6 CENTRALIZED RECOVERY

6.1 FAILURES

Beginning with this chapter we turn to the question of how to process transactions in a fault-tolerant manner. In this chapter we will explore this issue for centralized DBSs. We treat failure handling for distributed DBSs in Chapter 7 and 8.

Our first task is to define the sorts of faults we are going to consider. Computer systems fail in many ways. It is not realistic to expect to build DBSs that can tolerate all possible faults. However, a good system must be capable of recovering from the most common types of failures automatically, that is, without human intervention.

Many failures are due to incorrectly programmed transactions and data entry errors (supplying incorrect parameters to transactions). Unfortunately, these failures undermine the assumption that a transaction's execution preserves the consistency of the database. They can be dealt with by applying software engineering techniques to the programming and testing of transactions, or by semantic integrity mechanisms built into the DBS. However they're dealt with, they are intrinsically outside the range of problems our recovery mechanisms can automatically solve. Thus, we assume that transactions indeed satisfy their defining characteristic, namely, that they halt for all inputs and their execution preserves database consistency.

Many failures are due to operator error. For example, an operator at the console may incorrectly type a command that damages portions of the database, or causes the computer to reboot. Similarly, a computer technician may damage a disk or tape during a computer maintenance procedure. The risk of such errors can be reduced by better human engineering of the system's interface to operators and by improved operator education. Preventing these errors is outside the scope of problems treated by DBS recovery. However, DBS recovery mechanisms *are* designed to deal with some of the consequences of these errors, namely, the loss of data due to such errors.

Given these assumptions, there are three types of failures that are most important in centralized DBSs, known as *transaction failures, system failures* and *media failures*. A *transaction failure* occurs when a transaction aborts. A *system failure* refers to the loss or corruption of the contents of *volatile* storage (i.e., main memory). For example, this can happen to semiconductor memory when the power fails. It also happens when the operating system fails. Although an operating system failure may not corrupt all of main memory, it is usually too difficult to determine which parts were actually corrupted by the failure. So one generally assumes the worst and reinitializes all of main memory. Because of system failures, the database itself must be kept on a stable storage medium, such as disk. (Of course other considerations, such as size, may also force us to store the database on stable mass storage media.) By definition, *stable* (or *nonvolatile*) storage withstands system failures. A *media failure* occurs when any part of the stable storage is destroyed. For instance, this happens if some sectors of a disk become damaged.

The techniques used to cope with media failures are conceptually similar to those used to cope with system failures. In each case, we consider a certain part of storage to be unreliable: volatile storage, in the case of system failures; a portion of stable storage, in the case of media failures. To safeguard against the loss of data in unreliable storage, we maintain another copy of the data, possibly in a different representation. This redundant copy is kept in another part of storage that we deem reliable: stable storage, in the case of system failures, or another piece of stable storage, such as a second disk, in the case of media failures. Of course, the different physical characteristics of storage in the two cases may require the use of different strategies. But the principles are the same.

For pedagogical simplicity, we will focus principally on the problem of system failures. We explain how to extend recovery techniques for system failure to those for media failure in the last section of the chapter.

We'll assume that all failures are detected. This is not an issue with transaction failures, because a transaction failure by definition results in the execution of an Abort operation. But it is conceivable that volatile or stable storage gets corrupted without this being detected. Usually, storage devices have error detecting codes, such as parity checks, to detect bit errors in hardware; software can use redundant pointer structures and the like to detect data structure inconsistencies. While these techniques make an undetected failure highly unlikely, it *is* possible. In general, the techniques described here will not handle the occurrence of such an undetected system or media failure.

6.2 DATA MANAGER ARCHITECTURE

As in our discussion of schedulers, we'll continue using the TM-scheduler-DM model of a DBS. Unlike Chapters 3–5, where we focused on the scheduler, our center of attention will now shift to the DM. It's the DM that manipulates storage, and it's storage that is corrupted by failures. We will principally be concerned with system failures that can destroy volatile but not stable storage. We must therefore incorporate the distinction between volatile and stable storage into the model, which we briefly discussed in Section 1.4.

Let's review our model of a DBS, focusing on the issues that will occupy us in this chapter (see Fig. 6–1). Transactions submit their operations to the TM, which passes them on to the scheduler. The scheduler receives Read, Write, Commit, and Abort operations from the TM. The scheduler can pass Aborts to the DM immediately. For Read, Write, or Commit operations, the scheduler must decide, possibly after some delay, whether to reject or accept the operation. If it rejects the operation, the scheduler sends a negative acknowledgment to the TM, which sends an Abort back to the scheduler, which in turn promptly passes the Abort to the DM. If the scheduler accepts the operation, it sends it to the DM, which processes it by manipulating storage. The exact details of this storage manipulation depend on the DM algorithm, and are the main subject of this chapter. When the DM has finished processing the operation, it acknowledges to the scheduler, which passes the acknowledgment to the TM. For a Read, the acknowledgment includes the value read.

In addition to Read, Write, Commit and Abort, the DM may also receive a Restart operation. This is sent by an external module, such as the operating system, upon recovery from a system failure. The task of Restart is to bring the database to a consistent state, removing effects of uncommitted transactions and applying missing effects of committed ones. To be more precise, define the *last committed value* of a data item x in some execution to be the value last written into x in that execution by a committed transaction. Define the *committed database state* with respect to a given execution to be the state in which each data item contains its last committed value. The goal of Restart is to restore the database into its committed state with respect to the execution up to the system failure.

To see why this is the right thing to do, let's use the tools of Chapter 2. Let H be the history representing the partial order of operations processed by the DM up to the time of the system failure. The committed projection of H, $C(H)$, is obtained by deleting from H all operations not belonging to the committed transactions. If H was produced by a correct scheduler, then it is recoverable. Consequently, the values read or written in $C(H)$ are identical to the values read or written by the corresponding operations in H. Therefore, by restoring the last committed value of each data item, Restart makes the database reflect the execution represented by the history $C(H)$, that is, the execution of precisely the transactions that were committed at the time of the system failure. Moreover, $C(H)$ is SR, because it was produced by a correct scheduler.

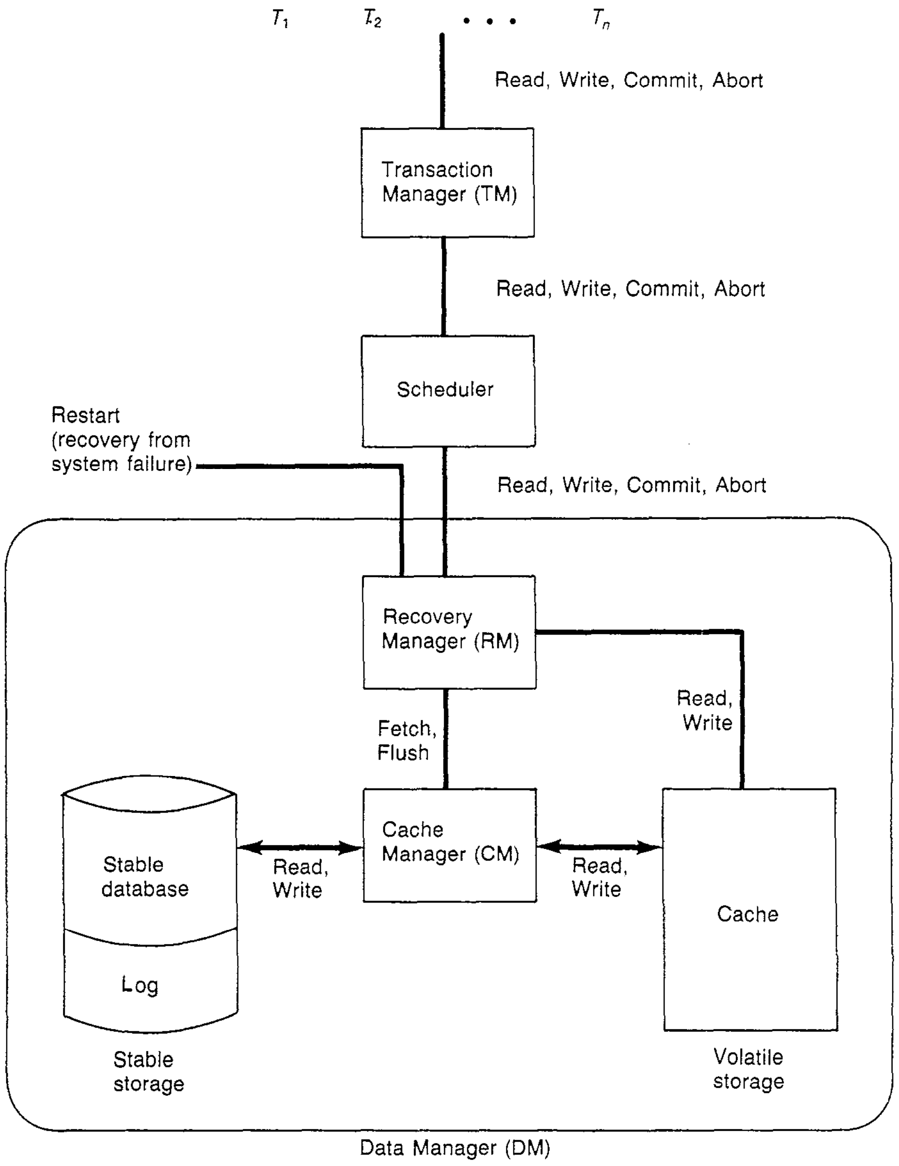

FIGURE 6–1
Model of a Centralized Database System

So when Restart terminates, the database is in a consistent state. A major goal of this chapter is to explain what data structures must be maintained by the DM so that Restart can do this using *only* information saved in stable storage.

Stable Storage

The DM is split into two components: a *cache manager* (*CM*), which manipulates storage, and a *recovery manager* (*RM*), which processes Read, Write, Commit, Abort, and Restart operations. The CM provides operations to *fetch* data from stable storage into volatile storage, and to *flush* data from volatile to stable storage. The RM partially controls the CM's flush operations, to ensure that stable storage always has the data that the RM needs to process a Restart correctly, should the need arise.

We assume that when the CM issues a Write operation to write a data item in stable storage, *the Write either executes in its entirety or not at all* and responds with a return code indicating which of the two outcomes occurred. This holds even if the system fails while the Write executes. Such Writes are called *atomic*. If a Write fails to be atomic (e.g., it modifies some but not all of a data item), then a media failure has occurred. For now, we assume that media failures do not occur. That is, all Writes are atomic. We treat media failures in Section 6.8.

For disks, currently the most popular form of stable storage, the granularity of data item that can be written is usually a fixed-sized *page* (or *block*). When a page is written to disk, there are two possible results: the old value of the page is correctly overwritten, and remains in the new state until it is overwritten again, or the new value of the page is corrupted somehow, in which case the error is detected when the page is subsequently read. That is, it either executes correctly or results in a media failure. Error detection is normally supported by the disk hardware. If a small number of bit errors alters the contents of a page, the disk hardware will detect the error through a checksum that is calculated when it reads the page. The checksum may also be a function of the page's disk address, *da*, thereby ensuring that the page that is read is one that was previously written to *da* (i.e., not one that was intended to be written to some other address but was incorrectly written to *da*). When these sorts of hardware error detection are unavailable, one can partially compensate for their absence using software error detection, with some degradation of performance (see Exercise 6.1).

The granularity of data items that are parameters to Writes issued to the DM may be different from that which can be atomically written to stable storage. That is, if stable storage supports atomic Writes to pages, the DM may receive Writes to short records (where each page may contain many such records) or to long records (which can span more than one page). This mismatch of data item granularity requires special attention when designing recovery algorithms, since individual data items cannot be written one by one (for short records) or atomically (for long records).

In this chapter, unless otherwise noted, we assume that the granularity of data items supported by the DM is identical to that supported by stable storage. With today's disk technology, this means a data item is a fixed-size page. We will also discuss some of the special problems that arise due to granularity mismatches. However, in these cases we will be careful to emphasize that we have abandoned the assumption of identical DM and stable storage granularities of data items (see also Exercise 6.2).

We distinguish between DMs that keep exactly one copy of each data item in stable storage and DMs that may keep more than one. If there is only one copy, then each time a data item is overwritten, the old value is destroyed. This is called *in-place updating*. If there is more than one copy, then the CM may write a data item to stable storage without destroying the older versions of that data item. The older versions are called *shadow copies*, and this technique is called *shadowing*.

With shadowing, the mapping of data items to stable storage locations changes over time. It is therefore convenient to implement this mapping using a *directory*, with one entry per data item, giving the name of the data item and its stable storage location. Such a directory defines a state of the database (see Fig. 6–2). With shadowing, there is usually more than one such directory, each directory identifying a different state of the database.

We define the *stable database* to be the state of the database in stable storage. With in-place updating, there is exactly one copy of each data item in stable storage, so the concept is well defined. With shadowing, we assume that there is a particular directory, *D*, in stable storage that defines the current state of the stable database. The copies of data items in stable storage that are referenced by directories other than *D* are shadow copies.

