```
1     procedure RECOVER ()
2     {                                    // Recovery procedure for a volatile, in-memory database.
3         winners ← NULL;
4         starting at end of log repeat until beginning
5         {
6             log_record ← previous record of log
7             if (type of log_record = OUTCOME)
8                 then winners ← winners + log_record;      // Set addition.
9         }
10        starting at beginning of log repeat until end
11        {
12            log_record ← next record of log
13            if (type of log_record = CHANGE)
14                and (action_id of log_record is in winners)
15                and (status of log_record of winners[action_id] = COMMITTED) then
16                perform redo_action of log_record;
17        }
18        return;
19    }
```

Figure 9–21: An idempotent redo-only recovery procedure for an in-memory database. Because RECOVER writes only to volatile storage, if a crash occurs while it is running it is safe to run it again.

The simplest recovery procedure performs two passes through the log. On the first pass, it scans the log *backward* from the last record, so the first evidence it will encounter of each all-or-nothing action is the last record that the all-or-nothing action logged. A backward log scan is sometimes called a LIFO (for last-in, first-out) log review. As the recovery procedure scans backward, it collects in a set the identity and completion status of every all-or-nothing action that logged an OUTCOME record before the crash. These actions, whether committed or aborted, are known as *winners*.

When the backward scan is complete the set of winners is also complete, so the recovery procedure begins a forward scan of the log. Since restarting after the crash reset the cell storage, on this pass the recovery procedure performs, in the order found in the log, all of the REDO actions of every winner whose OUTCOME record says that it COMMITTED, installing values in cell storage. At the end of this scan, the recovery procedure has restored cell storage to an all-or-nothing-action consistent state: its state is as though every all-or-nothing action that committed before the crash had run to completion, while every all-or-nothing action that aborted or that was still pending at crash time had never existed. The database system can now open for regular business. Figure 9–21 illustrates this recovery procedure.

This recovery procedure emphasizes the point that a log can be an authoritative version of the entire database, sufficient to completely reconstruct the reference copy in cell storage. On the other hand, unless the database system also has a durability requirement, that recovery procedure is actually overkill for our current purpose. The reason casts a spotlight on the distinction between all-or-nothing atomicity and durability. All-or-nothing atomicity is concerned with failures that occur while an atomic action is in progress. Durability is concerned with failures that occur after the action has completed. Many systems also require that the results of an atomic action survive after the action completes, but that is a durability,

rather than an atomicity, requirement. As will be seen in chapter 10, some systems have minimal durability requirements. For example, the all-or-nothing action may have been to make a group of changes to soft state in volatile storage, all of which changes must be made consistently. On the other hand, if that soft state is lost completely in the crash, it will probably have its own reconstruction procedure. In such a situation there would be no need to redo the installs of that atomic action. A useful exercise for the reader is to modify the recovery procedure to recover only those atomic actions that really require all-or-nothing atomicity; they would be the ones that logged a COMMITTED outcome but never got to the point of recording an END record, so their final results never became available.

A critical design property of the recovery procedure is that, if there should be another system crash during recovery, it must still be possible to recover. Moreover, it must be possible for any number of crash-restart cycles to occur without compromising the correctness of the ultimate result. The method is to design the recovery procedure to be *idempotent*. That is, design it so that if it is interrupted and restarted from the beginning it will produce exactly the same result as if it had run to completion to begin with. With the in-memory database configuration, this goal is not hard to obtain: just make sure that the recovery procedure modifies only volatile storage. Then, if a crash occurs during recovery, the loss of volatile storage automatically restores the state of the system to the way it was when the recovery started, and it is safe to run it again from the beginning. If the recovery procedure ever finishes, the state of the cell storage copy of the database will be correct, no matter how many interruptions and restarts intervened.

The ABORT procedure similarly needs to be idempotent, because if an all-or-nothing action decides to abort and, while running ABORT, some timer expires, the system may decide to terminate and call ABORT for that same all-or-nothing action. The version of abort in figure 9–19 will satisfy this requirement if the individual undo actions are themselves idempotent.

4. Other logging configurations: non-volatile cell storage

Placing cell storage in volatile memory is a *sweeping simplification* that works well for small and medium-sized databases, but some databases are too large for that to be practical, so the designer finds it necessary to move cell storage to some cheaper, non-volatile storage medium such as magnetic disk, as in the second configuration of figure 9–20. But with a non-volatile storage medium, installs survive system crashes, so the simple recovery procedure used with the in-memory database would have two shortcomings:

1. If, at the time of the crash, there were some pending all-or-nothing actions that had installed changes, those changes will survive the system crash. The recovery procedure must reverse the effects of those changes, just as if those actions had aborted.

