

Database Recovery Techniques

In this chapter, we discuss some of the techniques that can be used for database recovery in case of system failure. In Section 20.1.4 we discussed the different causes of failure, such as system crashes and transaction errors. In Section 20.2, we introduced some of the concepts that are used by recovery processes, such as the system log and commit points.

This chapter presents additional concepts that are relevant to recovery protocols and provides an overview of the various database recovery algorithms. We start in Section 22.1 with an outline of a typical recovery procedure and a categorization of recovery algorithms, and then we discuss several recovery concepts, including write-ahead logging, in-place versus shadow updates, and the process of rolling back (undoing) the effect of an incomplete or failed transaction. In Section 22.2, we present recovery techniques based on *deferred update*, also known as the NO-UNDO/REDO technique, where the data on disk is not updated until *after* a transaction commits. In Section 22.3, we discuss recovery techniques based on *immediate update*, where data can be updated on disk during transaction execution; these include the UNDO/REDO and UNDO/NO-REDO algorithms. In Section 22.4, we discuss the technique known as shadowing or shadow paging, which can be categorized as a NO-UNDO/NO-REDO algorithm. An example of a practical DBMS recovery scheme, called ARIES, is presented in Section 22.5. Recovery in multidatabases is briefly discussed in Section 22.6. Finally, techniques for recovery from catastrophic failure are discussed in Section 22.7. Section 22.8 summarizes the chapter.

Our emphasis is on conceptually describing several different approaches to recovery. For descriptions of recovery features in specific systems, the reader should consult the bibliographic notes at the end of the chapter and the online and printed user manuals for those systems. Recovery techniques are often intertwined with the concurrency control mechanisms. Certain recovery techniques are best used with

specific concurrency control methods. We will discuss recovery concepts independently of concurrency control mechanisms.

22.1 Recovery Concepts

22.1.1 Recovery Outline and Categorization of Recovery Algorithms

Recovery from transaction failures usually means that the database is *restored* to the most recent consistent state before the time of failure. To do this, the system must keep information about the changes that were applied to data items by the various transactions. This information is typically kept in the **system log**, as we discussed in Section 21.2.2. A typical strategy for recovery may be summarized informally as follows:

1. If there is extensive damage to a wide portion of the database due to catastrophic failure, such as a disk crash, the recovery method restores a past copy of the database that was *backed up* to archival storage (typically tape or other large capacity offline storage media) and reconstructs a more current state by reapplying or *redoing* the operations of committed transactions from the *backed-up* log, up to the time of failure.
2. When the database on disk is not physically damaged, and a noncatastrophic failure of types 1 through 4 in Section 21.1.4 has occurred, the recovery strategy is to identify any changes that may cause an inconsistency in the database. For example, a transaction that has updated some database items on disk but has not been committed needs to have its changes reversed by *undoing* its write operations. It may also be necessary to *redo* some operations in order to restore a consistent state of the database; for example, if a transaction has committed but some of its write operations have not yet been written to disk. For noncatastrophic failure, the recovery protocol does not need a complete archival copy of the database. Rather, the entries kept in the online system log on disk are analyzed to determine the appropriate actions for recovery.

Conceptually, we can distinguish two main policies for recovery from noncatastrophic transaction failures: deferred update and immediate update. The **deferred update** techniques do not physically update the database on disk until *after* a transaction commits; then the updates are recorded in the database. Before reaching commit, all transaction updates are recorded in the local transaction workspace or in the main memory buffers that the DBMS maintains (the DBMS main memory cache; see Section 20.2.4). Before commit, the updates are recorded persistently in the log file on disk, and then after commit, the updates are written to the database from the main memory buffers. If a transaction fails before reaching its commit point, it will not have changed the database on disk in any way, so UNDO is not needed. It may be necessary to REDO the effect of the operations of a committed

transaction from the log, because their effect may not yet have been recorded in the database on disk. Hence, deferred update is also known as the **NO-UNDO/REDO algorithm**. We discuss this technique in Section 22.2.