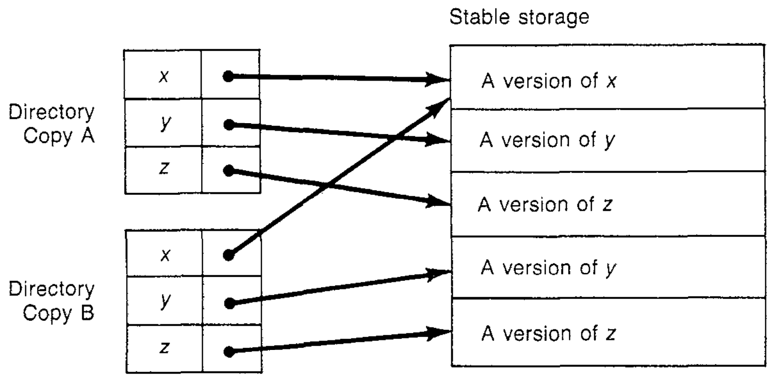

FIGURE 6–2
An Example of Shadowing
Using shadowing, directory copies A and B each define a database state.

The Cache Manager

To avoid accessing stable storage to process every Read and Write, we would like to keep a copy of the database in volatile storage. To read a data item we would simply obtain its value from the volatile storage copy. To write a data item we would record the new value in both the volatile and stable storage copies. The stable storage copy would be useful only for recovery from system failures. Since access to stable storage is slower than to volatile storage, this would improve performance in almost all cases. Unfortunately, keeping a copy of the entire database in volatile storage is usually too expensive due to the large size of databases and the relatively high cost of main memory. However, as main memory prices continue to drop, this approach may become more common in the future.

In any case, currently, we must cope with keeping less than the whole database in volatile storage. This can be done by using a technique called *caching* or *buffering*, similar to that of hardware caching and operating system virtual memory. A portion of volatile storage, called the *cache*, is reserved for holding portions of the database. The cache consists of a collection of *slots*, each of which can store the value of a data item (see Fig. 6–3). The granularity of data items stored in slots is that which can be atomically written to stable storage (i.e., a page). At any time a certain subset of data items occupies slots in the cache, in addition to occupying their more permanent locations in stable storage. A cache slot contains a value for the data item stored in that slot, and a *dirty bit* that is set if and only if the value of the data item stored in the cache slot is different from its value in stable storage (we'll see how this can arise momentarily). If the dirty bit is set, we say the slot is *dirty*. There is also a *cache directory* that gives the name of each data item in the cache and the number of its associated slot.

Cache

Slot Number	Dirty Bit	Data Item Value
1	1	"October12"
2	0	3.1416
	⋮	

Cache Directory

Data Item Name	Slot Number
x	2
y	1
⋮	

FIGURE 6-3
Cache Structure

The traffic of data items into and out of the cache is controlled by the CM via two operations: Flush and Fetch. *Flush* takes a cache slot *c* as its parameter. If *c* is not dirty, then Flush does nothing. If *c* is dirty, it copies *c*'s value into the stable storage location of the data item stored in *c*, and clears *c*'s dirty bit. Flush does not return to its caller until the update of *c* on stable storage has completed.

Notice that Flush must map each data item to a stable storage location. If the CM uses shadowing, then the mapping is via a directory. The CM's choice and manipulation of directories depends on the recovery algorithm. If the CM uses in-place updating, then the mapping is unique. In this case, the method for performing the mapping is unimportant to our study.

Fetch takes a data item name *x* as its parameter. It causes the CM to perform the following actions:

1. It selects an empty cache slot, say *c*. If all cache slots are occupied, it selects some slot *c*, flushes *c*, and uses that as its empty slot.
2. It copies the value of *x* from its stable storage location into *c*.
3. It clears the dirty bit of *c*.
4. It updates the cache directory to indicate that *x* now occupies *c*.

If slot *c* was occupied in step (1), we say that *x* *replaces* the data item that occupied *c*. The criterion according to which the CM chooses *c* is called the *replacement strategy*. Some well known replacement strategies are *least recently used* (LRU) and *first-in-first-out* (FIFO), specifying, respectively, that the slot least recently accessed or least recently fetched be used for replacement.

To read a data item named *x*, the CM fetches the value of *x* if it is not already in the cache, and returns this value from the cache. To write *x*, the CM allocates a slot for *x* if it is not already in the cache, records the new value in the cache slot, and sets the dirty bit for the cache slot. Whether it flushes the new value of *x* to stable storage at this point or later on is a decision left to the RM. As we'll see, different RM algorithms use different strategies with respect to this issue.

There will be times when the RM must ensure that a cache slot is not flushed for some time period, for example, while it is updating the contents of the slot. For this reason, the CM offers two additional operations, *Pin* and *Unpin*. The operation Pin(*c*) tells the CM not to flush *c*, while Unpin(*c*) makes a previously pinned slot again available for flushing. Thus, the CM never flushes a slot while it is pinned.

6.3 THE RECOVERY MANAGER

The RM interface is defined by five procedures:

1. *RM-Read*(T_i, *x*)*:* read the value of *x* for transaction T_i;
2. *RM-Write*(T_i, *x*, *v*)*:* write *v* into *x* on behalf of transaction T_i;

3. *RM-Commit(T_i):* commit T_i;
4. *RM-Abort(T_i):* abort T_i; and
5. *Restart:* bring the stable database to the committed state following a system failure.

The RM should execute these operations atomically. That is, the RM's execution should be equivalent to a serial execution of these operations. This requirement is easily enforced if a 2PL, TO, or SGT scheduler is used. These schedulers never send two conflicting operations to the RM concurrently, so the RM can safely process any of its pending Reads and Writes concurrently. However, the RM may also access local data structures that are shared by the execution of two operations. For example, an RM-Commit and RM-Abort of two different transactions may both update local lists of active and terminated transactions. To ensure that these operations are atomic with respect to each other, the RM must synchronize access to these shared structures, for example by using semaphores or locks.

We assume that the scheduler invokes RM operations in an order that produces a serializable and strict execution. Since executions are strict, committed Writes will execute in the same order that their corresponding transactions commit. In particular, the last committed value of x will be written by the last committed transaction that wrote into x.

The RM algorithms become considerably more complicated under the weaker assumption that the scheduler produces an execution that is serializable and recoverable, the weakest possible requirements on the scheduler that will not compromise correctness.

Recall from Sections 1.2 and 2.4 that strict executions avoid cascading aborts. Thus, to erase the effects of an aborted transaction from the database, we merely have to restore in the database the before images of its Writes. To establish the terminology that we'll use later in the chapter, suppose T_i wrote into x: The *before image of x with respect to (wrt) T_i* is the value of x just before T_i wrote into it; the *after image of x wrt T_i* is the value written into x by T_i.

Logging

Suppose the RM uses in-place updating. Then, each data item has a unique location in stable storage. Ideally, the stable database would contain, for each x, the last value written into x by a committed transaction. Practically, two factors prevent this ideal state: the continual updating by transactions that take some time to terminate, and the buffering of data items in cache. Therefore, the stable database might contain values written by uncommitted transactions, or might not contain values written by committed ones.

In the event of a system failure, the RM's Restart operation must be able to transform the stable database state into the committed database state. In doing this, it can only rely on data in stable storage. For this reason, the RM usually stores information in stable storage in addition to the stable database itself. A *log* is one such type of information.

Conceptually, a log is a representation of the history of execution. A *physical log* is a type of log that contains information about the values of data items written by transactions. Like the stable database, the structure and contents of the log are highly dependent on the RM algorithm. Abstractly, however, we can think of a (physical) log as consisting of entries of the form $[T_i, x, v]$, identifying the value v that transaction T_i wrote into data item x.

The data structures used for the log must enable Restart to determine, for each x, which log entry contains x's last committed value. Thus, they must encode the *order* in which Writes occurred. An easy way to record this information is to require that the log be a sequential file and that the entries in the log be consistent with the order of their corresponding Writes. Thus, we assume that $[T_i, x, u]$ precedes $[T_j, x, v]$ in the log iff $w_i[x]$ executed before $w_j[x]$. Since we assume a strict execution, if $[T_i, x, u]$ precedes $[T_j, x, v]$ in the log and both T_i and T_j committed, then T_i committed before T_j.

Instead of containing values that were written into the database, a log may contain descriptions of higher level operations. This is called *logical logging*. For example, a log entry may say that "record r was inserted into file F, and F's indices were updated to reflect this insertion." Using logical logging, only this one log entry is recorded, instead of several log entries corresponding to the physical Writes of F and its indices. By recording higher level operations, fewer log entries are needed. Shortening the log in this way can improve the RM's performance, but sometimes at the expense of added complexity in interpreting log entries by the Restart algorithm. We will deal with the complexity of logical logging later in the chapter. Unless otherwise noted, when we use the term "log," we mean a physical log.

In addition to the stable database and the log, the RM may also keep in stable storage one or more of the *active*, *commit* and *abort* lists. These lists contain the identifiers of the set of transactions that are active, committed or aborted (respectively). These lists are often stored as part of the log.

In most RM algorithms, it is the act of adding a transaction identifier to the commit list that causes the transaction to become committed. Thus, after a system failure, the RM regards a transaction as having been committed iff its transaction identifier is in the commit list.

Undo and Redo

Whatever replacement strategy the CM uses, there are times when the RM must *insist* that the CM flush certain data items to stable storage. These flushes coordinate writing the stable database and the log, so that Restart will always find the information it needs in stable storage, be it in the stable database or the log. In this section we will investigate flushing requirements that all RMs must satisfy. This will lead to a natural categorization of RM algorithms that will carry us through the rest of the chapter.

We say that an RM *requires undo* if it allows an uncommitted transaction to record *in the stable database* values it wrote. Should a system failure occur at this point, on recovery the stable database will contain effects of the uncommitted transaction. These effects must be *undone* by Restart in order to restore the stable database to its committed state with respect to the execution up to the failure.

We say that an RM *requires redo* if it allows a transaction to commit before all the values it wrote have been recorded *in the stable database.* If a system failure occurs at this point, on recovery the stable database will be missing some of the effects of the committed transaction. These must be *redone* by Restart to restore the stable database to its committed state.

Notice that we use the terms "requires undo" and "requires redo" only relative to system failures, not media failures. That is, when we say that "an RM requires redo," we really mean that "the RM's Restart procedure requires redo for handling system failures." We treat the undo and redo requirements for media failures in Section 6.8.[1]

By regulating the order of a transaction's commitment relative to writing its values in the stable database, an RM can control whether it requires undo or redo. Thus, we can classify RM algorithms into four categories: (1) those that require both undo and redo; (2) those that require undo but not redo; (3) those that require redo but not undo; and (4) those that require neither undo nor redo.[2] Implementations of all four types have been proposed and we'll examine them later in this chapter.

The prospect that an RM may require undo or redo should raise some concern in view of our desire to be able to recover from system failures. In particular, since the stable database may contain inappropriate updates or be missing appropriate ones, the RM had better keep sufficient information in the log for Restart to undo the former and redo the latter. The requirements implied by this can be conveniently stated as two design rules that all RM implementations must observe.

Undo Rule:[3] If x's location in the stable database presently contains the last committed value of x, then that value must be saved in stable storage *before* being overwritten in the stable database by an uncommitted value.

[1]Recovery mechanisms for media failure generally require redo. Most such mechanisms keep a stable copy of the database, called the *archive*, which is almost surely out-of-date. So the RM must redo committed updates that occurred after the archive was created.

[2]In the paradigm of [Haerder, Reuter 83], these categories correspond to (1) Steal/No-Force, (2) Steal/Force, (3) No-Steal/No-Force, and (4) No-Steal/Force.

[3]This is often called the write ahead log protocol, because it requires that the before image of a Write be logged ahead of the Write being installed in the stable database.

Redo Rule: Before a transaction can commit, the value it wrote for each data item must be in stable storage (e.g., in the stable database or the log).

The Undo and Redo Rules ensure that the last committed value of each data item is always available in stable storage. The Undo Rule ensures that the last committed value of x is saved in stable storage (e.g., in the log) before being overwritten by an uncommitted value. And the Redo Rule ensures that a value is in stable storage at the moment it becomes committed. Observe that an RM that does *not* require undo (or redo) necessarily satisfies the Undo (or Redo) Rule.

Garbage Collection

Even though stable storage is usually abundant, it has bounded capacity. It is therefore necessary to restrict the amount of information the RM keeps in stable storage. Clearly, the size of the stable database is bounded by the number of data items it stores. To satisfy the Undo and Redo Rules, the RM has to keep inserting information into the log. To bound the growth of the log, the RM must free up and recycle any space used to store information that it can be certain will never be needed by Restart. Recycling space occupied by unnecessary information is called *garbage collection.*

Restart's requirement for log information is: for each data item x, if the stable database copy of x does not contain the last committed value of x, then Restart must be able to find that value in the log. To restate this requirement in terms of log entries requires a knowledge of the detailed structure of log entries. In terms of our abstract log entries of the form $[T_i, x, v]$, the following rule tells precisely what information can be dispensed with at any given time, insofar as Restart is concerned.

Garbage Collection Rule: The entry $[T_i, x, v]$ can be removed from the log iff (1) T_i has aborted or (2) T_i has committed but some other committed transaction wrote into x after T_i did (hence v is not the last committed value of x).

In case (1), once T_i has aborted we no longer care about what it wrote, since all of its Writes will be undone. In case (2), once a newly committed value is written into x, older committed values can be disposed of, because only the last committed value is needed for recovery. In both cases, deleting $[T_i, x, v]$ never affects Restart's ability to determine which Write on x is the last committed one, as long as the order of committed Writes is recorded in the log.