2. The in-memory database recovery procedure reinstalls the entire database, even though much of it is probably intact in non-volatile storage. If the database is large enough that it requires non-volatile storage to contain it, the cost of unnecessarily reinstalling it in its entirety at every recovery is likely to be unacceptable.

In addition, reads and writes to non-volatile cell storage are likely to be slow, so it is nearly always the case that the designer installs a cache in volatile memory, along with a multilevel memory manager, thus moving to the third configuration of figure 9–20. But that addition introduces yet another shortcoming:

> *3.* In a multilevel memory system, the order in which data is written from volatile levels to non-volatile levels is generally under control of a multilevel memory manager, which may, for example, be running a least-recently-used algorithm. As a result, at the instant of the crash some things that were thought to have been installed may not yet have migrated to the non-volatile memory.

To postpone consideration of this shortcoming, let us temporarily assume that the multilevel memory manager implements a write-through cache. (Subsection C.6, below, will return to the case where the cache is not write-through.) With a write-through cache, we can be certain that everything that the application program has installed has been written to non-volatile storage. This assumption drops the third shortcoming out of our list of concerns and the situation is the same if we were using the second configuration of figure 9–20. But we still have to do something about the first two shortcomings, and we also must make sure that the modified recovery procedure is still idempotent.

To address the first shortcoming, that the database may contain installs from actions that should be undone, we need to modify the recovery procedure of figure 9–21. As the recovery procedure performs its initial backward scan, it should create a list of those all-or-nothing actions that were still in progress at the time of the crash. This set of actions are known as *losers*, and it can include both actions that committed and actions that did not. Losers are easy to identify because the first log record that contains their identity that is encountered in a backward scan will be something other than an END record. To collect the list of losers, the pseudocode keeps track of which actions logged an END record in an auxiliary list named *completeds*. Actions that are not in *completed* are the ones that go into the the list of *losers*. In addition, as it scans backwards, whenever the recovery procedure encounters a CHANGE record belonging to a loser, it performs the UNDO action listed in the record. In the course of the LIFO log review, all of the installs performed by losers will thus be rolled back and the state of the cell storage will be as if those all-or-nothing actions had never started. Next, RECOVER performs the forward log scan of the log, performing the redo actions of those losers that committed, as shown in figure 9–22. Finally, the recovery procedure logs an END record for every all-or-nothing action in the list of losers. This END record transforms the loser into a completed action, thus ensuring that future recoveries will ignore it and not perform its undos again. For future recoveries to ignore aborted losers is not just a performance enhancement, it is essential, to avoid incorrectly undoing updates to those same variables made by future all-or-nothing actions.

As before, the recovery procedure must be idempotent, so that if a crash occurs during recovery the system can just run the recovery procedure again. In addition to the technique used earlier of placing the temporary variables of the recovery procedure in volatile storage, each individual undo action must also be idempotent. For this reason, both redo and undo actions are usually expressed as *blind writes*. A blind write is a simple overwriting of a data value without reference to its previous value. Because blind writes are inherently idempotent, no matter how many times one repeats it, the result is always the same. Thus, if

```
1     procedure RECOVER ()
2     {                                          // Recovery procedure for non-volatile cell memory
3         completeds ← NULL;
4         losers ← NULL;
5         starting at end of log repeat until beginning
6         {
7             log_record ← previous record of log
8             if (type of log_record = END)
9                 then completeds ← completeds + log_record;        // Set addition.
10            if (action_id of log_record is not in completeds) then
11            {
12                losers ← losers + log_record;                     // Add if not already in set.
13                if (type of log_record = CHANGE) then
14                    perform undo_action of log_record;
15            }
16        }
17        starting at beginning of log repeat until end
18        {
19            log_record ← next record of log
20            if (type of log_record = CHANGE)
21                and (status of action_id of log_record = COMMITTED) then
22                perform redo_action of log_record;
23        }
24        for each log_record in losers do
25            log (action_id of log_record, END);                   // Show action completed.
26        return;
27    }
```

Figure 9–22: An idempotent undo/redo recovery procedure for a system that performs installs to non-volatile cell memory. In this recovery procedure, *losers* are all-or-nothing actions that were in progress at the time of the crash.

a crash occurs part way through the logging of abort records, immediately rerunning the recovery procedure will still leave the database correct. Any losers that now have abort records will be treated as completed on the rerun, but that is OK because the previous attempt of the recovery procedure has already undone their installs.

As for the second shortcoming, that the recovery procedure unnecessarily redoes every install, we can significantly simplify (and speed up) recovery by analyzing why we have to redo any installs at all. The reason is that, although the WAL protocol requires logging of changes to occur before install, there is no necessary ordering between commit and install. Until a committed action logs its END record, there is no assurance that any particular install of that action has actually happened yet. On the other hand, any committed action that has logged an END record has completed its installs. The conclusion is that the recovery procedure does not need to redo installs for any committed action that has logged its END record. A useful exercise is to modify the procedure of figure 9–22 to take advantage of that observation.