In the **immediate update** techniques, the database *may be updated* by some operations of a transaction *before* the transaction reaches its commit point. However, these operations must also be recorded in the log *on disk* by force-writing *before* they are applied to the database on disk, making recovery still possible. If a transaction fails after recording some changes in the database on disk but before reaching its commit point, the effect of its operations on the database must be undone; that is, the transaction must be rolled back. In the general case of immediate update, both *undo* and *redo* may be required during recovery. This technique, known as the **UNDO/REDO algorithm**, requires both operations during recovery and is used most often in practice. A variation of the algorithm where all updates are required to be recorded in the database on disk *before* a transaction commits requires *undo* only, so it is known as the **UNDO/NO-REDO algorithm**. We discuss these two techniques in Section 22.3.

The UNDO and REDO operations are required to be **idempotent**—that is, executing an operation multiple times is equivalent to executing it just once. In fact, the whole recovery process should be idempotent because if the system were to fail during the recovery process, the next recovery attempt might UNDO and REDO certain *write_item* operations that had already been executed during the first recovery process. The result of recovery from a system crash *during recovery* should be the same as the result of recovering *when there is no crash during recovery*!

22.1.2 Caching (Buffering) of Disk Blocks

The recovery process is often closely intertwined with operating system functions—in particular, the buffering of database disk pages in the DBMS main memory cache. Typically, multiple disk pages that include the data items to be updated are **cached** into main memory buffers and then updated in memory before being written back to disk. The caching of disk pages is traditionally an operating system function, but because of its importance to the efficiency of recovery procedures, it is handled by the DBMS by calling low-level operating systems routines (see Section 20.2.4).

In general, it is convenient to consider recovery in terms of the database disk pages (blocks). Typically a collection of in-memory buffers, called the **DBMS cache**, is kept under the control of the DBMS for the purpose of holding these buffers. A **directory** for the cache is used to keep track of which database items are in the buffers.¹ This can be a table of <Disk_page_address, Buffer_location, ... > entries. When the DBMS requests action on some item, first it checks the cache directory to determine whether the disk page containing the item is in the DBMS cache. If it is not,

¹This is somewhat similar to the concept of page tables used by the operating system.

the item must be located on disk, and the appropriate disk pages are copied into the cache. It may be necessary to **replace** (or **flush**) some of the cache buffers to make space available for the new item (see Section 20.2.4).

The entries in the DBMS cache directory hold additional information relevant to buffer management. Associated with each buffer in the cache is a **dirty bit**, which can be included in the directory entry to indicate whether or not the buffer has been modified. When a page is first read from the database disk into a cache buffer, a new entry is inserted in the cache directory with the new disk page address, and the dirty bit is set to 0 (zero). As soon as the buffer is modified, the dirty bit for the corresponding directory entry is set to 1 (one). Additional information, such as the transaction id(s) of the transaction(s) that modified the buffer, are also kept in the directory. When the buffer contents are replaced (flushed) from the cache, the contents must first be written back to the corresponding disk page *only if its dirty bit is 1*.

Another bit, called the **pin-unpin** bit, is also needed—a page in the cache is **pinned** (bit value 1 (one)) if it cannot be written back to disk as yet. For example, the recovery protocol may restrict certain buffer pages from being written back to the disk until the transactions that changed this buffer have committed.

Two main strategies can be employed when flushing a modified buffer back to disk. The first strategy, known as **in-place updating**, writes the buffer to the *same original disk location*, thus overwriting the old value of any changed data items on disk.² Hence, a single copy of each database disk block is maintained. The second strategy, known as **shadowing**, writes an updated buffer at a different disk location, so multiple versions of data items can be maintained, but this approach is not typically used in practice.

In general, the old value of the data item before updating is called the **before image (BFIM)**, and the new value after updating is called the **after image (AFIM)**. If shadowing is used, both the BFIM and the AFIM can be kept on disk; hence, it is not strictly necessary to maintain a log for recovering. We briefly discuss recovery based on shadowing in Section 22.4.