Notice that even if v is the last committed value of x and is stored in the stable database copy of x, $[T_i, x, v]$ cannot be deleted from the log. If a transaction T_j subsequently wrote into x in the stable database and then aborted, then $[T_i, x, v]$ would be needed to undo T_j's Write. However, if the RM does not require undo, then T_j's Write cannot be flushed to the stable database before T_j commits, so this scenario cannot occur. Therefore, we can augment the Garbage Collection Rule with a third case: (3) $[T_i, x, v]$ can be removed from the log if the RM does not require undo, v is the last committed value of x, v is the value of x in the stable database, and $[T_i, x, v]$ is the only log entry for x. The last condition ensures that Restart does not misinterpret some earlier log entry for x to be x's last committed value, after $[T_i, x, v]$ is deleted.

The Garbage Collection Rule defines the *earliest* time that a log entry can be deleted. Some Restart algorithms require keeping the log entries longer than required by the Garbage Collection Rule. This rule may also be overruled by the requirements of media recovery (see Section 6.8).[4]

As we mentioned before, the RM may also keep in stable storage one or more of the active, abort, or commit lists. The size of the active list is bounded by the number of transactions that are in progress at any given time, presumably a manageable number. The RM somehow has to bound the size of the abort and commit lists, since it would be impractical to keep a record of all transactions that have committed or aborted since the genesis of the system. In practice, the RM only needs to know about recently committed or aborted transactions. Exactly what "recently" means depends on the details of the RM algorithm.

Idempotence of Restart

Although Read, Write, Commit, and Abort execute atomically with respect to each other, Restart can interrupt any of them, because a system failure can happen at any time. Indeed, Restart can even interrupt its own execution, should a system failure occur while Restart is recovering from an earlier failure. That Restart can interrupt itself leads to another important requirement: Restart must be *idempotent*. This means that if Restart stops executing at any moment and starts executing again from the beginning, it produces the same result in the stable database as if the first execution had run to completion. Said more abstractly, it means that any sequence of incomplete executions of Restart followed by a complete execution of Restart has the same effect as just one complete execution of Restart.

If an execution of Restart is interrupted by a system failure, then a second execution of Restart will begin by reinitializing volatile storage, since it assumes that this storage was corrupted by the failure. However, it will accept the state of stable storage as is, including updates produced by the first execution of Restart. Thus, the pragmatic implication of idempotence is that Restart should ensure that stable storage is always in a state that a new execution of

[4]For media recovery, the Garbage Collection Rule applies, but with respect to the archive database. That is, case (2) applies iff some other transaction wrote into x in *the archive database* after T_i did.

Restart can properly interpret. This amounts to being careful about the values Restart writes to stable storage and the order in which it writes them. In this sense, idempotence is similar to the Undo and Redo Rules, in that it also restricts the order of certain updates to stable storage so that Restart can do its job.

Preview

In the following sections, we'll describe the four different types of RMs mentioned previously: undo/redo, undo/no-redo, no-undo/redo, and no-undo/no-redo. For convenience of exposition, we present an RM as a collection of five procedures representing the RM's five operations: RM-Read, RM-Write, RM-Commit, RM-Abort, and Restart. In each of the four RM algorithms presented, we first describe the five procedures at a fairly abstract level. The purpose is to state *what* the RM has to do, rather than *how* it does it. The "how" issues are taken up later, when we examine various implementation strategies.

6.4 THE UNDO/REDO ALGORITHM

In this section we describe an RM algorithm that requires both undo and redo. This is the most complicated of the four types of RMs. However, it has the significant advantage of being very flexible about deciding when to flush dirty cache slots into the stable database. It leaves the decision to flush almost entirely to the CM. This is desirable for two reasons. First, an RM that uses undo/redo avoids forcing the CM to flush unnecessarily, thereby minimizing I/O. By contrast, a no-redo algorithm generally flushes more frequently, since it must ensure that all of a transaction's updated items are in the stable database before the transaction commits. Second, it allows a CM that uses in-place updating to replace a dirty slot last written by an uncommitted transaction. A no-undo algorithm cannot flush the slot in this case, since it would be writing an uncommitted update into the stable database. In general, the undo/redo algorithm is geared to maximize efficiency during normal operation, at the expense of less efficient processing of failures than is possible with other algorithms.

Suppose transaction T_i writes value v into data item x. In this algorithm, the RM fetches x if it isn't already in cache, records v in the log and in x's cache slot, c, but does not ask the CM to flush c. The CM only flushes c when it needs to replace x to free up c for another fetch. Thus, recording an update in the stable database is in the hands of the CM, not the RM. If the CM replaces c and either T_i aborts or the system fails before T_i commits, then undo will be required. On the other hand, if T_i commits and the system fails before the CM replaces c, then redo will be required.

We assume the granularity of data items on which RM procedures operate is the same as the unit of atomic transfer to stable storage. We assume a physical log that is an ordered list of records of the form $[T_i, x, v]$, and that is recycled according to the Garbage Collection Rule. We assume that the initial value of each data item is written in the log before any transactions are processed. (Alternatively, the first Write to each data item can store the initial value of the data item in the log.) Each update of the log and the commit list goes directly to the stable storage device and must be acknowledged by that device before proceeding to the next step.

We now outline the five RM procedures for this algorithm.

RM-Write(T_i, x, v)

1. Add T_i to the active list, if it's not already there.
2. If x is not in the cache, fetch it.
3. Append $[T_i, x, v]$ to the log.
4. Write v into the cache slot occupied by x.(A) [5]
5. Acknowledge the processing of RM-Write(T_i, x, v) to the scheduler.

RM-Read(T_i, x)

1. If x is not in the cache, fetch it.
2. Return the value in x's cache slot to the scheduler.

RM-Commit(T_i)

1. Add T_i to the commit list.(B)
2. Acknowledge the commitment of T_i to the scheduler.
3. Delete T_i from the active list.(C)

RM-Abort(T_i)

1. For each data item x updated by T_i:
   - if x is not in the cache, allocate a slot for it;
   - copy the before image of x wrt T_i into x's cache slot.(D)
2. Add T_i to the abort list.
3. Acknowledge the abortion of T_i to the scheduler.
4. Delete T_i from the active list.

Restart

1. Discard all cache slots.(E)

[5]All comments are listed after the descriptions of the operations and are cross-referenced with superscripted capital letters.

2. Let *redone* := { } and *undone* := { }.[F]
3. Start with the last entry in the log and scan backwards toward the beginning. Repeat the following steps until either *redone* ∪ *undone* equals the set of all data items in the database, or there are no more log entries to examine. For each log entry $[T_i, x, v]$, if $x \notin$ *redone* ∪ *undone*, then
   - if x is not in the cache, allocate a slot for it;
   - if T_i is in the commit list, copy v into x's cache slot[G] and set *redone* := *redone* ∪ $\{x\}$;
   - otherwise (i.e., T_i is in the abort list or in the active but not the commit list), copy the before image of x wrt T_i[H] into x's cache slot and set *undone* := *undone* ∪ $\{x\}$.
4. For each T_i in the commit list, if T_i is in the active list, remove it from there.
5. Acknowledge the completion of Restart to the scheduler.[I]

Comments

A. [Step (4) of RM-Write] To avoid repetitive comments, we assume here, and through the rest of the chapter, that when a cache slot is written into (thus making its value different from the value of the corresponding location in the stable database), the RM sets the slot's dirty bit. We also assume that the RM pins a cache slot before reading or writing it, and unpins it afterwards, thereby ensuring that RM-Reads and RM-Writes are atomic wrt flushes.

B. [Step (1) of RM-Commit(T_i)] It is the act of adding T_i to the commit list (in stable storage) that declares T_i committed. Should a system failure occur before this step completes, T_i will be considered uncommitted. Otherwise it will be considered committed even if Step (2) of RM-Commit(T_i) was not completed.

C. [Step (3) of RM-Commit] The significance of the active and abort lists will be discussed in a later subsection, on checkpointing.

D. [Step (1) of RM-Abort] At this point, the before image is only restored in the cache. The CM will restore it in the stable database when it replaces x's cache slot.

E. [Step (1) of Restart] A system failure destroys the contents of volatile storage, and hence the cache. Thus when Restart is invoked, the values in the cache cannot be trusted.

F. [Step (2) of Restart] *redone* and *undone* are variables local to Restart that keep track of which data items have been restored to their last committed value by a redo or undo action (respectively).

G. [Step (3) of Restart] At this point, x's last committed value is only restored in the cache. The CM will restore it in the stable database when it replaces x's cache slot.

H. [Step (3) of Restart] The before image of x wrt T_i can be found in the log.

I. [Step (4) of Restart] Upon recovery from a system failure, the scheduler must wait for the acknowledgment that Restart has been completed by the RM. It may then start sending operations to the RM again.

Undo and Redo Rules

This algorithm satisfies the Undo Rule. Suppose the location of x in the stable database contains the last committed value of x, say v, written by transaction T_i. When T_i wrote into x, the RM inserted $[T_i, x, v]$ in the log (see step (2) of RM-Write). By the Garbage Collection Rule, since v is the last committed value of x, this record cannot have been removed. In particular, it will still be in the log when the CM overwrites v in the stable database, as desired for the Undo Rule.

The Redo Rule is likewise satisfied. All of a transaction's updates are recorded in the log before the transaction commits, whether or not they were also recorded in the stable database. By the Garbage Collection Rule they must still be there when the transaction commits.

Since the algorithm satisfies the Undo and Redo Rules, Restart can always find in stable storage the information that it needs for restoring the last committed value of each data item in the stable database. In step (3), Restart redoes an update to x by a committed transaction, or undoes an update to x by an active or aborted transaction, only if no other committed transaction subsequently updated x. Thus, when Restart terminates, each data item will contain its last committed value. Moreover, Restart is idempotent. If it is interrupted by a system failure and reexecutes from the beginning, the updates it redoes and undoes in step (3) are the same as those it would have done if its first execution had not been interrupted.

Checkpointing

The Restart procedure just sketched may have to examine every record ever written in the log — except, of course, those that have been garbage collected. This is still a very large number of records, since garbage collection is an expensive activity and is carried out fairly infrequently. Moreover, since most data items in the database probably contain their last committed values at the time of the failure, Restart is doing much more work than necessary. This inefficiency of Restart is an important issue, because after a system failure, the DBS is unavailable to users until Restart has finished its job.

This problem is solved by the use of checkpointing methods. In general, *checkpointing* is an activity that writes information to stable storage during normal operation in order to reduce the amount of work Restart has to do after a failure.

Checkpointing performs its work by a combination of two types of updates to stable storage: (1) marking the log, commit list, and abort list to indicate which updates are already written or undone in the stable database, and (2) writing the after images of committed updates or before images of aborted updates in the stable database. Technique (1) tells Restart *which* updates don't have to be undone or redone again. Technique (2) reduces the amount of work that Restart has to do by doing that work during checkpointing. Technique (1) is essential to any checkpointing activity. Technique (2) is optional.

One simple checkpointing scheme is periodically to stop processing transactions, wait for all active transactions to commit or abort, flush all dirty cache slots, and then mark the end of the log to indicate that the checkpointing activity took place. This is called *commit consistent checkpointing*, because the stable database now contains the last committed value of each data item relative to the transactions whose activity is recorded in the log. With commit consistent checkpointing, Restart scans the log backward, beginning at the end, undoing and redoing updates corresponding to log records, until it reaches the last checkpoint marker. It may have to examine log records that precede the marker in order to find certain before images: namely, the before images of each data item that was updated after the last checkpoint marker, but not by any committed transaction.

The activity of checkpointing and the stable database created by checkpointing are both sometimes called *checkpoints*; we will use the word *Checkpoint* (capital "C") as the name of the procedure that performs the checkpointing activity.

The main problem with this Checkpoint procedure is performance. Users may suffer a long delay waiting for active transactions to complete and the cache to be flushed. We can eliminate the first of these two delays by using the following Checkpoint procedure, called *cache consistent checkpointing*, which ensures that all Writes written to cache are also in the stable database.

Periodically, Checkpoint causes the RM to stop processing other operations (temporarily leaving active transactions in a blocked state), flushes all dirty cache slots, and places markers at the end of the log and abort list to indicate that the flushes took place. Consider now what Restart must do after a system failure assuming that this Checkpoint procedure is used. All updates of committed transactions that happened before the last Checkpoint were installed in the stable database during that Checkpoint and need not be redone. Similarly, all updates of transactions that aborted prior to the last Checkpoint were undone during that Checkpoint, and need not be undone again. Therefore Restart need only redo those updates of transactions in the commit list that appear after the last checkpoint marker in the log. And it need only undo updates of those transactions that are in the active (but not the commit) list, or are in the abort list and appear after the last checkpoint marker in that list.

This checkpointing scheme still delays transactions while the cache is being flushed. We can reduce this delay by using the following technique, called *fuzzy checkpointing*. Instead of flushing *all* dirty cache slots, the Checkpoint procedure only flushes those dirty slots that have not been flushed since *before* the previous checkpoint. The hope is that the CM's normal replacement activity will flush most cache slots that were dirty since before the previous checkpoint. Thus Checkpoint won't have much flushing to do, and therefore won't delay active transactions for very long.

This Checkpoint procedure guarantees that, at any time, all updates of committed transactions that occurred before the *penultimate* (i.e., second to last) Checkpoint have been applied to the stable database — during the last Checkpoint, if not earlier. Similarly, all updates of transactions that had aborted before the penultimate Checkpoint have been undone.

As in cache consistent checkpointing, after flushing the relevant cache slots, the fuzzy Checkpoint procedure appends markers to the log and to the abort list. Thus, after a system failure, Restart redoes just those updates of transactions in the commit list that come after the penultimate checkpoint marker in the log, and it undoes just the updates of those transactions that are either in the active (but not the commit) list, or are in the abort list and follow the penultimate checkpoint marker in that list. Checkpoint and Restart algorithms that use this strategy are described in detail in the next subsection.