But it would be even better if the recovery procedure never had to redo *any* installs. We can arrange for that by placing another requirement on the application: it must perform all of its installs *before* it logs its OUTCOME record. That requirement, together with the write-through cache, ensures that the installs of every completed all-or-nothing action are safely in

```
1     procedure RECOVER ()
2     {                                                  // Recovery procedure for roll-back recovery.
3         completeds ← NULL;
4         losers ← NULL;
5         starting at end of log repeat until beginning  // Perform undo scan.
6         {
7             log_record ← previous record of log
8             if (type of log_record = OUTCOME)
9                 then completeds ← completeds + log_record;   // Set addition.
10            if (action_id of log_record is not in completeds) then
11            {
12                losers ← losers + log_record;                // Must be a new loser.
13                if (type of log_record = CHANGE) then
14                    perform undo_action of log_record;
15            }
16        }
17        for each log_record in losers do
18            log (action_id of log_record, OUTCOME, ABORT);   // Block future undos.
19        return;
20    }
```

Figure 9–23: An idempotent undo-only recovery procedure for a roll-back logging system.

non-volatile cell storage and there is thus never a need to perform *any* redo actions. (It also means that there is no need to log an END record.) The result is that the recovery procedure needs only to undo the installs of losers, and it can skip the entire forward scan, leading to the simpler recovery procedure of figure 9–23. This scheme, because it requires only undos, is sometimes called *undo logging* or *roll-back recovery.* A property of roll-back recovery is that for completed actions, cell storage is just as authoritative as the log. As a result, one can garbage collect the log, discarding the log records of completed actions, and can then place the—now, much smaller—log in a faster storage medium for which the durability requirement is only that it outlast pending actions.

There is an alternative, symmetric constraint used by some logging systems. Rather than requiring that all installs be done *before* logging the OUTCOME record, one can instead require that all installs be done *after* recording the OUTCOME record. With this constraint, the set of CHANGE records in the log that belong to that all-or-nothing action become a description of its intentions. If there is a crash before logging an OUTCOME record, we are assured that no installs have happened, so the recovery never needs to perform any undos. On the other hand, it may have to perform installs for all-or-nothing actions that committed. This scheme is called *redo logging* or *intentions-list recovery.* Furthermore, because we are uncertain about which installs actually have taken place, the recovery procedure must perform *all* logged installs for all-or-nothing actions that did not log an END record. Any all-or-nothing action that logged an END record must have completed all of its installs, so there is no need for the recovery procedure to perform them. The recovery procedure thus reduces to doing installs just for all-or-nothing actions that were interrupted between the logging of their OUTCOME and END records. Recovery with redo logging can thus be quite swift, though it still requires both a backward and forward scan of the entire log.

We can summarize the procedures for atomicity logging as follows:

- Log to journal storage before installing in cell storage (WAL protocol)

- If all-or-nothing actions perform *all* installs to non-volatile storage before logging their OUTCOME record, then recovery needs only to undo the installs of incomplete uncommitted actions. (roll-back/undo recovery)

- If all-or-nothing actions perform *no* installs to non-volatile storage before logging their OUTCOME record, then recovery needs only to redo the installs of incomplete committed actions. (intentions-list/redo recovery)

- If all-or-nothing actions are not disciplined about when they do installs to non-volatile storage, then recovery needs to both redo the installs of incomplete committed actions *and* undo the installs of incomplete uncommitted ones.

In addition to reading and updating memory, an all-or-nothing action may also need to send messages, for example, to report its success to the outside world. The action of sending a message is just like any other component action of the all-or-nothing action. To provide all-or-nothing atomicity, message sending can be handled in a way analogous to memory update. That is, log a CHANGE record with a redo action that sends the message. If a crash occurs after the all-or-nothing action commits, the recovery procedure will perform this redo action along with other redo actions that perform installs. In principle, one could also log an *undo_action* that sends a compensating message ("Please ignore my previous communication!"). However, an all-or-nothing action will usually be careful not to actually send any messages until after the action commits, so intentions-list recovery applies. For this reason, a designer would not normally specify an undo action for a message or for any other action that has outside-world visibility such as printing a receipt, opening a cash drawer, drilling a hole, or firing a missile.

5. Checkpoints

Constraining the order of installs to be all before or all after the logging of the OUTCOME record is not the only thing we could do to speed up recovery. Another technique that can shorten the log scan is to occasionally write some additional information, known as a *checkpoint,* to non-volatile storage. Although the principle is always the same, the exact information that is placed in a checkpoint varies from one system to another. A checkpoint can include information written either to cell storage or to the log (where it is known as a *checkpoint record*) or both.