22.1.3 Write-Ahead Logging, Steal/No-Steal, and Force/No-Force

When in-place updating is used, it is necessary to use a log for recovery (see Section 21.2.2). In this case, the recovery mechanism must ensure that the BFIM of the data item is recorded in the appropriate log entry and that the log entry is flushed to disk before the BFIM is overwritten with the AFIM in the database on disk. This process is generally known as **write-ahead logging** and is necessary so we can UNDO the operation if this is required during recovery. Before we can describe a protocol for write-ahead logging, we need to distinguish between two types of log entry information included for a write command: the information needed for UNDO

²In-place updating is used in most systems in practice.

and the information needed for REDO. A **REDO-type log entry** includes the **new value** (AFIM) of the item written by the operation since this is needed to *redo* the effect of the operation from the log (by setting the item value in the database on disk to its AFIM). The **UNDO-type log entries** include the **old value** (BFIM) of the item since this is needed to *undo* the effect of the operation from the log (by setting the item value in the database back to its BFIM). In an UNDO/REDO algorithm, both BFIM and AFIM are recorded into a single log entry. Additionally, when cascading rollback (see Section 22.1.5) is possible, *read_item* entries in the log are considered to be UNDO-type entries.

As mentioned, the DBMS cache holds the cached database disk blocks in main memory buffers. The DBMS cache includes not only *data file blocks*, but also *index file blocks* and *log file blocks* from the disk. When a log record is written, it is stored in the current log buffer in the DBMS cache. The log is simply a sequential (append-only) disk file, and the DBMS cache may contain several log blocks in main memory buffers (typically, the last n log blocks of the log file). When an update to a data block—stored in the DBMS cache—is made, an associated log record is written to the last log buffer in the DBMS cache. With the write-ahead logging approach, the log buffers (blocks) that contain the associated log records for a particular data block update *must first be written to disk* before the data block itself can be written back to disk from its main memory buffer.

Standard DBMS recovery terminology includes the terms **steal/no-steal** and **force/no-force**, which specify the rules that govern *when* a page from the database cache can be written to disk:

1. If a cache buffer page updated by a transaction *cannot* be written to disk before the transaction commits, the recovery method is called a **no-steal approach**. The pin-unpin bit will be set to 1 (pin) to indicate that a cache buffer cannot be written back to disk. On the other hand, if the recovery protocol allows writing an updated buffer *before* the transaction commits, it is called **steal**. Steal is used when the DBMS cache (buffer) manager needs a buffer frame for another transaction and the buffer manager replaces an existing page that had been updated but whose transaction has not committed. The *no-steal rule* means that UNDO will never be needed during recovery, since a committed transaction will not have any of its updates on disk before it commits.
2. If all pages updated by a transaction are immediately written to disk *before* the transaction commits, the recovery approach is called a **force approach**. Otherwise, it is called **no-force**. The *force rule* means that REDO will never be needed during recovery, since any committed transaction will have all its updates on disk before it is committed.

The deferred update (NO-UNDO) recovery scheme discussed in Section 22.2 follows a *no-steal* approach. However, typical database systems employ a *steal/no-force* (UNDO/REDO) strategy. The *advantage of steal* is that it avoids the need for a very large buffer space to store all updated pages in memory. The *advantage of no-force* is that an updated page of a committed transaction may still be in the buffer when

another transaction needs to update it, thus eliminating the I/O cost to write that page multiple times to disk and possibly having to read it again from disk. This may provide a substantial saving in the number of disk I/O operations when a specific page is updated heavily by multiple transactions.

To permit recovery when in-place updating is used, the appropriate entries required for recovery must be permanently recorded in the log on disk before changes are applied to the database. For example, consider the following **write-ahead logging (WAL)** protocol for a recovery algorithm that requires both UNDO and REDO:

1. The before image of an item cannot be overwritten by its after image in the database on disk until all UNDO-type log entries for the updating transaction—up to this point—have been force-written to disk.
2. The commit operation of a transaction cannot be completed until all the REDO-type and UNDO-type log records for that transaction have been force-written to disk.

To facilitate the recovery process, the DBMS recovery subsystem may need to maintain a number of lists related to the transactions being processed in the system. These include a list for **active transactions** that have started but not committed as yet, and they may also include lists of all **committed** and **aborted transactions** since the last checkpoint (see the next section). Maintaining these lists makes the recovery process more efficient.

22.1.4 Checkpoints in the System Log and Fuzzy Checkpointing

Another type of entry in the log is called a **checkpoint**.³ A [checkpoint, *list of active transactions*] record is written into the log periodically at that point when the system writes out to the database on disk all DBMS buffers that have been modified. As a consequence of this, all transactions that have their [commit, *T*] entries in the log before a [checkpoint] entry do not need to have their WRITE operations *redone* in case of a system crash, since all their updates will be recorded in the database on disk during checkpointing. As part of checkpointing, the list of transaction ids for active transactions at the time of the checkpoint is included in the checkpoint record, so that these transactions can be easily identified during recovery.