In reviewing the checkpointing schemes we have discussed, we see at work a fundamental parameter of checkpointing: the maximum length of time, t, that a cache slot can remain dirty before being flushed. Increasing t decreases the work of Checkpoint and increases the work of Restart. That is, it speeds up the system during normal operation at the expense of slowing down the recovery activity after a system failure. Hence t is a system parameter that should be tuned to optimize this trade-off.

An Implementation of Undo/Redo

A problem with the undo/redo algorithm is that after images of data items can consume a lot of space. If data items are pages of a file, then each update to a page generates a little over a page of log information, consisting of the page's address, its after image, and the identifier of the transaction that wrote the page. This produces a lot of disk I/O, which can seriously degrade performance. And it consumes a lot of log space, which makes garbage collection of the log a major factor.

These problems are especially annoying if most updates only modify a small portion of a data item. If data items are pages, then an update might only modify a few fields of one record stored on that page. In this case, it would be more efficient to log only *partial data items*, namely, the portion of each data item that was actually modified. A log record should now also include the offset and length of the portion of the data item that was modified.

The algorithm described next logs partial data items and uses fuzzy checkpointing. We will call it the *partial data item logging algorithm*.

The algorithm incorporates the active, commit, and abort lists into the log, which is stored as a sequential file. Each log entry may now be made smaller than a data item, which is the unit of transfer to stable storage. It is therefore inefficient to write each log entry to stable storage at the time it is created, as in step (2) of RM-Write. It is more efficient to write log entries into a log buffer in volatile storage. When the buffer fills up, the log buffer is appended to the stable copy of the log. This significantly reduces the number of Writes to the (stable) log, but jeopardizes the Undo and Redo Rules.

It jeopardizes the Redo Rule because it is possible to commit a transaction before the after images of all of its updates are in the log. This problem is avoided almost automatically. Recall that commit list entries are stored in the log. Therefore, to add T_i to the commit list in step (1) of RM-Commit, we simply add T_i's commit list entry to the log buffer and flush the log buffer to the (stable) log. Since T_i's commit list entry follows all of the log entries describing its Writes, this ensures that the after images of all of its Writes are in the log before it commits, thereby satisfying the Redo Rule. Of course, the log buffer may not be full at the time a transaction commits, thereby sending a partially full buffer to stable storage. This partially defeats the reason for using the log buffer in the first place. It may therefore be worthwhile to use the *delayed commit* (sometimes called *group commit*) heuristic, which deliberately delays a requested Commit operation if it arrives when the last log buffer is largely empty. This delay provides some extra time for other transactions to write log entries to the log buffer before the Commit flushes it.

Buffering the log jeopardizes the Undo Rule because it is illegal to write an uncommitted update to the stable database before its before image is in the (stable) log. To solve this problem, it is helpful to identify each log entry by its *log address* or *log sequence number* (*LSN*), and to add to each cache slot another field containing an LSN. After RM-Write(T_i, x, v) inserts a log entry for x in the log buffer and updates x's cache slot, and before it unpins that slot, it writes the LSN of that entry into the slot's LSN. Before the CM flushes a slot, it ensures that all log entries up to and including the one whose LSN equals the cache slot's LSN have been appended to the log. Only then may it flush the slot. Since the slot is flushed only after its log entries are written to stable storage, the Undo Rule is satisfied.

Notice that step (3) of RM-Write(T_i, x, v) writes the log record to stable storage before x's cache slot is updated. This is sooner than necessary. By using the LSN mechanism just described, or by simply keeping track of the order in which certain pages must be written, we can avoid forcing this log record to be written to stable storage until x's cache slot is flushed.

We now describe the contents of each of the four types of log records in detail.

1. *Update:* This type of record documents a Write operation of a transaction. It contains the following information:
   - the name of the transaction that issued the Write;
   - the name or stable database location of the data item written;
   - the offset and length of the portion of the data item that was updated;
   - the old value of the portion of the data item that was updated (its before image);[6]
   - the new value of the portion of the data item that was updated (its after image); and
   - a pointer to (i.e., the LSN of) the previous update record of the same transaction (Null if this is the first update of the transaction); this can be easily found by maintaining a pointer to the last update record of each active transaction.

   This record is inserted in the log by step (3) of RM-Write.
2. *Commit:* This type of record says that a transaction has committed, and simply contains the name of the transaction. It is appended to the log by step (1) of RM-Commit.
3. *Abort:* This record says that a transaction has aborted and contains the name of that transaction. It is appended to the log by step (2) of RM-Abort.
4. *Checkpoint:* This type of record documents the completion of a checkpoint. It contains the following information:
   - a list of the active transactions at the time of the checkpoint; and
   - a list of the data items that were in dirty cache slots, along with the "stable-LSNs" of these slots, at the time the checkpoint was taken.

The *stable-LSN* is an additional field of information that we associate with each cache slot. It is the LSN of the last record in the log buffer at the time the data item presently occupying the slot was last fetched or flushed. The stable-LSN of a cache slot storing x marks a point in the log where it is known that the value of the stable database copy of x reflects (at least) all of the log records up to that point.

[6]In the previous abstract description of logs, update records did *not* contain the before image of the updated data item. It turns out that keeping such information in these records greatly facilitates the processing of RM-Abort and Restart.

The checkpoint record is inserted in the log by the Checkpoint procedure, shown below.

Checkpoint

1. Stop the RM from processing any more Read, Write, Commit, and Abort operations, and wait until it finishes processing all such operations that are in progress.
2. Flush each dirty cache slot that has not been flushed since the previous checkpoint. To achieve this, scan the cache and flush any dirty slot whose stable-LSN is smaller than the LSN of the previous checkpoint record (which can be stored in a special location in main memory for convenience), and update the stable-LSN of the slot accordingly.
3. Create a checkpoint record containing the relevant information (see the previous description of checkpoint records) and append this record to the log.
4. Acknowledge the processing of Checkpoint, thereby allowing the RM to resume processing operations.

The Checkpoint procedure is invoked periodically by the RM itself. For example, the RM may invoke it whenever the number of update records inserted in the log since the previous checkpoint exceeds a certain amount. As we mentioned earlier, increasing the frequency of checkpoints decreases the work of Restart and therefore decreases the recovery time after a system failure.

Given this structure for the log, the implementation of RM-Read, RM-Write, RM-Commit, and RM-Abort follows the outline given earlier in the section. Note how efficiently RM-Abort can be carried out. Since all of a transaction's update records are linked together, only the relevant log records need to be considered. Because such links may have to be followed, it is much better to keep the log on disk or another direct access device, rather than on tape. And since before images are included in the update records, no additional searching of the log is needed.

Restart

Restart processes the log in two scans: a backwards scan during which it undoes updates of uncommitted transactions, followed by a forward scan during which it redoes updates of committed transactions. We will first describe a simple version of the algorithm, after which we will look at potential optimizations.

The backwards scan begins at the end of the log (see Exercise 6.11). During this scan, Restart builds lists of committed and aborted transactions, denoted *CL* and *AL* (respectively). When it reads a commit or abort record, it adds the transaction to the appropriate list. When it reads an update record of some transaction T_i for data item x, it performs the following steps:

1. If T_i is in *CL*, it ignores the update record.
2. If T_i is in neither *CL* nor *AL*, then T_i was active at the time of failure, so it adds T_i to *AL*.
3. If T_i is (now) in *AL*, then it fetches x if it is not already in cache, and restores the before image of the portion of x recorded in the update record. Moreover, if the update record has no predecessor (i.e., this is the first log record of T_i), then T_i is removed from *AL* (since there's nothing more of T_i to be undone).

Restart ignores the last checkpoint record. When it reads the penultimate checkpoint record, Restart examines the list of active transactions stored in that record and adds to *AL* any of those transactions that are not in *AL* or *CL*. These are transactions that were active at the time of the penultimate Checkpoint and didn't commit or abort (or perform any updates) ever since. Thus, they were active at the time of the failure and should be aborted.

From the penultimate checkpoint record, Restart continues its backward scan of the log, ignoring all records except update records of transactions in *AL*. These are processed as in step (3) just given. The backward scan terminates when *AL* is left empty (recall that when the first update record of some transaction T_i in *AL* is processed, T_i is removed from *AL*).

To understand the effect of the backward scan, consider a single byte b of some data item. Let U be the last update record in the log that reflects an update to b, and whose transaction committed before the failure. By definition, U's after image defines b's last committed value relative to the execution at the time of the failure. We claim that if U precedes the penultimate checkpoint record, then b's value is contained in U's after image at the conclusion of the backward scan. To see this, first observe that since U precedes the penultimate checkpoint record, its after image must have been written to the stable database before the failure. Let V be any update record that reflects an update to b and follows U in the log. Let T_V be the transaction corresponding to V. By definition of U, T_V did not commit before the failure. If its abort record precedes the penultimate checkpoint record, then V must have been undone in the stable database. If not, then Restart's backward scan of the log undid V's update. In either case, b contains its after image in U as claimed.

By the preceding argument, all that remains after the backward scan is to install the correct value in those bytes whose last committed value is defined by an update record that follows the penultimate checkpoint record. This is done by a forward scan of the log beginning at the penultimate checkpoint record. For each update record U whose transaction is in *CL*, the corresponding data item is fetched if it is not already in cache, and U's after image is written into the cache slot. Update records of transactions not in *CL* are ignored. The scan terminates when it reaches the end of the log, at which time the database (some of which is still dirty in cache) is in the committed state with respect to the log.

This Restart algorithm is idempotent. It makes no assumptions about the stable database state other than those implied by the checkpoint records. Thus, if it is interrupted by a system failure after having performed some of its undos and redos, it can still be reexecuted from the beginning after the failure.

With its two scans of the log and updates of many data items, Restart can be a time consuming process. If the system were to fail after Restart terminates but before a Checkpoint was executed, all of that work would have to be repeated after the second failure. Restart can protect itself against such a failure by performing two checkpoints after it terminates. In effect, this results in a commit consistent checkpoint.

Using the information in the last checkpoint record that tells which data items were in dirty cache slots, we can improve this Restart procedure and avoid undoing or redoing certain update records. More specifically, suppose that during the backward scan of the log, Restart reads an update record of transaction T_i for data item x, where T_i is in *AL*. Such an update record need not be undone if:

A1: T_i's abort record lies between the penultimate and last checkpoint records, but x is not among the data items occupying dirty slots at the time of the last checkpoint; or

A2: T_i's abort record lies between the penultimate and last checkpoint records, and x was in a dirty cache slot at the last checkpoint, but its stable-LSN (also stored in the checkpoint record) is greater than the LSN of T_i's abort record.

To see A1, observe that data item x's absence from a dirty cache slot at the last checkpoint means that x's cache slot was replaced after T_i aborted. Therefore the before image of T_i's update for x was restored in the stable database.

To see A2, recall that before T_i's abort record was written, RM-Abort restored in the cache the before image of T_i's update of x. The cache slot for x must have been replaced between this time and the last checkpoint. Otherwise the stable-LSN for x's cache slot at the last checkpoint would not have been greater than the LSN of T_i's abort record. Thus, the before image of T_i's update of x was restored in the stable database.

Similarly, suppose that during the forward scan of the log, Restart reads an update record of some transaction T_i in *CL* for data item x. Such a record need not be redone if:

C1: T_i's update record lies between the last two checkpoint records, but x is not in the list of data items that were in dirty cache slots at the time of the last checkpoint; or

C2: T_i's update record lies between the last two checkpoint records, x is in the list of data items in dirty cache slots at the last checkpoint, but its stable-LSN is greater than the LSN of the update record at hand.

The reasons why these conditions make it unnecessary to redo T_i's update of x are analogous to those of the corresponding conditions A1 and A2. However, there is a subtle difference: conditions C1 and C2 describe the position of an *update* record relative to other records, but the corresponding conditions A1 and A2 describe the position of an *abort* record relative to other records. The reason for the difference will become apparent if you attempt to carry out the arguments to justify C1 and C2.

Logical Logging

Even with partial data item logging, before and after images of data items may consume too much space. This will occur if each Write on a data item x modifies most of the contents of x. For example, suppose each data item is a page of a file. A user operation that inserts a new record, r, at the beginning of a page, p, may have to shift down the remaining contents of p to make room for r. From the RM's viewpoint, all of p is being written, so p's before and after image must be logged in an update record, even with partial data item logging.

Using logical logging, one could reduce the size of this two page update record by replacing it with a log record that says "insert record r on page p." This log record would be much smaller than one containing a before and after image of p. To interpret this log record after a failure, Restart can redo it by inserting r on p, or can undo it by deleting r from p, depending on whether the corresponding transaction committed or aborted.

To implement logical logging, we need to expand the RM's repertoire of update operations beyond the simple Write operation. The larger repertoire may include operations such as insert record, delete record, shift records within page, etc. For each update operation o, the RM must have a procedure that creates a log record for o, a procedure that redoes o based on what the log record says, and a procedure that undoes o based on what the log record says.

These procedures are then interpreted by Restart in much the same way that update records are interpreted. However, there is one important difference. When interpreting an update record in physical logging, we can restore a before or after image without worrying about the current state of the data item. With logical log records we must be more careful. Some undos or redos corresponding to logical log records may only be applicable to a data item when it is in exactly the same (logical) state as when the log record was created.

