11.5 CONCURRENCY FOR HIERARCHICALLY STRUCTURED ITEMS

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.
### 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, disk, drums, etc. may also have mechanical problems such as jams; and an entire machine may start running amok in a total panic. The machine and all its programs may be obliterated.

For these reasons, it is necessary to periodically, at a time at which the copy of the database is not needed, to make a copy from the computer (in case of a fire), and to make whatever checks are necessary to ensure security, several times per day.

When making a backup copy of the database, we must make sure that the read-locks are released while the write operation is being processed.

### The Journal

We must also be sure that the journal reflects the situation that existed before the transaction was completed if we want to be able to recover it if something goes wrong. We call a journal operation a transaction, and the order of events that made up the transaction is recorded in the journal.

1. A unique identifier for the transaction.
2. The old value of each item involved in the transaction.
3. The new value of each item involved in the transaction.
4. A brief description of the transaction.

We also expect that the journal will be stored away from the database itself so that it may not be lost in a single disaster, but rather than being lost in a single disaster, it may be expected to survive. The journal is read and written to a separate file system.

### Committed Transactions

When dealing with the database, it is important to think in terms of committed transactions. This consistent state is a single set of transactions that has been completed, or even abandoned, but the database is in a consistent state. It is possible to detect a set of transactions that is not consistent, but the transaction has not yet been committed.

### LOCK/CANCEL Transactions

The LOCK/CANCEL transactions are a way of locking items in a database. The LOCK transaction locks an item, and the CANCEL transaction releases the lock. The LOCK/CANCEL transactions are used to ensure that a transaction is committed before it is released.
11.6 PROTECTING AGAINST CRASHES

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 incompletely, 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,
(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) \( A := A \times 2 \)
(10) READ B
(11) WRITE A
(12) COMMIT
(13) UNLOCK A
(14) \( B := B / A \)

\[ T_1 \quad T_2 \]

**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. \( \square \)

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