The recovery manager of a DBMS must decide at what intervals to take a checkpoint. The interval may be measured in time—say, every *m* minutes—or in the number *t* of committed transactions since the last checkpoint, where the values of *m* or *t* are system parameters. Taking a checkpoint consists of the following actions:

1. Suspend execution of transactions temporarily.
2. Force-write all main memory buffers that have been modified to disk.

³The term *checkpoint* has been used to describe more restrictive situations in some systems, such as DB2. It has also been used in the literature to describe entirely different concepts.

3. Write a [checkpoint] record to the log, and force-write the log to disk.
4. Resume executing transactions.

As a consequence of step 2, a checkpoint record in the log may also include additional information, such as a list of active transaction ids, and the locations (addresses) of the first and most recent (last) records in the log for each active transaction. This can facilitate undoing transaction operations in the event that a transaction must be rolled back.

The time needed to force-write all modified memory buffers may delay transaction processing because of step 1, which is not acceptable in practice. To overcome this, it is common to use a technique called **fuzzy checkpointing**. In this technique, the system can resume transaction processing after a [begin_checkpoint] record is written to the log without having to wait for step 2 to finish. When step 2 is completed, an [end_checkpoint, ...] record is written in the log with the relevant information collected during checkpointing. However, until step 2 is completed, the previous checkpoint record should remain valid. To accomplish this, the system maintains a file on disk that contains a pointer to the valid checkpoint, which continues to point to the previous checkpoint record in the log. Once step 2 is concluded, that pointer is changed to point to the new checkpoint in the log.

22.1.5 Transaction Rollback and Cascading Rollback

If a transaction fails for whatever reason after updating the database, but before the transaction commits, it may be necessary to **roll back** the transaction. If any data item values have been changed by the transaction and written to the database on disk, they must be restored to their previous values (BFIMs). The undo-type log entries are used to restore the old values of data items that must be rolled back.

If a transaction T is rolled back, any transaction S that has, in the interim, read the value of some data item X written by T must also be rolled back. Similarly, once S is rolled back, any transaction R that has read the value of some data item Y written by S must also be rolled back; and so on. This phenomenon is called **cascading rollback**, and it can occur when the recovery protocol ensures *recoverable* schedules but does not ensure *strict* or *cascadeless* schedules (see Section 20.4.2). Understandably, cascading rollback can be complex and time-consuming. That is why almost all recovery mechanisms are designed so that cascading rollback *is never required*.

Figure 22.1 shows an example where cascading rollback is required. The read and write operations of three individual transactions are shown in Figure 22.1(a). Figure 22.1(b) shows the system log at the point of a system crash for a particular execution schedule of these transactions. The values of data items A , B , C , and D , which are used by the transactions, are shown to the right of the system log entries. We assume that the original item values, shown in the first line, are $A = 30$, $B = 15$, $C = 40$, and $D = 20$. At the point of system failure, transaction T_3 has not reached its conclusion and must be rolled back. The WRITE operations of T_3 , marked by a single * in Figure 22.1(b), are the T_3 operations that are undone during transaction rollback. Figure 22.1(c) graphically shows the operations of the different transactions along the time axis.