To see why this matters, consider the following example. Suppose that the logical log contains a record LR, "insert record r on page p." Suppose that p's cache slot that includes LR's insertion was not flushed to the stable database before the system failed, and that the transaction that issued the insertion is aborted by Restart. When scanning back through the log, Restart will undo LR. However, the procedure *undo*(LR) is operating on a copy of p that does not have record r stored in it. Unfortunately, *undo*(LR) may not be able to tell whether or not r is in p. If it tries to delete r anyway (i.e., undo the insertion), it may obliterate some other data in p, thereby corrupting p. Notice that this wouldn't be a problem if it were simply restoring a before image, since correctly restoring a before image does not depend on the current state of the data item.

One way to avoid this problem is to write undo and redo procedures that have no effect when applied to a data item that is already in the appropriate state. For example, *undo*(*LR*) should have no effect if p does not include *LR*'s update, and *redo*(*LR*) should have no effect if p already does include *LR*'s update.

A second way to avoid the problem is to keep a copy of the stable database state as of the last checkpoint. After a system failure, Restart works from the "checkpoint" copy instead of the current stable database. It undoes update records that precede the last checkpoint record and that were produced by transactions that were active at the last checkpoint and did not subsequently commit. Then it redoes update records that follow the last checkpoint and were produced by committed transactions. Given that the execution is strict, each undo and redo of a log record will be applied to the same database state as when the log record was created.

This technique is used in IBM's prototype database system, System R. In their algorithm, shadowing is used to define the stable database at each checkpoint. Two directories are maintained: D_{cur}, describing the current stable database state, and D_{ckpt}, describing the stable database state just after the execution of the Checkpoint procedure. To checkpoint, the cache slots are flushed and a copy of D_{cur} is saved as the new D_{ckpt}. Subsequent updates are written to new locations, pointed to by D_{cur}. The shadow copies pointed to by D_{ckpt} are not overwritten, and therefore are available to Restart in the event of a system failure.

A third way to avoid the problem is to store LSNs in *data items*. Each data item contains the LSN of the log record that describes the last update applied to that data item by an active or committed transaction. In practice, many databases are structured with header information attached to each data item (e.g., a page header). In such databases, the LSN would be a field of the header.

The LSN in a data item x is very helpful to Restart, because it tells exactly which updates in the log have been applied to x. All update records whose LSN is less than or equal to LSN(x) (i.e., the LSN stored in x) *have* been applied to x. All those with larger LSN have *not*. This information enables Restart to undo or redo an update record on x only if x is in the same state as when the update record was generated.

LSNs in data items also help Restart be more efficient. Since Restart can tell if an update has already been applied to the stable copy of a data item by examining that data item's LSN, it can avoid unnecessary undos and redos to that data item.

To help it maintain the correct value for the LSN in each data item, the RM uses a new field for each update record U for data item x. The field contains the LSN of the previous active or committed update record for x before U. This information is easy to obtain at the time U is produced, because it is simply the LSN in x just before U's update is performed. Thus, all of the updates to each data item are chained backward in the log through this field.

We will explain how the RM uses LSNs in data items by describing a modified version of the partial data item logging algorithm, presented earlier in the section. We call it the *LSN-based logging algorithm*. In this description, we assume logical logging where each update record describes an update to a single data item (where a data item is the unit of transfer to stable storage). We also assume that executions are strict and that fuzzy checkpointing is used, as described earlier in the Section.

To process an RM-Write on x, the RM creates an update record U. It stores the current LSN(x) in U, updates x, and assigns LSN(x) to be LSN(U) (i.e., U's address in the log). When undoing U in the event of an Abort, it reassigns LSN(x) to be the previous LSN(x) stored in U.[7]

Restart does two scans of the log: a backward scan for undo, and a forward scan for redo. During the backward scan, suppose Restart encounters a log record U reflecting an update to x by a transaction T_i that subsequently aborted. It therefore fetches x and examines LSN(x). There are three cases:

1. If LSN(x) = LSN(U), then Restart undoes U, assigning to LSN(x) the previous LSN, which is stored in U. Notice that LSN(x) = LSN(U) implies that x is in the same state as it was after U was first applied, so it is safe to undo the logical log record.
2. If LSN(x) < LSN(U), then x does not contain U's update. So Restart should not undo U. Notice that LSN(x) helps us avoid incorrectly undoing U in this case.
3. If LSN(x) > LSN(U), then x contains an update described in an update record V appearing after U in the log. Since V was already encountered in the backward scan and was not undone, the transaction that produced V must have committed. Since the execution is strict and U precedes V in the log, U was undone before V updated x. Thus, as in case (2), it would be incorrect for Restart to undo U, since x is not in the same state as when U was written.

The backward scan terminates after Restart has reached the penultimate checkpoint and has processed all update records from transactions that were active at the penultimate checkpoint and did not subsequently commit.

During the forward scan, Restart begins at the penultimate checkpoint and processes each update record U (on x, say) from a committed transaction.

[7]This requires that the execution is strict. See the later discussion on record level locking.

If LSN(x) < LSN(U), then it must be that LSN(x) is the LSN in of the previous update record for x, so Restart redoes U. If LSN(x) $\geq$ LSN(U), then Restart ignores U, because x either has the state originally produced by U (i.e., LSN(x) = LSN(U)) or x contains an update "later" than U's (i.e., LSN(x) > LSN(U)). The "later" update must have been done by a committed transaction, or else it would have been undone in the backward scan of the log.

This Restart algorithm not only uses LSNs in data items to ensure a data item is in the appropriate state before applying a logical log record, but it also avoids unnecessary undos and redos. For example, during Restart's undo scan of the log, in case (3) just given (LSN(x) > LSN(U)), x already contains a committed value that is "later" than the one that undo(U) would have restored (had it been correct to do so). The Restart algorithm for partial data item *physical* logging would have undone U anyway, and then would redo V (the update record that produced the current value of x) during the forward scan. Using LSNs, we save the unnecessary work of *undo*(U) and *redo*(V). Thus, the use of LSNs in data items is valuable even with partial data item physical logging (see Exercise 6.26).

Logical logging may be especially useful for logical operations that update more than one data item. For example, in a dynamic search structure, such as a B-tree or dynamic hash table, an insertion of a single record may cause a page (i.e., data item) to be split, resulting in updates to three or more pages. One can save considerable log space by only logging the insertion, and leaving it to the undo and redo procedures to update all of the relevant pages.

However, a problem arises if the system fails when the database does not contain all of the updates performed by a single logical operation. For example, suppose operation o updates data items x, y, and z. (Operation o could be an insertion into a B-tree, which causes half of node x to be moved into a new node, z, and causes x's parent, y, to be updated to include a pointer to z.) After logging o in log record LR, the RM updates x and y, which are written to the stable database. But before the RM updates z, the system fails. Now, x and y contain LR's LSN, but z contains an earlier LSN. A straightforward implementation of *undo*(LR) and *redo*(LR) may not be able to properly interpret this mixed state of x, y, and z. To avoid this problem, a separate update record should be produced for each data item that is modified by the logical update.

Record Level Locking

Suppose we use a 2PL scheduler that locks records, where many records are stored on each page, and a page is the unit of atomic transfer to stable storage. To gain the benefit of record level locking, executions cannot be strict at the level of pages. If they were, then two active transactions could not concurrently update two different records on the same page, in which case they might as well lock pages instead of records. The next best alternative is to have executions be strict at the level of records. That is, a transaction can only write into a record r if all previous transactions that wrote into r have either committed or aborted.

Since the execution is strict, we can use the undo/redo algorithm for physical data item logging. Alternatively, we can use the logical logging algorithm, provided we make an adjustment for the way LSNs are handled during undo. To illustrate the problem, suppose records r_1 and r_2 are stored on the same page p. Consider the following history, which represents an execution of transactions T_i, T_j, and T_k: $w_i[r_1]\ c_i\ w_j[r_1]\ w_k[r_2]\ c_k\ a_j$. As the RM processes this execution, it produces a log. In fact, since there are no Reads in the execution, the log has exactly one record for each operation in the history. It will be convenient, for the moment, to use the history notation as representation for the log (e.g., the log record describing T_i's update of database record r_1 will be denoted "$w_i[r_1]$," etc.).

Using the previous approach on this log, to process a_j we would undo $w_j[r_1]$ and install $w_i[r_1]$'s LSN on p (which is the last update to p before $w_j[r_1]$). But this is the wrong LSN, because p contains $w_k[r_2]$, which has a larger LSN. In fact, there is no LSN that one can install in p to represent the precise state of p.

One good way around this problem is to write a log record for each undo that is performed. In the example, the log would now be $w_i[r_1]\ c_i\ w_j[r_1]\ w_k[r_2]\ c_k$ $undo(w_j[r_1])$, where "$undo(w_j[r_1])$" is a log record that records the fact that $w_j[r_1]$ was undone. When performing the undo, the RM can install the LSN of $undo(w_j[r_1])$ in p, which correctly describes the state of p relative to the log. Since we can trust a data item's LSN to tell us the exact state of the data item relative to the log, we can use the Restart procedure that we just described for LSN-based logging (Exercise 6.23).

However, there are complications to consider. One is undoing and redoing log records that describe undos. For example, if the system fails after $undo(w_j[r_1])$ in the execution being discussed, then during the backward log scan Restart should undo the log record for $undo(w_j[r_1])$ and then undo the log record for $w_j[r_1]$. Another issue is the logging done by Restart itself. Suppose Restart logs the undos it performs during the backward scan. Then an execution of Restart that is interrupted by a system failure has lengthened the log, giving it more work to do when it is invoked again. In theory, it might never terminate, even if the intervals between successive failures grow monotonically (see Exercise 6.24).

A second approach is to store LSNs in records rather than pages. To undo $w_j[r_1]$ in the example under discussion now requires no special treatment; just assign to r_1's LSN the LSN of the previous update record reflecting a Write on r_1 (see Exercise 6.25). This method avoids the complications associated with logging undos, but incurs extra space overhead for an LSN per data item.

6.5 THE UNDO/NO-REDO ALGORITHM

In this section we present a DM algorithm that never requires redo. To achieve this, the algorithm records all of a transaction's updates in the stable database before that transaction commits. This can be achieved by a slight modification of the algorithm in Section 6.4. In fact, the RM-Write, RM-Read, and RM-Abort procedures are precisely the same. RM-Commit and Restart are outlined next.

RM-Commit(T_i)

1. For each data item x updated by T_i, if x is in the cache, flush the slot it occupies.
2. Add T_i to the commit list.(A)
3. Acknowledge the commitment of T_i to the scheduler.
4. Delete T_i from the active list.

Restart

1. Discard all cache slots.
2. Let *undone* := { }.
3. Start with the last entry in the log and scan backwards to the beginning. Repeat the following steps until either *undone* equals the set of data items in the database, or there are no more log entries to examine. For each log entry [T_i, x, v], if T_i is not in the commit list and $x \notin$ *undone*, then:
   - allocate a slot for x in the cache;
   - copy the before image of x wrt T_i into x's cache slot;

   undone := *undone* $\cup$ {x}.
4. For each T_i in the commit list, if T_i is in the active list, remove it from the active list.(B)
5. Acknowledge the completion of Restart to the scheduler.

Comments

A. [Step (2) of RM-Commit(T_i)] This is the action that declares a transaction committed. If the system fails before T_i is added to the commit list, Restart will consider T_i uncommitted and will abort it (see Restart).

B. [Step (2) of Restart] The system failure may have occurred after step (2) but before step (4) of RM-Commit(T_i). Thus it is possible for T_i to be found both in the active and the commit list. In this case, the transaction must be considered committed (see comment (A)).

This RM-Commit procedure is essentially the same as in the undo/redo algorithm of Section 6.4, with the addition of step (1) to ensure that all of a transaction's updates are in the stable database by the time the transaction commits. This means that redo is never required, resulting in a more efficient Restart procedure.

This algorithm satisfies the Undo Rule. The argument is the same as for the undo/redo algorithm. It also satisfies the Redo Rule for the trivial reason that it does not require redo, because all Writes are recorded in the stable database before the transaction that issued them can commit. Since the two rules are satisfied, Restart can always restore in the stable database the last committed value of each data item, using only information in stable storage. Also, Restart is idempotent. Since Restart does not alter the set of transactions it acts on, if it were interrupted by a system failure it would repeat exactly the same work.

Implementation

With minor modifications, the log structure that was described for partial data item logging can be used to implement the undo/no-redo algorithm. As before, the log is a sequential disk file consisting of update, commit, abort, and checkpoint records. The only difference is that update records need not include after images. Since this algorithm never requires redo, the information becomes useless.

To reflect the change in the RM-Commit procedure, all cache slots written by a transaction must be flushed before the commit record is appended to the log. One can view this as a limited kind of checkpoint. However, we still need checkpointing to ensure that restored before images of aborted transactions are eventually recorded in the stable database. It is possible to eliminate checkpoints altogether from this algorithm by appropriately modifying RM-Abort (see Exercise 6.27).

If transactions have random reference patterns, then few of the data items updated by a transaction are likely to be updated again before becoming candidates for replacement. Therefore, the work in flushing a data item at commit isn't wasted, although forcing it at commit may increase response time. However, for a hot spot data item, the required flush for every committed Write may create a heavy I/O load that would not be experienced using undo/redo.

Another way of implementing RM-Commit in this algorithm is to take a cache consistent checkpoint just before adding a transaction to the commit list (i.e., instead of step (1) of RM-Commit). This means checkpointing (by flushing *all* dirty cache slots) each time a transaction commits. This is a viable method if the cache is not too large and the database system is not designed to handle a very high rate of transactions.