Suppose, for example, that the logging system maintains in volatile memory a list of identifiers of incomplete all-or-nothing actions and their pending/committed/aborted/ended status, keeping it up to date by observing the calls that log BEGIN, OUTCOME, and END records. The logging system then occasionally logs this list as a CHECKPOINT record. When a crash occurs sometime later, the recovery procedure begins a LIFO log scan as usual, collecting the sets of completed actions and losers. When it comes to a CHECKPOINT record it can immediately fill out the set of losers by adding those all-or-nothing actions that were listed in the checkpoint that did not later log an END record. This list may include some all-or-nothing actions listed in the CHECKPOINT record as COMMITTED, but that did not log an END record by the time of the crash. Their installs still need to be performed, so they need to be added to the set of losers. The LIFO scan continues, but only until it has found the BEGIN record of every loser.

With the addition of CHECKPOINT records, the recovery procedure becomes more complex, but is potentially shorter in time and effort:

1. Do a LIFO scan of the log back to the last CHECKPOINT record, collecting identifiers of losers and undoing all actions they logged.

2. Complete the lists of losers from information in the checkpoint.

3. Continue the LIFO scan, undoing the actions of losers, until finding every BEGIN record belonging to every loser.

4. Perform a forward scan from that point to the end of the log, performing any committed actions belonging to all-or-nothing actions in the list of losers that logged an OUTCOME record with status COMMITTED.

In systems in which long-running all-or-nothing actions are uncommon, step 3 will typically be quite brief or even empty, greatly shortening recovery. A good exercise is to modify the recovery program of figure 9–22 to accommodate checkpoints.

Checkpoints are also used with in-memory databases, to provide durability without the need to reprocess the entire log after every system crash. A useful checkpoint procedure for an in-memory database is to make a snapshot of the complete database, writing it to one of two alternating (for all-or-nothing atomicity) dedicated non-volatile storage regions, and then logging a CHECKPOINT record that contains the address of the latest snapshot. Recovery then involves scanning the log back to the most recent CHECKPOINT record, collecting a list of committed all-or-nothing actions, restoring the snapshot described there, and then performing redo actions of those committed actions from the CHECKPOINT record to the end of the log. The main challenge in this scenario is isolating the writing of the snapshot from any concurrent update activity. That can be done either by preventing all updates for the duration of the snapshot or by applying more complex isolation techniques such as those described in later sections of this chapter.

6. What if the cache is not write-through? (advanced topic)

Between the log and the write-through cache, the logging configurations just described require, for every data update, two synchronous writes to non-volatile storage, with attendant delays waiting for the writes to complete. Since the original reason for introducing a log was to increase performance, these two synchronous write delays usually become the system performance bottleneck. Designers who are interested in maximizing performance would prefer to use a cache that is not write-through, so that writes can be deferred until a convenient time when they can be done in batches. Unfortunately, the application then loses control of the order in which things are actually written to non-volatile storage. Loss of control of order has a significant impact on our all-or-nothing atomicity algorithms, since they require, for correctness, constraints on the order of writes and certainty about which writes have been done.

The first concern is for the log itself, because the write-ahead log protocol requires that appending a CHANGE record to the log precede the corresponding install in cell storage. One simple way to enforce the WAL protocol is to make just log writes write-through, but allow

cell storage writes to occur whenever the cache manager finds it convenient. However, this relaxation means that if the system crashes there is no assurance that any particular install has actually migrated to non-volatile storage. The recovery procedure, assuming the worst, cannot take advantage of checkpoints and must again perform installs starting from the beginning of the log. To avoid that possibility, the usual design response is to flush the cache as part of logging each checkpoint record. Unfortunately, flushing the cache and logging the checkpoint must be done as a single atomic action that is isolated from concurrent updates, which creates another design challenge. This challenge is surmountable, but the complexity is increasing.

Some systems pursue performance even farther. A popular technique is to write the log to a volatile buffer, and *force* that entire buffer to non-volatile storage only when an all-or-nothing action commits. This strategy allows batching several CHANGE records with the next OUTCOME record in a single synchronous write. Although this step would appear to violate the write-ahead log protocol, that protocol can be restored by making the cache used for cell storage a bit more elaborate; its management algorithm must avoid writing back any install for which the corresponding log record is still in the volatile buffer. The trick is to *number* each log record in sequence, and tag each record in the cell storage cache with the sequence number of its log record. Whenever the system forces the log, it tells the cache manager the sequence number of the last log record that it wrote, and the cache manager is careful never to write back any cache record that is tagged with a higher log sequence number.

We have in this section seen some good examples of the *law of diminishing returns* at work: schemes that improve performance sometimes require significantly increased complexity. Before undertaking any such scheme, it is essential to evaluate carefully how much extra performance one stands to gain.