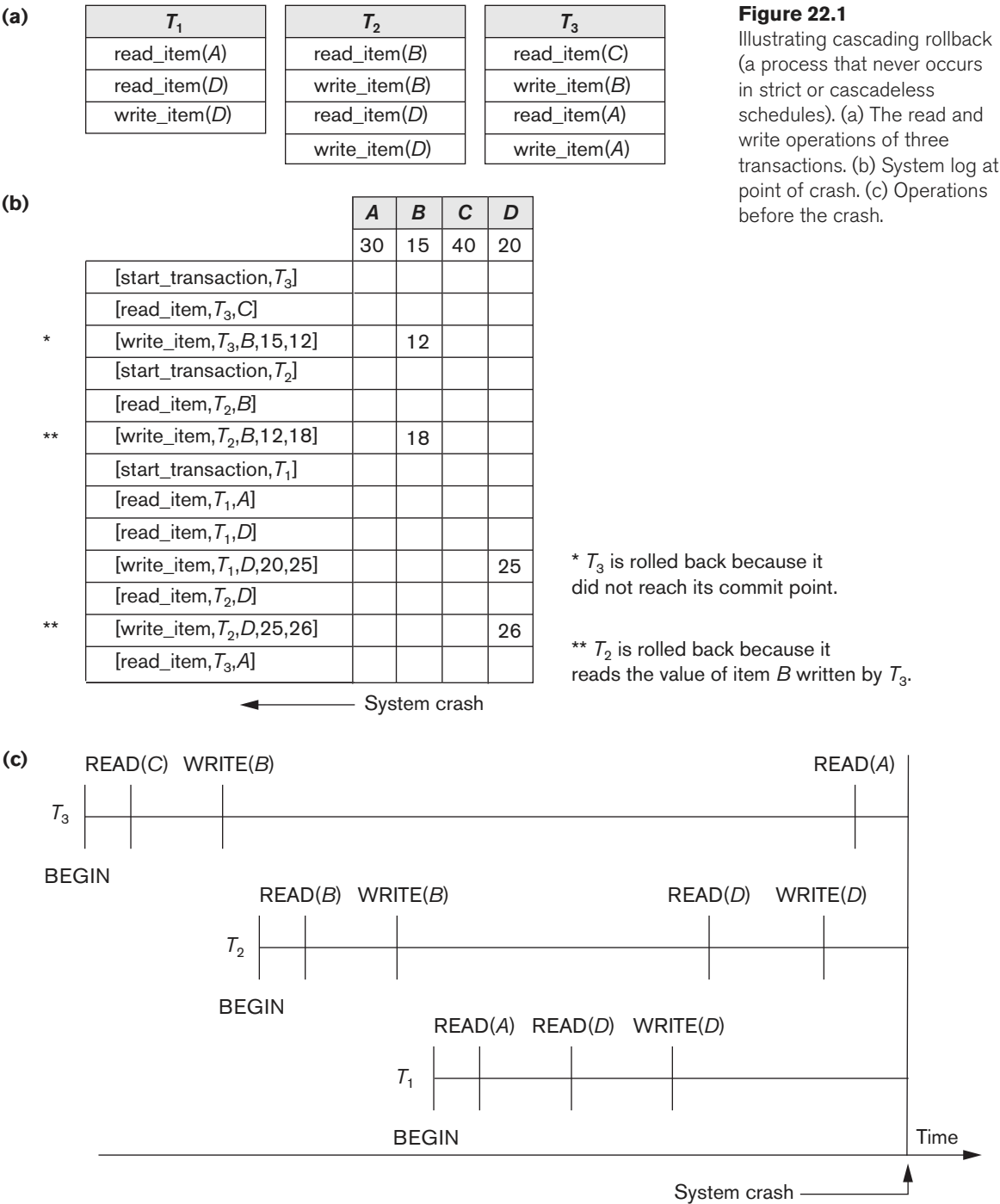


Figure 22.1
Illustrating cascading rollback (a process that never occurs in strict or cascadeless schedules). (a) The read and write operations of three transactions. (b) System log at point of crash. (c) Operations before the crash.

* T_3 is rolled back because it did not reach its commit point.

** T_2 is rolled back because it reads the value of item B written by T_3 .

We must now check for cascading rollback. From Figure 22.1(c), we see that transaction T_2 reads the value of item B that was written by transaction T_3 ; this can also be determined by examining the log. Because T_3 is rolled back, T_2 must now be rolled back, too. The WRITE operations of T_2 , marked by ** in the log, are the ones that are undone. Note that only write_item operations need to be undone during transaction rollback; read_item operations are recorded in the log only to determine whether cascading rollback of additional transactions is necessary.

In practice, cascading rollback of transactions is *never* required because practical recovery methods *guarantee cascadeless or strict* schedules. Hence, there is also no need to record any read_item operations in the log because these are needed only for determining cascading rollback.