The undo/no-redo algorithm can be integrated nicely with the multiversion concurrency control algorithm described in Section 5.4. All versions of a data item are chained together in a linked list in stable storage. The versions appear in the list in the order in which they were produced, from youngest to oldest. The RM-Abort procedure can arrange to exclude versions produced by aborted transactions from the chain. The head of the list is the most recently created version (possibly written by an active transaction). Since the multiple copies themselves contain the data item's before images, we do not need a log. Said differently, the multiple copies of all data items constitute the log, but they are not structured as a sequential file. To ensure no-redo, the RM must flush the cache slots containing versions produced by a transaction before that transaction commits. To undo versions written by active or aborted transactions that were flushed to stable storage before a system failure, Restart must be able to distinguish versions written by committed transactions from those written by aborted or active ones. This is accomplished by tagging each version by the (unique) transaction that produced it, as in Section 5.4, and by maintaining commit, abort, and active lists (see Exercise 6.28). This implementation is used in DBS products of Prime Computer.

An interesting variation of this scheme is to transfer the undo activity from Restart to RM-Read. That is, Restart does not perform any undos, leaving it with nothing to do at all. When RM-Read reads a data item, it checks whether the data item's tag is in the commit list. If so, it processes it normally. If not, then it discards (i.e., undoes) that version and tries reading the next older version instead. It continues reading older versions until it finds a committed one. This algorithm eliminates Restart activity at the cost of more expensive RM-Reads, a good trade-off if system failures are frequent.

6.6 THE NO-UNDO/REDO ALGORITHM

In this section we'll present another RM algorithm, one that may require redo but never requires undo. To avoid undo, we must avoid recording updates of uncommitted transactions in the stable database. For this reason, when a data item is written, the new value is not recorded in the cache at that time; this happens only after a transaction commits. Consequently, when a new value is recorded in the stable database as a result of a cache slot's being replaced or flushed, it is assuredly the value of a committed transaction and will never need to be undone.

The five RM procedures are outlined next.

RM-Write(T_i, x, v)

1. Append a [T_i, x, v] record to the log.
2. Acknowledge to the scheduler the processing of RM-Write(T_i, x, v).

RM-Read(T_i, x)

1. If T_i has previously written into x, then return to the scheduler the after image of x wrt T_i.[4]

```
2. Otherwise,
     ■ if $x$ is not in the cache, fetch it;
     ■ return to the scheduler the value in $x$'s cache slot.
RM-Commit($T_i$)
  1. Add $T_i$ to the commit list.(B)
  2. For each $x$ updated by $T_i$:
     ■ if $x$ is not in the cache, fetch it;
     ■ copy the after image of $x$ wrt $T_i$ into $x$'s cache slot.(A)
  3. Acknowledge the processing of RM-Commit($T_i$) to the scheduler.
RM-Abort($T_i$)
  1. Add $T_i$ to the abort list.(C)
  2. Acknowledge the processing of RM-Abort($T_i$) to the scheduler.
Restart
  1. Discard all cache slots.
  2. Let redone := {}.
  3. Start with the last entry in the log and scan backwards toward the
     beginning. Repeat the following steps until either redone equals the set
     of data items in the database, or there are no more log entries to exam-
     ine. For each log entry [$T_i$, $x$, $v$], if $T_i$ is in the commit list and $x \notin$
     redone, then
     ■ allocate a slot for $x$ in the cache;
     ■ copy $v$ into $x$'s cache slot;
     ■ redone := redone $\cup$ {$x$}.
  4. Acknowledge the processing of Restart to the scheduler.
```

Comments

A. [Step (1) of RM-Read(T_i, x), step (2) of RM-Commit(T_i)] The after image of x wrt T_i can be found in the log. It is inserted there by RM-Write.

B. [Step (1) of RM-Commit(T_i)] This is the action that declares a transaction committed.

C. [Step (1) of RM-Abort(T_i)] At the level we are describing things, the abort list is not needed (it is only mentioned in this step). In practice, information in that list might be used by the garbage collector to recycle log space that contains information pertaining to aborted transactions.

This algorithm satisfies the Redo Rule: Each value written by a transaction is recorded in the log by RM-Write and, by the Garbage Collection Rule, it cannot have been deleted by the time the transaction commits. The Undo Rule is also satisfied for the trivial reason that the algorithm does not require undo. Since both rules are satisfied, Restart will always find in stable storage all of the information it needs to restore the stable database to its committed state. Since Restart does not affect the set of committed transactions, if it were interrupted by a system failure it would repeat exactly the same work and hence is idempotent.

Implementation

We can implement this algorithm using a log structure similar to that employed for the previous two algorithms. The elimination of undo simplifies matters in several ways. First, there is no need to bother with abort records. Second, since undo is never required, the update records need not contain before images. Since finding the before images to be included in the update record may require an additional access to the stable database, this feature could be important. As usual, checkpointing must be used to limit the amount of time an update can stay in the cache before it is flushed. Any of the techniques described in Section 6.4 can be used.

Various other structures for the log have been proposed for the no-undo/redo algorithm. A common one is *intentions lists*. In effect, the log is organized as a collection of lists, one per transaction, which are kept in stable storage. T_i's list contains the after images (wrt T_i) of all data items updated by T_i. These updates are not applied to the stable database before the transaction commits. Thus, we can think of the list as containing the transaction's "intentions." If a transaction aborts, its intentions list is simply discarded. If a transaction commits, its list is marked accordingly and flushed to stable storage, and the updates contained therein are applied to the stable database one at a time. When this is completed, the intentions list is discarded. On Restart, all intentions lists are inspected. Those not marked as committed are simply discarded. The rest have their updates applied to the stable database and are then discarded as in the commit process. Note that a particular update could be redone more than once if Restart is interrupted by a system failure, but this causes no harm.

Although Writes are not applied to the database until the transaction commits, they may need to be read sooner than that. If a transaction writes x and subsequently reads it, then the Read must return the previously written value.[8] Since the transaction is not yet committed, the value of x is not in the

[8]This is same problem we encountered in Section 5.5, where we delayed the application of a transaction's Writes until it terminates. Since that multiversion concurrency control algorithm requires this mechanism for reading from the intentions list, it fits especially neatly with no-undo/redo recovery.

database or cache. So the RM must find the value in the intentions list (i.e., step (1) of RM-Read). Doing this efficiently takes some care. One way is to index the intentions list by data item name. On each RM-Read(T_i, x), the RM checks the index for an entry for x. If there is one, it returns the last intentions list value for x. Otherwise, it finds x in the database (i.e., step (2) of RM-Read).

Another way to solve the problem is by using shadowing; see Exercise 6.30.

6.7 THE NO-UNDO/NO-REDO ALGORITHM

To avoid redo, all of a transaction T_i's updates must be in the stable database by the time T_i is committed. To avoid undo, none of T_i's updates can be in the stable database before T_i is committed. Hence, to eliminate both undo and redo, all of T_i's updates must be recorded in the stable database in a single atomic operation, at the time T_i commits. The RM-Commit(T_i) procedure would have to be something like the following:

RM-Commit(T_i)

1. In a single atomic action:
   - For each data item x updated by T_i, write the after image of x wrt T_i in the stable database.
   - Insert T_i into the commit list.
2. Acknowledge to the scheduler the processing of RM-Commit(T_i).

Incredible as it may sound, such a procedure is realizable! The difficulty, of course, is to organize the data structures so that an atomic action — a single atomic Write to stable storage — has the entire effect of step (1) in RM-Commit. That is, it must indivisibly install all of a transaction's updates in the stable database and insert T_i into the commit list. It should do this without placing an unreasonable upper bound on the number of updates each transaction may perform.

We can attain these goals by using a form of shadowing. The location of each data item's last committed value is recorded in a directory, stored in stable storage, and possibly cached for fast access. There are also working directories that point to uncommitted versions of some data items. Together, these directories point to all of the before and after images that would ordinarily be stored in a log. We therefore do not maintain a log as a separate sequential file.

When a transaction T_i writes a data item x, a new version of x is created in stable storage. The working directory that defines the database state used by T_i is updated to point to this version. Conceptually, this new version is part of the log until T_i commits. When T_i commits, the directory that defines the committed database state is updated to point to the versions that T_i wrote. This makes the results of T_i's Writes become part of the committed database state, thereby committing T_i.

With this structure, an RM-Commit procedure with the desired properties requires atomically changing the directory entries for *all* data items written by the transaction that is being committed. If the directory fits in a single data item, then the problem is solved. Otherwise, it seems we have simply moved our problem to a different structure. Instead of atomically installing updates in the stable database, we now have to atomically install updates in the directory.

The critical difference is that since the directory is much smaller than the database, it is feasible to keep two copies of it in stable storage: a *current* directory, pointing to the committed database, and a *scratch* copy. To commit a transaction T_i, the RM updates the scratch directory to represent the stable database state that includes T_i's updates. That is, for each data item x that T_i updates, the RM makes the scratch directory's entry for x point to T_i's new version of x. For data items that T_i did not update, it makes the scratch directory's entries identical to the current directory's entries. Then it swaps the current and scratch directories in an atomic action. This atomic swap action is implemented through a *master record* in stable storage, which has a bit indicating which of the two directory copies is the current one. To swap the directories, the RM simply complements the bit in the master record, which is surely an atomic action! Writing that bit is the operation that commits the transaction. Notice that the RM can only process one Commit at a time. That is, the activity of updating the scratch directory followed by complementing the master record bit is a critical section.

Figure 6–4 illustrates the structures used in the algorithm to commit transaction T_i which updated data items x and y. In Fig. 6–4(a) the transaction has created two new versions, leaving the old versions intact as shadows (appropriately shaded). In Fig. 6–4(b) T_i has set up the scratch directory to reflect the stable database as it should be after its commitment. In Fig. 6–4(c) the master record's bit is flipped, thereby committing T_i and installing its updates in the stable database. Note that there are two levels of indirection to obtain the current value of a data item. First the master record indicates the appropriate directory, and then the directory gives the data item's address in the stable database.

Before describing the five RM procedures, let us define some notation for the stable storage organization used in this algorithm. We have a master record, M, that stores a single bit. We have two directories D^0 and D^1. At any time D^b is the current directory, where b is the present value of M. $D^b[x]$ denotes the entry for data item x in directory D^b. It contains x's address in the stable database. We use $-b$ to denote the complement of b, so D^{-b} is the scratch directory. There may be one or two versions of a data item at any given time: one in the stable database (pointed to by D^b and possibly a new version. All this information — the stable database, the new versions, the two directories, and the master record — must be kept in stable storage. The master record and the directories can also be cached for efficient access.

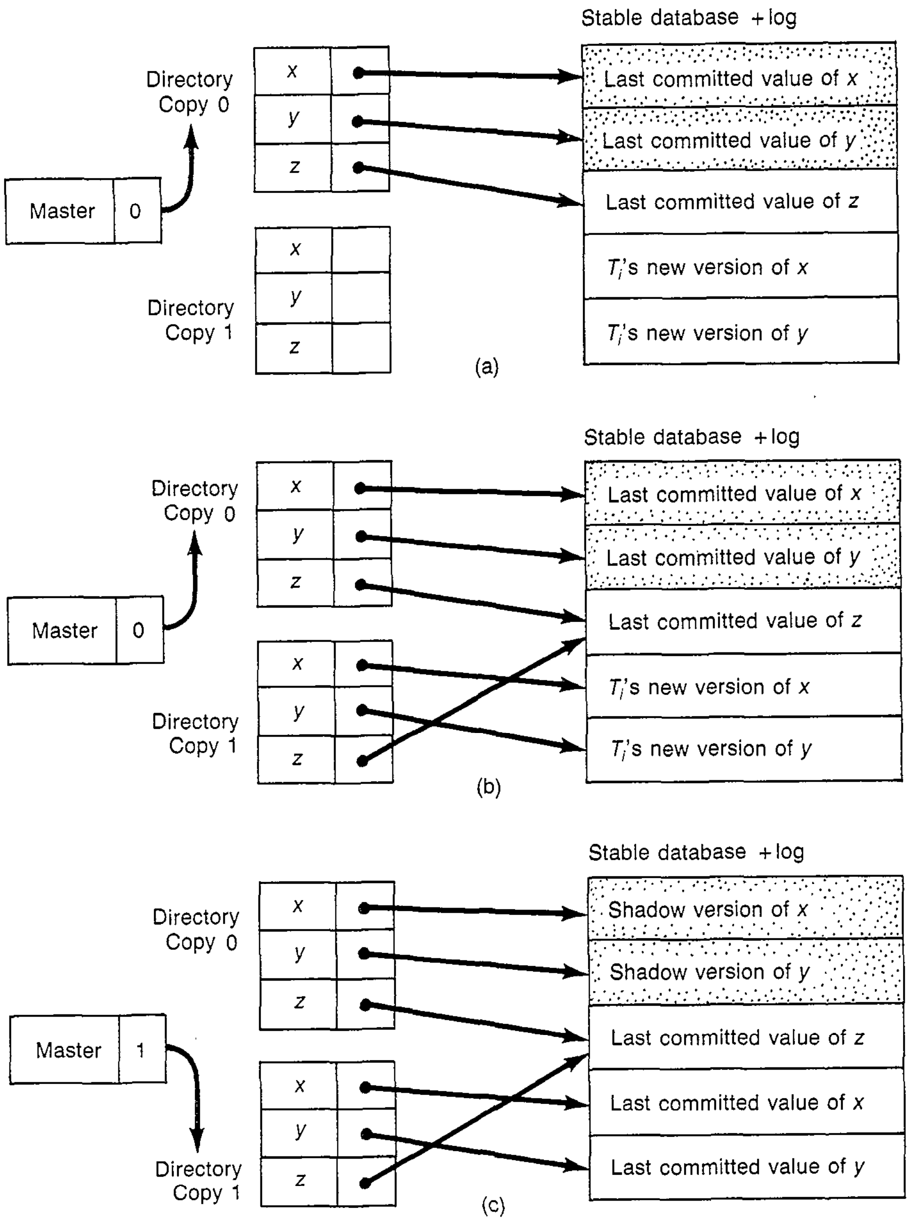

FIGURE 6–4
An Example of the No-Undo/No-Redo Algorithm
(a) Database state after creating new versions for T_i (b) Database state after preparing directory for T_i's commitment (c) Database state after committing T_i

In addition, for each active transaction T_i there is a directory D_i with the addresses of the new versions of the data items written by T_i. $D_i[x]$ denotes the entry of D_i that corresponds to data item x (presumably T_i wrote into x). These directories need not be stored in stable storage. Given this organization of data, the RM procedures are as follows.

