash, it is necessary routinely to enter into the journal the fact that a particular transaction has begun. Also note that a crash may cause locks to be left on items, either from committed or uncommitted transactions, and these must be removed by the recovery routine.

Failure of Individual Transactions

Less serious than a system crash is the failure of a single transaction, when, for example, it causes deadlock or is interrupted for some reason. If we follow the two-phase commit policy, we know that there is no effect on the database, provided that no interruption of a transaction can occur after commitment. If we are following the two-phase protocol, then no locks or calculation can occur after commitment, so it is not possible that a deadlock is created after commitment, or that an arithmetic error causes an interrupt after commitment. Thus, failed transactions leave no trace on the database. A notation indicating

† There is no connection between the "two-phase protocol" and "two-phase commitment," except that they are both sensible ideas.

that a transaction was cancelled should be placed in the journal, so if restart occurs after a system crash, we know to ignore any journal entries for that transaction.

Transactions That do not Obey the Two-Phase Commit Policy

Let us briefly consider what happens if transactions are not required to reach their commit point before writing into the database. By relaxing this requirement, we may allow transactions to unlock items earlier, and thereby allow other transactions to execute concurrently instead of waiting. However, this potential increase in concurrency is paid for by making recovery after a crash more difficult, as we shall see.

We still assume that each item has its changes entered in the journal before the database itself is actually changed, and we assume transactions obey the two-phase protocol with regard to locking. We also assume a transaction does not commit until it has completed writing into the journal whatever items it changes. Under our new assumptions it is not impossible to recover from system crashes or failures of individual transactions, but it becomes more difficult for two reasons.

1. A transaction that is uncommitted when the crash occurs must have the changes that it made to the database undone.
2. A transaction that has read a value written by some transaction that must be undone, must itself be undone. This effect can propagate indefinitely.

Example 11.13: Consider the two transactions of Fig. 11.19. Fundamentally these transactions follow the model of Section 11.2, although to make clear certain details of timing, we have explicitly shown commitment, reads, writes, and the arithmetic done in the workspace of each transaction. The WRITE steps are presumed to write the old and new values in the journal and then in the database. Suppose that after step (14) there is a crash. Since T_1 is the only active transaction, it doesn't matter whether it was a system crash or a failure of T_1, say because division by 0 occurred at step (14).

We must undo T_1 because it is uncommitted. Since it holds a lock on B, that lock must be removed. Then we must restore to A its value prior to step (1). We must also undo T_2, even though it is committed, and in fact completed. If some other transaction T_3 had read A between steps (13) and (14), then T_3 would have to be redone as well, even if T_3 were completed, and so on.

To undo transactions that must be undone, we consider each item C written by one or more of the transactions that must be undone. Examine the journal for the earliest write of C by one of the undone transactions. This journal entry will have the old value of C, which can be placed in the database. Note that since we assume all transactions are two-phase, and we are using the model of Section 11.2, where all items locked are assumed to be read as well as written,

	T_1	T_2
(1)	LOCK A	
(2)	READ A	
(3)	$A := A - 1$	
(4)	WRITE A	
(5)	LOCK B	
(6)	UNLOCK A	
(7)		LOCK A
(8)		READ A
(9)		$A := A * 2$
(10)	READ B	
(11)		WRITE A
(12)		COMMIT
(13)		UNLOCK A
(14)	$B := B/A$	

Fig. 11.19. A schedule.

it is not possible that some transaction T, which does not have to be redone, wrote a value for C later than the earliest undone transaction wrote C. We leave as an exercise the correct algorithm for modifying the database to reflect the undoing of transactions when the model of Section 11.4, which permits writing without reading, is used.

In the case of our example in Fig. 11.19, only A was written by the transactions T_1 and T_2 prior to the crash. We find that the earliest write of A by either of these transactions was by T_1 at step (4). The journal entry for step (4) will include the old value of A, the value read at step (2). Replacing A by that value cancels all effects of T_1 and T_2 on the database. □

One might assume that having undone T_1 and T_2 in Example 11.13, it is now possible to redo T_2, since it was committed, simply by examining the journal, rather than by running it again. Such is not the case, since T_2 read the value of A written into the database by T_1, and that value is no longer there. To retrieve that value of A from the journal, without rerunning T_1, might lead to an inconsistency in the database.