22.1.6 Transaction Actions That Do Not Affect the Database

In general, a transaction will have actions that do *not* affect the database, such as generating and printing messages or reports from information retrieved from the database. If a transaction fails before completion, we may not want the user to get these reports, since the transaction has failed to complete. If such erroneous reports are produced, part of the recovery process would have to inform the user that these reports are wrong, since the user may take an action that is based on these reports and that affects the database. Hence, such reports should be generated only *after the transaction reaches its commit point*. A common method of dealing with such actions is to issue the commands that generate the reports but keep them as batch jobs, which are executed only after the transaction reaches its commit point. If the transaction fails, the batch jobs are canceled.

22.2 NO-UNDO/REDO Recovery Based on Deferred Update

The idea behind deferred update is to defer or postpone any actual updates to the database on disk until the transaction completes its execution successfully and reaches its commit point.⁴

During transaction execution, the updates are recorded only in the log and in the cache buffers. After the transaction reaches its commit point and the log is force-written to disk, the updates are recorded in the database. If a transaction fails before reaching its commit point, there is no need to undo any operations because the transaction has not affected the database on disk in any way. Therefore, only **REDO-type log entries** are needed in the log, which include the **new value** (AFIM) of the item written by a write operation. The **UNDO-type log entries** are not needed since no undoing of operations will be required during recovery. Although this may simplify the recovery process, it cannot be used in practice unless transactions are short and each transaction changes few items. For other types of transactions, there is the potential for running out of buffer space because transaction changes must be held

⁴Hence deferred update can generally be characterized as a *no-steal approach*.

in the cache buffers until the commit point, so many cache buffers will be *pinned* and cannot be replaced.

We can state a typical deferred update protocol as follows:

1. A transaction cannot change the database on disk until it reaches its commit point; hence all buffers that have been changed by the transaction must be pinned until the transaction commits (this corresponds to a *no-steal policy*).
2. A transaction does not reach its commit point until all its REDO-type log entries are recorded in the log *and* the log buffer is force-written to disk.

Notice that step 2 of this protocol is a restatement of the write-ahead logging (WAL) protocol. Because the database is never updated on disk until after the transaction commits, there is never a need to UNDO any operations. REDO is needed in case the system fails after a transaction commits but before all its changes are recorded in the database on disk. In this case, the transaction operations are redone from the log entries during recovery.

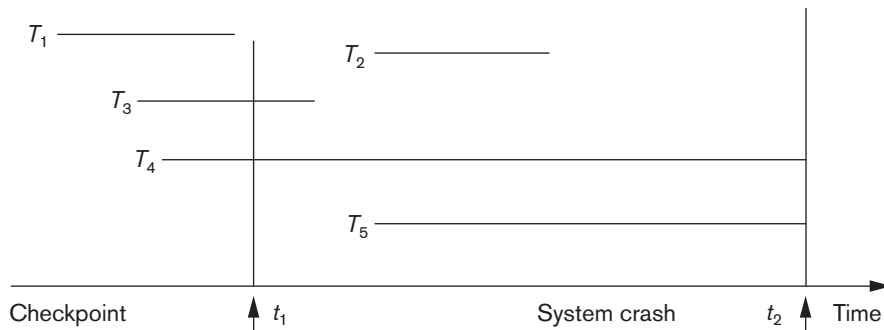
For multiuser systems with concurrency control, the concurrency control and recovery processes are interrelated. Consider a system in which concurrency control uses strict two-phase locking, so the locks on written items remain in effect *until the transaction reaches its commit point*. After that, the locks can be released. This ensures strict and serializable schedules. Assuming that [checkpoint] entries are included in the log, a possible recovery algorithm for this case, which we call RDU_M (Recovery using Deferred Update in a Multiuser environment), is given next.

Procedure RDU_M (NO-UNDO/REDO with checkpoints). Use two lists of transactions maintained by the system: the committed transactions T since the last checkpoint (**commit list**), and the active transactions T' (**active list**). REDO all the WRITE operations of the committed transactions from the log, *in the order in which they were written into the log*. The transactions that are active and did not commit are effectively canceled and must be resubmitted.

The REDO procedure is defined as follows:

Procedure REDO (WRITE_OP). Redoing a write_item operation WRITE_OP consists of examining its log entry [write_item, T , X , new_value] and setting the value of item X in the database to new_value, which is the after image (AFIM).