RM-Write(T_i, x, v)

1. Write v into an unused location in stable storage and record this location's address in $D_i[x]$.(A)
2. Acknowledge to the scheduler the processing of RM-Write(T_i, x, v).

RM-Read(T_i, x)

1. If T_i has previously written into x, return to the scheduler the value stored in the location pointed to by $D_i[x]$.
2. Otherwise, return to the scheduler the value stored in the location pointed to by $D^b[x]$, where b is the present value of the bit in the master record M.(B)

RM-Commit(T_i)

1. For each x updated by T_i: $D^{-b}[x] := D_i[x\ ]$, where b is the value of M.(C)
2. $M := -b$.(D)
3. For each x updated by T_i: $D^{-b}[x] := D_i[x]$, where b is the (new) value of M.(E)
4. Discard D_i (free any storage used by it).
5. Acknowledge to the scheduler the processing of RM-Commit(T_i).

RM-Abort(T_i)

1. Discard D_i.
2. Acknowledge to the scheduler the processing of RM-Abort(T_i).

Restart

1. Copy D^b into D^{-b}.
2. Free any storage reserved for active transactions' directories and their new versions.
3. Acknowledge to the scheduler the processing of Restart.

Comments

A. [Step (1) of RM-Write] This step creates the new version of x, leaving the shadow version in the stable database untouched.

B. [Step (2) of RM-Read] If T_i has written into x it reads the new version of x that it created when it wrote into x (see step (1) of RM-Write); otherwise it reads the version of x in the stable database.

C. [Step (1) of RM-Commit] This step sets up the scratch directory to reflect the updates of T_i.

D. [Step (2) of RM-Commit] This step complements the bit in the master record, thereby making the scratch directory into the current one (and what used to be the current into the scratch). It is the atomic action that makes T_i committed. Failure before this step will result in T_i's abortion.

E. [Step (3) of RM-Commit] This step records T_i's changes in D^{-b}, which has now become the scratch directory. This ensures that when that directory again becomes the current one, T_i's updates will be properly reflected in the stable database.

The algorithm satisfies the Undo Rule since the stable database never has values written by uncommitted transactions. It also satisfies the Redo Rule because at the time of commitment all of a transaction's updates are in the stable database. In fact, under this algorithm the stable database *always* contains the last committed database state. As a result, virtually no work is needed to abort a transaction or restart the system following a failure.

While Restart is efficient, this algorithm does have three important costs during normal operation. First, accesses to stable storage are indirect and therefore more expensive. However, this cost may be small if the directory is small enough to be stored in cache. Second, finding uncommitted versions and reclaiming their space may be difficult to do efficiently, given the absence of a log. Third, and most importantly, the movement of data to new versions destroys the original layout of the stable database. That is, when a data item is updated, there may not be space for the new copy close to the original data item's (i.e., its shadow's) location. When the update is committed, the data item has changed location from the shadow to the new version. Thus, if the database is designed so that related data items are stored in nearby stable storage locations, that design will be compromised over time as some of those data items are updated. For example, if records of a file are originally stored contiguously on disk for efficient sequential access, they will eventually be spread into other locations thereby slowing down sequential access. This problem is common to many implementations of shadowing.

Because of the organization of the log, this algorithm is also known as the *shadow version algorithm*. And because of the way in which it commits a transaction, by atomically recording all of a transaction's updates in the stable database, it has also been called the *careful replacement algorithm*.

6.8 MEDIA FAILURES

As with recovery from system failures, the goal of recovery from media failures is to ensure the system's ability to reconstruct the last committed value of each data item. The difference is in the availability of memory we can rely on. In a system failure, we could rely on the availability of all the information the DM had been wise enough to save in stable storage. In a media failure, recovery cannot proceed on this assumption, since by definition such failures destroy part of stable storage — perhaps exactly the part needed by Restart.

Our only recourse is to maintain redundant copies of every data item's last committed value. This provides protection to the extent it can be assured that one of the redundant copies will survive a media failure. Thus the recovery mechanism for media failures must be designed with a probabilistic goal in mind: minimize the probability that all copies of a data item's last committed value are destroyed. By contrast, the recovery mechanism for system failures is designed with an absolute goal: guarantee the availability of the last committed value of each data item *in stable storage*. Of course, this only provides absolute protection against *system* failures, in which stable storage survives by definition.

Resiliency to media failure can be increased by increasing the number of copies kept. But at least as important as the *number* of copies is *where* these copies are stored. We want the copies to be kept on devices with *independent failure modes*. This means that no single failure event can destroy more than one copy. To achieve, or rather to approximate, independence of failures requires a detailed knowledge of the kinds of events that are likely to cause media failures in a particular system. Thus, keeping two copies on different disk drives is better than two copies on the same drive; since disk drives tend to fail independently, copies on two different drives may survive a single failure while copies on the same drive may not. Using disk drives with different controllers helps tolerate a controller's failure. Keeping the disk drives in separate rooms enhances the probability of the drives' surviving a fire. The perceived probability of such failures and the cost of minimizing their impact will determine how far the designer cares to go in protecting different media from common failure modes.

In practice, keeping two copies of the last committed value of each data item on two different devices is deemed to be sufficient protection for most applications. One approach, called *mirroring*, is to keep a duplicate copy of each disk on-line, in the form of a second disk. That is, the contents of each disk have an identical copy on another disk, its *mirror*. Since every data item is now stored on two disks, every Write must now be applied to two disks. Moreover, to ensure that the two disks are kept identical, Writes must be applied to both disks in the same order. To protect against the simultaneous failure of both disks, e.g., due to a power failure, it is safer to perform each pair of Writes in sequence: First write to one disk, wait for the acknowledgment, then write to the other disk. A Read can be sent to either disk that holds the desired data. So, mirrored disks increase the disks' capacity for Reads but not for Writes.

If a disk fails, then its mirror automatically handles all of the Reads that were formerly split between them. When the failed disk recovers or is replaced, it must be brought up-to-date by copying the contents of its up-to-date mirror. This can be done efficiently using the techniques for initializing a replicated database described in Chapter 8.[9]

Another approach to keeping duplicate copies is *archiving*. Periodically, the value of each data item is written (or *dumped*) to an *archive database*. The log contains (at least) all of the updates applied to the database since the last dump. Moreover, the log itself is mirrored. If a media failure destroys one of the log copies, the other one can be used. If a media failure destroys the contents of the stable database, then we execute a Restart algorithm using the log and archive database to bring the archive to the committed state with respect to the log.

This approach to media failures surely requires redo to bring the archive database to the committed state. Therefore, after images of Writes must be in the log. These after images are needed for media failures, even if we are using a no-redo algorithm for system failures (which ordinarily doesn't need after images). If the archive database contains uncommitted updates, then media failure recovery requires undo as well.

Producing an archive database is essentially the same as checkpointing the stable database. To distinguish them, we call the former *archive checkpointing* and the latter *stable checkpointing*. In each case, we are trying to avoid too much redo activity by making a copy of the database more up-to-date: the archive database for media failures, and the stable copy for system failures. For stable checkpointing, we (1) update the log to indicate which logged updates are in the stable database, and possibly (2) update the stable database to include updates that are only in cache. For archive checkpointing, we (1) update the log to indicate which logged updates are in the archive database, and possibly (2) update the archive database to include updates that are only in the stable database and cache. Thus, to modify a stable Checkpoint procedure to become an archive Checkpoint procedure, we simply substitute "the archive database" for "the stable database" and substitute "the stable database and the cache" for "the cache" in the definition of the procedure. Let's revisit the stable checkpointing algorithms of Section 6.4 with this correspondence in mind.

In commit consistent stable checkpointing, we complete the execution of all active transactions, flush the cache, and mark the log to indicate that the checkpoint has occurred. To modify this for archive checkpointing, we replace

[9]Briefly speaking, we can bring the mirror up-to-date by running "copier transactions." Copiers are synchronized like any user transaction (e.g., using 2PL). As soon as the recovering mirror is operational (i.e., before it is up-to-date), user transactions write into both disks. A data item copy in the recovering mirror cannot be read until it has been written at least once. See Sections 8.5 and 8.6.

"flush the cache" by "write to the archive all data items that were updated (in the stable database or cache) since the last archive checkpoint." To do this, for each data item in stable storage there must be a dirty bit that tells whether the data item has been updated in stable storage since it was last written to the archive database. (It needn't tell whether the data item has been updated in cache, since the cache slots' dirty bits give this information.) Checkpoint clears this bit after the data item is written to the archive (i.e., after the archive storage device has acknowledged the Write). After a media failure, Restart brings the archive to the committed state by redoing updates of transactions that committed after the last *archive checkpoint record* (i.e., the last checkpoint record written by archive checkpointing).

In cache consistent stable checkpointing, we complete the execution of all RM operations, flush the cache, and mark the log to indicate that the checkpoint has occurred. We modify this for archive checkpointing in exactly the same way we modified commit consistent checkpointing. The archive checkpoint record in the log should include a list of transactions that were active at the last checkpoint. After a media failure, Restart brings the archive to the committed state by redoing the update records that followed the last archive checkpoint record and were issued by committed transactions, and undoing transactions that were active at the last archive checkpoint and did not subsequently commit.

Cache consistent archive checkpointing is an improvement over commit consistent archive checkpointing in that the RM does not need to wait for all active transactions to terminate before initiating the checkpointing activity. However, commit consistent archive checkpointing provides a second line of defense if the stable database and both copies of the log are destroyed. We can at least restore a database state that reflects a consistent point in the past, merely by loading the archive database as the stable database. In some applications, this is an acceptable compromise.

The problem with both of these algorithms is the delay involved in bringing the archive up-to-date during the checkpoint procedure. This is a much bigger performance problem than flushing the cache in stable checkpointing, for two reasons. First, there is more data to write to the archive — all data items that were written since the previous archive checkpoint, even if they are in the stable database. And second, much of the data to be written to the archive database must first be read from the stable database. This delay is intolerable for many applications.

One way to avoid the delay is to use shadowing. When archive checkpointing begins, an *archive shadow directory* is created, which defines the state of the database at the time checkpointing began. An archive checkpoint record is written to the log to indicate that Checkpointing has begun. Now the RM can resume normal operation in parallel with checkpointing. When the RM processes an update to the stable database, it leaves intact the shadow page which is still referenced by the archive shadow directory. When the checkpointing procedure terminates, it writes another archive checkpoint record in the log, which says that all of the log updates up to the *previous* archive checkpoint record are now in the archive database.

Another way to avoid the delay is to use a variation of fuzzy checkpointing. When archive checkpointing begins, it writes a begin-archive-checkpoint record that includes a list of transactions that are active at this time. It then reads from the stable database those data items that have been written since the previous begin-archive-checkpoint record and copies them to the archive database. The RM can process operations concurrently with this activity. When Checkpoint terminates, it writes an end-archive-checkpoint record, indicating that all log updates up to the previous begin-archive-checkpoint record are now in the archive database, and possibly some later ones as well. Restart for media failures can now function like the Restart algorithm described in Section 6.4 for partial data item physical logging. The only difference is in the interpretation of checkpoint records; the last matched pair of begin-archive-checkpoint/end-archive-checkpoint records should be regarded as the penultimate and last checkpoint records, respectively. Notice that the backward scan of the log for undo is only needed beginning with the second (i.e., the one closer to the end of the log) of the last pair of archive checkpoint records, because later update records in the log could not have been written to the archive during the last checkpointing activity.

Media failures frequently only corrupt a small portion of the database, such as a few cylinders of a disk, or only one of many disk packs. The Restart procedures we just described are designed to restore the *entire* archive database to its committed state. The performance of Restart for partial media failures can be improved substantially by designing it to restore a defined set of data items (see Exercise 6.34).

To reduce the software complexity of the RM, it is valuable to design archive and stable checkpointing so that exactly the same Restart procedure can be used for system and media failures, the only difference being the choice of stable or archive database and stable or archive log. The algorithms we described for commit consistent checkpointing and cache consistent checkpointing (with and without shadowing) have this property. The one we described for fuzzy checkpointing requires some modification to attain this property (see Exercise 6.35).

BIBLIOGRAPHIC NOTES

The undo-redo paradigm for understanding centralized DBS recovery grew from the early work of [Bjork 73], [Bjork, Davies 72], and [Davies 73], and from work on fault tolerant computing (e.g., see [Anderson, Lee 81], [Shrivastava 85], and [Siewiorek, Swarz 82]). The categorization of algorithms by their undo and redo characteristics was developed independently in the survey papers [Bernstein, Goodman, Hadzilacos 83] and [Haerder, Reuter 83]. Shadowing is discussed in [Lorie 77], [Reuter 80], [Verhofstad 77], and [Verhofstad 78]. The undo/redo model and its connection to recoverability is formalized in [Hadzilacos 83] and [Hadzilacos 86]. Strategies for cache management are described in [Effelsberg, Haerder 84].