11.7 OPTIMISTIC CONCURRENCY CONTROL

Let us briefly consider a method of synchronizing transactions that is radically different from the locking methods discussed in the previous sections. Suppose that we were "optimistic," and executed transactions with no locking at all, reading and writing into the database as we wished. Naturally, if we were executing two transactions such as those of Fig. 11.1, with steps executed in the

order shown in that figure, we would have a nonserializable behavior. However, suppose that when we read or wrote into the database, we at least had some way of realizing that we were doing something that might violate serial order, for example, reading a value that had been written by another transaction that belonged after our transaction in the serial order. Then we could abort our transaction and start it over.

This approach is quite efficient, provided that the probability of two transactions conflicting in this way is small. The chance of interaction between two transactions executing at about the same time will be small provided each accesses only a tiny fraction of the database, i.e., the lock granularity is small. If this situation holds, there is substantial benefit in using the "optimistic" approach, as no time will be wasted requesting or waiting for locks. The time wasted redoing transactions that have to be aborted may well be much less than the time spent waiting for locks or undoing deadlocks.

Yet in order to adopt this method, a number of questions must be answered. How do we tell that we are violating serial order? How do we avoid leaving in the database values written by transactions that later abort? When we must abort a transaction, how do we avoid livelock, the situation where the transaction is restarted over and over, but aborts each time? We shall answer these questions in turn.

Timestamps

The key to establishing a serial order and detecting its violations is to have the system generate *timestamps*, which are numbers generated at each clock "tick," where ticks of the computer's internal clock occur with sufficient frequency, say once every microsecond, that two events such as the start of transactions cannot occur at the same "tick." At a special time in the life of every transaction, say when it initiates, or when it issues its first read or write on the database, the transaction is issued its timestamp, which is the current time on the clock. No two transactions can have the same timestamp, and we shall take as our criterion of correctness that transactions should behave as if the order of their timestamps were their serial order as well.

As an aside, one might think that adding one to the clock every microsecond would produce enormous numbers as timestamps. However, the fact is that the number of ticks in a century fits in a 32-bit word, and even 24 bits holds numbers large enough that they would have to repeat only every half year. If we make the reasonable assumption that half a year is long enough for any transaction to complete, no confusion will result if we let timestamps recycle after that amount of time. Even a sixteen bit timestamp is adequate for most purposes.

Now, we must consider how timestamps are used to force those transactions that do not abort to behave as if they were run behave serially. We store

with each item in the database two times, the *read time*, which is the highest timestamp possessed by any transaction to have read the item, and the *write time*, which is the highest timestamp possessed by any transaction to have written the item. By so doing, we can maintain the fiction that each transaction executes instantaneously, at the time indicated by its timestamp.

We use the timestamps associated with the transactions, and the read and write times stored with the items, to check that nothing physically impossible happens. What, we may ask, is not possible?

1. It is not possible that a transaction can read the value of an item if that value was not written until after the transaction executed. That is, a transaction with a timestamp t_1 cannot read an item with a write time of t_2 if $t_2 > t_1$. If such an attempt is made, the transaction must abort and be restarted.
2. It is not possible that a transaction can write an item if that item has its old value read at a later time. That is, a transaction with timestamp t_1 cannot write an item with a read time t_2, if $t_2 > t_1$. That transaction must abort.

Notice that the other two possible conflicts do not present any problems. Not surprisingly, two transactions can read the same item at different times, without any conflict. That is, a transaction with timestamp of t_1 can read an item with a read time of t_2, even if $t_2 > t_1$. Less obviously, the transaction with timestamp t_1 need not abort if it tries to write an item with write time t_2, even if $t_2 > t_1$. We simply do not write the item. The justification is that in the serial order based on timestamps, the transaction with timestamp t_1 wrote the item, then the transaction with timestamp t_2 wrote it. However, between t_1 and t_2, apparently no transaction read the item, or else the read time of the item would exceed t_1 when the first transaction came to write, and that transaction would abort by rule (2).

To summarize, the rule for preserving serial order using timestamps, but no locking, is the following. Suppose we have a transaction with timestamp t that attempts to perform an operation X on an item with read time t_r and write time t_w.

a) Perform the operation if X=READ and $t \geq t_w$ or if X=WRITE, $t \geq t_r$, and $t \geq t_w$. In the former case, set the read time to t if $t > t_r$, and in the latter case, set the write time to t if $t > t_w$.
b) Do nothing if X=WRITE and $t_r \leq t < t_w$.
c) Abort the transaction if X=READ and $t < t_w$ or X=WRITE and $t < t_r$.