Figure 22.2 illustrates a timeline for a possible schedule of executing transactions. When the checkpoint was taken at time t_1 , transaction T_1 had committed, whereas transactions T_3 and T_4 had not. Before the system crash at time t_2 , T_3 and T_2 were committed but not T_4 and T_5 . According to the RDU_M method, there is no need to redo the write_item operations of transaction T_1 —or any transactions committed before the last checkpoint time t_1 . The write_item operations of T_2 and T_3 must be redone, however, because both transactions reached their commit points after the last checkpoint. Recall that the log is force-written before committing a transaction. Transactions T_4 and T_5 are ignored: They are effectively canceled or rolled back because none of their write_item operations were recorded in the database on disk under the deferred update protocol (no-steal policy).

**Figure 22.2**

An example of a recovery timeline to illustrate the effect of checkpointing.

We can make the NO-UNDO/REDO recovery algorithm *more efficient* by noting that, if a data item X has been updated—as indicated in the log entries—more than once by committed transactions since the last checkpoint, it is only necessary to REDO *the last update of X* from the log during recovery because the other updates would be overwritten by this last REDO. In this case, we start from *the end of the log*; then, whenever an item is redone, it is added to a list of redone items. Before REDO is applied to an item, the list is checked; if the item appears on the list, it is not redone again, since its latest value has already been recovered.

If a transaction is aborted for any reason (say, by the deadlock detection method), it is simply resubmitted, since it has not changed the database on disk. A drawback of the method described here is that it limits the concurrent execution of transactions because *all write-locked items remain locked until the transaction reaches its commit point*. Additionally, it may require excessive buffer space to hold all updated items until the transactions commit. The method's main benefit is that transaction operations *never need to be undone*, for two reasons:

1. A transaction does not record any changes in the database on disk until after it reaches its commit point—that is, until it completes its execution successfully. Hence, a transaction is never rolled back because of failure during transaction execution.
2. A transaction will never read the value of an item that is written by an uncommitted transaction, because items remain locked until a transaction reaches its commit point. Hence, no cascading rollback will occur.

Figure 22.3 shows an example of recovery for a multiuser system that utilizes the recovery and concurrency control method just described.

22.3 Recovery Techniques Based on Immediate Update

In these techniques, when a transaction issues an update command, the database on disk can be updated *immediately*, without any need to wait for the transaction to reach its commit point. Notice that it is *not a requirement* that every update be