In this chapter, we assumed strict executions throughout. Some aspects of cascading aborts are discussed in [Briatico, Ciuffoletti, Simoncini 84] and [Hadzilacos 82].

The undo/redo algorithm for partial data item physical logging is from [Gray 78]. A discussion of the many subtleties of logging algorithms appears in [Traiger 82]. Using LSNs in pages is described in [Lindsay 80]. Logging algorithms for particular DBSs are described in [Crus 84] (IBM's DB2), [Gray et al. 81a] (IBM's System R), [Ong 84] (Synapse), and [Peterson, Strickland 83] (IBM's IMS).

Undo/no-redo algorithms are described in [Chan et al. 82] (CCA's Adaplex), [Dubourdieu 82] (Prime's DBSs), and [Rappaport 75] (where undo is performed by Reads, as described at the end of Section 6.5). Redo/no-undo algorithms are described in [Bayer 83], [Elhardt, Bayer 84], [Lampson, Sturgis 76], [Menasce, Landes 80], and [Stonebraker 79b] (the university version of INGRES). The no-undo/no-redo algorithm in Section 6.7 is from [Lorie 77].

The performance of recovery algorithms has been analyzed in [Garcia-Molina, Kent, Chung 85], [Griffeth, Miller 84], and [Reuter 84]. Recovery algorithms for database machines are discussed in [Agrawal, DeWitt 85a].

EXERCISES

6.1 Consider disk hardware that provides no checksum protection. We can partially compensate for this missing functionality by the following technique: Put a serial number in the first and last word of the block; put the disk address of the block in the second word of the block; and increment the serial number every time the block is written. What types of errors can we detect using this method? What types of errors are not detected? What types of CM and RM algorithms would best compensate for the weak error detection functionality?

6.2 Suppose that the disk hardware provides checksum protection on the address and data of each *sector* (i.e., each fixed length segment of a track). Suppose that the operating system offers Read and Write commands on pages, each page consisting of two sectors. What additional behavior of the disk hardware and operating system would you require (if any) so that the RM can view each Write of a page as an atomic operation? Suppose the operating system can report errors on a sector-by-sector basis. How might this information be used by the CM or RM to improve its performance?

6.3 Suppose each Write is either atomic or causes a media failure. Is it still useful for a Write to respond with a return code indicating whether it executed in its entirety or not at all, assuming that the return code is not reliable in the event of a media failure? Why?

6.4 Sketch the design of a cache manager for in-place updating, assuming page level granularity. In addition to supporting Fetch and Flush, your CM should give the RM the ability to specify the order in which any pair of pages is written to disk (e.g., to satisfy the undo and redo rules). You may assume that the CM will be used by an RM that does undo/redo logging. Remember that the RM will use the interface to read and write log pages as well as data item pages. Treat the replacement strategy as an uninterpreted procedure, but do define the interface to that procedure. What measures of performance did you use in selecting your design? What assumptions about application behavior did you make when deciding among alternative approaches? What alternatives did you reject for performance reasons?

6.5 Suppose the scheduler produces SR executions that avoid cascading aborts but are not strict. Suppose that the order of log records is consistent with the order in which the RM executed Writes. If $[T_i, x, u]$ precedes $[T_j, x, v]$ in the log, then what can you say about the order of Write, Commit, and Abort operations in the history that produced that log? What else can you say about the order of operations if you are given that T_j read x before writing it?

6.6 Since many transactions are writing entries to the end of the log concurrently, the RM must synchronize those log updates. This becomes a hot spot when the transaction rate becomes too high. Propose two mechanisms to minimize the effect of this hot spot. Describe the effects that each of the mechanisms has on Restart, if any.

6.7 Suppose we use an archive database and log to handle media recovery. Restate the Undo, Redo, and Garbage Collection Rules to properly reflect the combined requirements of system failures and media failures.

6.8 In cache consistent or fuzzy checkpointing, it is possible that a long running transaction prevents the RM from garbage collecting the log. How can this happen? One solution is to rewrite the log records of long running transactions at the end of the log. Suppose we take this approach in the partial data item logging algorithm. Checkpoint should find all log records that precede the penultimate checkpoint record and that are needed to handle transactions that are still active at the last checkpoint, and it should rewrite them at the end of the log. Restart should make appropriate use of these rewritten log records. Sketch the modifications needed for Checkpoint and Restart to perform these activities. Compare the cost and benefit of this algorithm with the approach in which Checkpoint aborts any transactions that are active at two consecutive checkpoints.

6.9 Consider the following variation on fuzzy checkpointing. Each execution of Checkpoint *initiates* flushes of all those cache slots that have not been flushed since before the previous checkpoint. Before writing its checkpoint record and allowing the RM to resume normal operation, it waits until all flushes initiated by the *previous* invocation of Checkpoint have completed. However, it does not wait until the flushes that *it* initiated have completed. Compare the performance of this fuzzy Checkpoint procedure with Restart algorithm described in the chapter. How does this new Checkpoint procedure affect the behavior of Restart for the partial data item logging algorithm?

6.10 Suppose that we allow the RM to continue to process operations while a cache consistent checkpointing procedure is in progress. Checkpoint must pin (or otherwise lock) each data item x while it is writing it to the stable database, to avoid interfering with RM-Writes to x. Under what conditions can Checkpoint write a checkpoint record in the log and terminate? Is Checkpoint in any danger of becoming involved in a deadlock or of being indefinitely postponed? If so, give a method for circumventing the problem.

6.11 When Restart begins executing after a system failure, it must find the end of the log. Propose a method for doing this. Remember that your method can only use information in stable storage for this purpose. What control information regarding the log is kept on stable storage? How often is it updated? Estimate the average time required for Restart to find the end of the log, e.g., measured in number of stable storage accesses.

6.12 Rewrite the Restart procedure for the partial data item logging algorithm in pseudo-code. Include the optimizations A1, A2, C1, and C2. Write a procedure to garbage collect the log, assuming that the log is not needed for media recovery. Explain why the algorithm produces the committed database state.

6.13 In the partial data item logging algorithm, it is unnecessary to undo any of T_i's updates if the LSN of T_i's abort record is less than the minimum stable-LSN over all dirty cache slots at the time of the last checkpoint. Similarly, it it is unnecessary to redo any update record with LSN less than the minimum stable-LSN over those cache slots. Explain why. Modify your solution to Exercise 6.12 to incorporate these optimizations.

6.14 Ordinarily, the Restart procedure for the partial data item logging algorithm would have to reexecute from the beginning if it were interrupted by a system failure. Propose a method for checkpointing the Restart procedure, to reduce the amount of work Restart has to repeat after a system failure. To reduce the chances of damaging the existing log, Restart should only append records to the log; it should not modify any previously existing log records.

6.15 Is the Restart procedure for the partial data item logging algorithm still correct under the assumption that the scheduler avoids cascading abort but is not strict? If so, explain why. If not, propose a way of circumventing the problem without strengthening the scheduling guarantees beyond ACA and SR.

6.16 Design a garbage collection procedure for the partial data item logging algorithm. The procedure should run concurrently with ordinary RM processing, and should not affect Restart's ability to recover from a system failure. How would you lay out the log on disk to avoid head contention between RM logging and the garbage collection algorithm?

6.17 The last step of the Restart procedure for the partial data item logging algorithm was to execute Checkpoint twice. Why is one checkpoint not enough? What is the benefit of doing only one checkpoint? Under what conditions, if any, might this be a good compromise?

6.18 In a tightly-coupled multiprocessor computer, the sequential Restart procedure for the partial data item logging algorithm is limited to execute on only one processor. Propose a modification to the algorithm and/or log structure to exploit the inherent parallelism of the computer.

6.19 Suppose we use an undo/redo algorithm that logs complete data items, where data items *don't* contain the LSN of the update record that last modified it. Given the large amount of log space that update records will consume, it is undesirable to log both the before and after image of each Write. Suppose we avoid this problem by logging just the after image, since the before image must be somewhere earlier in the log. Suppose we use commit consistent checkpointing.

Define a data structure for the log (update records, checkpoint records, etc.). Describe algorithms that use that data structure to implement the five RM procedures and Checkpoint. What problems would you have to solve to extend your algorithm to handle cache consistent checkpointing?

6.20 Suppose we use an undo/redo algorithm that logs complete data items, and uses fuzzy checkpointing. Suppose each update record in the log contains the (complete) before and after image of the data item updated, and a pointer to the previous update record of the same transaction. Assume that each checkpoint record contains lists of transactions that committed and aborted since the previous checkpoint, in addition to a list of active transactions at the time of the checkpoint and the stable-LSNs of dirty cache slots.

Write a Restart procedure that recovers from a system failure by doing a single backward scan of the log. Compare the working storage requirements of this algorithm with those of Restart in the partial data item logging algorithm. What modifications would you need to make to handle partial data item logging and still be able to perform Restart with a single log scan?

6.21 In logical logging, suppose each update record describes an operation that is applied to at most one data item. Suppose we implement undo and redo procedures for all operations so that for each log record *LR* on data item *x*, *undo*(*LR*) has no effect if *x* does not include *LR*'s update, and *redo*(*LR*) has no effect if *x* already includes *LR*'s update. Does Restart in the partial data item logging algorithm work correctly on a log with this structure? That is, assuming fuzzy checkpointing, is it correct to undo all uncommitted updates during a backward scan of the log, and then redo all committed updates during a forward scan? If so, argue the correctness of the algorithm. If not, explain why.

6.22 System R allows a single logical update record to cause an update to more than one data item. Work out the details of the System R RM algorithm described in Section 6.4 so that such multi-data-item updates are properly handled.

6.23 Revise the LSN-based logical logging algorithm for record locking so that it logs undos.

6.24 Suppose we use LSN-based logging and record locking, with logging of undos (see Exercise 6.23). If Restart logs undos during its backward scan of the log, and it is interrupted by a system failure, then it has more work to do on its next execution. It might not terminate, even if the inter-failure time grows with each execution. One approach is to have Restart periodically checkpoint its activity. Using this approach, or one of your own, sketch a method for ensuring that Restart always terminates, assuming that Restart logs undos and the interval between some two system failures is greater than some minimum value.

6.25 Revise the LSN-based logging algorithm so that it stores an LSN in every record rather than in every page (i.e., data item), and does not log undos.

6.26 The Restart procedure for the LSN-based logging algorithm saves unnecessary work by only undoing or redoing update records whose effect is not already in the stable copy of the data item. Characterize the set of data items that are read and written by Restart in the partial data item logging algorithm *without* LSNs in data items but are not read or written *with* LSNs in data items. What application and system parameters affect the size of this set (e.g., the number of dirty data items in cache at the time of failure, the number of log pages in between each pair of checkpoint records)? Derive a formula that estimates the size of this set as a function of these parameters.

6.27 In the undo/no-redo algorithm, checkpointing is needed because a data item containing an undone update may sit in cache for a long time without being flushed. Propose a modification to RM-Abort that avoids this problem, and therefore eliminates the need for checkpointing.

6.28 Describe the RM procedures for the undo/no-redo algorithm using multiversion concurrency control, as sketched in Section 6.5.

6.29 Suppose the entire database can fit in volatile storage. Design a no-undo/redo algorithm that logs partial data items and protects against system failures (i.e., no media recovery). Assume that the database is updated frequently, so that keeping the log small is important. Assume also that there is limited bandwidth to stable storage, so that a relatively small number of data items can be flushed per unit time. Notice that the only reason to flush data items is for checkpointing purposes. Propose a strategy for checkpointing that minimizes the time to recover from a system failure.

6.30 In the no-undo/redo algorithm, shadowing can be used to facilitate a transaction's reading data items that it previously wrote. When a transaction T_i first writes a data item x, a new version of x is created in stable storage and is written into a cache slot. The previous (i.e., shadow) version of x is not overwritten. Using this approach, each RM-Read from T_i can be processed normally on the database state that includes T_i's updates. Work out the details of this approach by providing algorithms for the five RM procedures.

6.31 Is it possible to modify the no-undo/no-redo algorithm to use record locking? If so, explain how. What are the performance implications? If not, explain why.

6.32 To compress the archive log, it is helpful to use a checkpointing method that allows most before images and abort records to be dropped from the log. Design such an archive checkpointing strategy for the partial data item logging algorithm. Propose an efficient algorithm for compressing the stable log before moving it to the archive.

6.33 Explain why each of the archive recovery methods proposed in Section 6.8 is idempotent.

6.34 Since only a small portion of the database may be affected by a media failure, it is helpful if the archive Restart procedure is able to recover a given set of data items, rather than the whole database. Suppose we use LSN-based logging and fuzzy archive checkpointing. Design an efficient algorithm that can recover individual data items independently. Notice the problem with redo; update records on the same data item do not normally have forward pointers, so it is hard to avoid a complete scan of the log. Consider the possibility of using an extra forward pointer field in each update record, which is filled in by Restart during the backward scan of the log, in order to speed up the redo scan by following those pointers.

6.35 Explain why the procedure outlined in Section 6.8 for fuzzy archive checkpointing leads to a different Restart procedure than the one we used for fuzzy stable checkpointing in the partial data item logging algorithm. Propose modifications to the checkpointing algorithms so that their Restart algorithms can be identical (except for the choice of stable or archive database and log as input).