Example 11.14: Let us review the transactions of Fig. 11.1. Suppose that T_1 is given timestamp 150 and T_2 has timestamp 160. Also, assume the initial read and write times of A are both 0. Then A would be given read time 150 when T_1 reads it and 160 at the next step, when it is read by T_2. At the fifth step, when T_2 writes A, the timestamp, 160, is not less than the read time of

	T_1	T_2	T_3	A	B	C
	200	150	175	RT=0	RT=0	RT=0
				WT=0	WT=0	WT=0
(1)	READ B				RT=200	
(2)		READ A		RT=150		
(3)			READ C			RT=175
(4)	WRITE B				WT=200	
(5)	WRITE A			WT=200		
(6)		WRITE C				
(7)			WRITE A			

Fig. 11.20. Optimistically executing transactions.

A, which is also 160, nor is it less than the write time, which is 0. Thus the write is permitted, and the write time of A is set to 160. When T_1 attempts to write at the last step, its timestamp, 150, is less than the read time of A, so T_1 is aborted, preventing the anomaly illustrated in Fig. 11.1. A similar sequence of events occurs if the timestamp of T_1 is larger than that of T_2. □

Example 11.15: Figure 11.20 illustrates three transactions, with timestamps 200, 150, and 175, operating on three items, A, B, and C, each of which are assumed to have read and write times of 0 initially. The last three columns indicate changes to the read time (RT) and write time (WT) of the items.

At step (6), an attempt to write C is made. However, the transaction doing the writing, T_2, has a timestamp of 150, and the read time of C is then 175. Thus, T_2 cannot perform the write and must be aborted. Then, at step (7), T_3 tries to write A. As T_3 has a timestamp, 175, that is bigger than the read time of A, which is 150, T_3 need not abort. However, the write time of A is 200, so the value of A written by T_3 is not entered into the database, but rather is discarded, no write taking place. □

Commitment of Transactions

The above algorithm will sometimes fail to keep the database in a state that could be reached by the serial execution of transactions, because we have so far failed to account for the possibility that a transaction that aborts may have written something successfully into the database before doing so. To prevent such values written from fouling the database, we must regard the actual write operation as being only phase one of a two-phase commit.

That is, the WRITE action by a transaction does not permanently alter the database. Rather, the value is entered into the log, and the write time of the item tentatively updated, with the old value and write time stored in the log. After performing all its writes, the transaction executes a COMMIT action, whereupon all the values written by the transaction are made permanent.

Between the time the item is written and the time it is committed, no transaction may read the new value, since it is tentative, and a transaction that read it would have to be rolled back if the writer aborts, as discussed in the previous section. We must make any such reading transaction wait for the commitment of the new value or its removal.† If transactions do their writing at the end, just before the commitment, this waiting time will be short. Writes with a higher timestamp may be made, and supercede the tentative write; they are themselves tentative, however. If we keep the old value and write time available in the database, then we may let READ requests with timestamps between the old and new write times read the old value, independent of whether the new value gets committed.

Even with the commitment phase, the optimistic strategy calls for only one step for the READ operation and two for the WRITE operation (write and commit). In comparison, locking with two-phase commitment requires three or four steps per operation: LOCK, READ or WRITE, UNLOCK, COMMIT (if write). Thus, in situations where conflicts and their resulting rollback of transactions was expected not to be common, we might well prefer the optimistic approach.

Restart of Transactions

As we mentioned, the strategy with which we have been dealing does not prevent livelock, a situation where a transaction is aborted repeatedly. While we expect transactions to be aborted rarely, or the whole approach should be abandoned in favor of the locking methods described earlier, we should be aware that the potential for cyclic behavior involving only two transactions exists.

Example 11.16: Suppose we have transaction T_1 that writes B and then reads A, while T_2 writes A and then reads B.‡ If T_1 executes, say with timestamp 100, and T_2 executes with timestamp 110, we might find that T_2 wrote A before T_1 read it. In that case, T_1 would abort, because it cannot read a value with a write time greater than its own timestamp. If we immediately restart T_1, say with timestamp 120, it might write B before T_2 reads it, causing T_2 to abort and restart, say with timestamp 130. Then the second try at T_2 might write A before the second try of T_1 reads A, causing that to abort, and so on. The pattern is illustrated in Fig. 11.21. □

The solution to the problem indicated by Example 11.16 is not easy to find. Probably the simplest approach is to use a random number generator to select a random amount of time that an aborted transaction must wait before

† An alternative is to let the reading transaction proceed, but be prepared to roll it back, along with any transactions that read values it writes, as discussed in Section 11.6.

‡ These transactions may read and write other items, so writing before reading need not mean that the transactions are unrealistic.