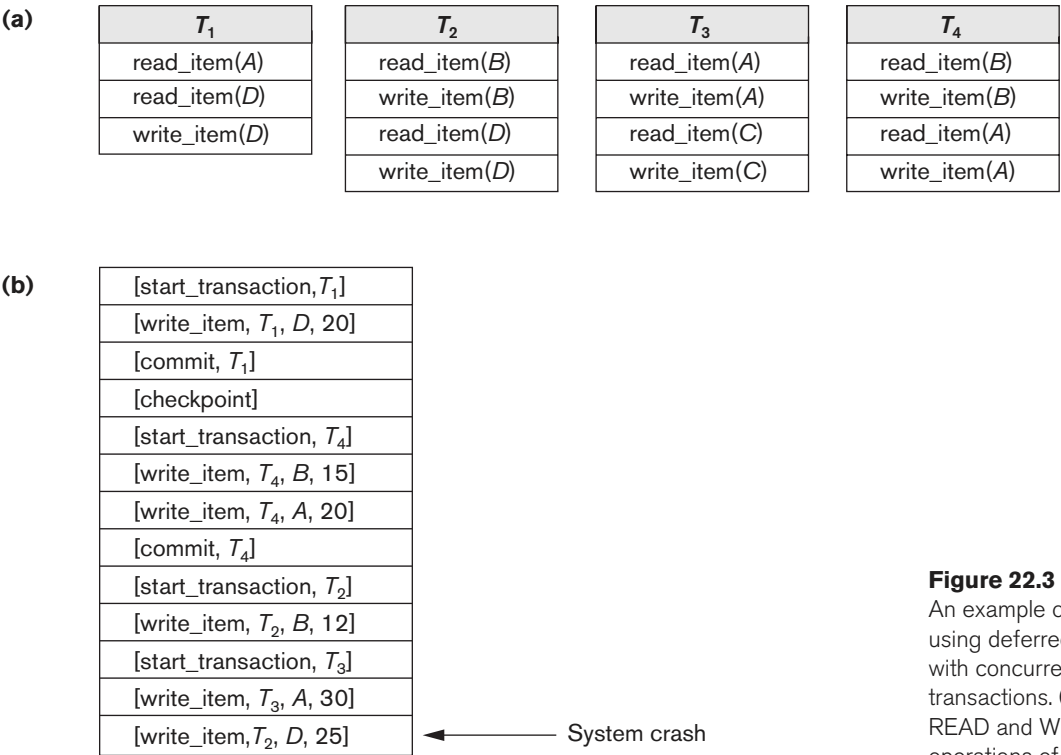


Figure 22.3
An example of recovery using deferred update with concurrent transactions. (a) The READ and WRITE operations of four transactions. (b) System log at the point of crash.

T_2 and T_3 are ignored because they did not reach their commit points.
 T_4 is redone because its commit point is after the last system checkpoint.

applied immediately to disk; it is just possible that some updates are applied to disk *before the transaction commits*.

Provisions must be made for *undoing* the effect of update operations that have been applied to the database by a *failed transaction*. This is accomplished by rolling back the transaction and undoing the effect of the transaction's write_item operations. Therefore, the **UNDO-type log entries**, which include the **old value** (BFIM) of the item, must be stored in the log. Because UNDO can be needed during recovery, these methods follow a **steal strategy** for deciding when updated main memory buffers can be written back to disk (see Section 22.1.3).

Theoretically, we can distinguish two main categories of immediate update algorithms.

1. If the recovery technique ensures that all updates of a transaction are recorded in the database on disk *before the transaction commits*, there is never a need to REDO any operations of committed transactions. This is called the **UNDO/NO-REDO recovery algorithm**. In this method, all updates by a transaction must be recorded on disk *before the transaction commits*, so that REDO is never needed. Hence, this method must utilize the **steal/force**

strategy for deciding when updated main memory buffers are written back to disk (see Section 22.1.3).

2. If the transaction is allowed to commit before all its changes are written to the database, we have the most general case, known as the **UNDO/REDO recovery algorithm**. In this case, the **steal/no-force strategy** is applied (see Section 22.1.3). This is also the most complex technique, but the most commonly used in practice. We will outline an UNDO/REDO recovery algorithm and leave it as an exercise for the reader to develop the UNDO/NO-REDO variation. In Section 22.5, we describe a more practical approach known as the ARIES recovery technique.

When concurrent execution is permitted, the recovery process again depends on the protocols used for concurrency control. The procedure RIU_M (Recovery using Immediate Updates for a Multiuser environment) outlines a recovery algorithm for concurrent transactions with immediate update (UNDO/REDO recovery). Assume that the log includes checkpoints and that the concurrency control protocol produces *strict schedules*—as, for example, the strict two-phase locking protocol does. Recall that a strict schedule does not allow a transaction to read or write an item unless the transaction that wrote the item has committed. However, deadlocks can occur in strict two-phase locking, thus requiring abort and UNDO of transactions. For a strict schedule, UNDO of an operation requires changing the item back to its old value (BFIM).

Procedure RIU_M (UNDO/REDO with checkpoints).

1. Use two lists of transactions maintained by the system: the committed transactions since the last checkpoint and the active transactions.
2. Undo all the `write_item` operations of the *active* (uncommitted) transactions, using the UNDO procedure. The operations should be undone in the reverse of the order in which they were written into the log.
3. Redo all the `write_item` operations of the *committed* transactions from the log, in the order in which they were written into the log, using the REDO procedure defined earlier.

The UNDO procedure is defined as follows:

Procedure UNDO (WRITE_OP). Undoing a `write_item` operation `write_op` consists of examining its log entry [`write_item`, *T*, *X*, `old_value`, `new_value`] and setting the value of item *X* in the database to `old_value`, which is the before image (BFIM). Undoing a number of `write_item` operations from one or more transactions from the log must proceed in the *reverse order* from the order in which the operations were written in the log.

As we discussed for the **NO-UNDO/REDO** procedure, step 3 is more efficiently done by starting from the *end of the log* and redoing only *the last update of each item X*. Whenever an item is redone, it is added to a list of redone items and is not redone again. A similar procedure can be devised to improve the efficiency of step 2 so that an item can be undone at most once during recovery. In this case, the earliest UNDO is applied first by scanning the log in the forward direction (starting from

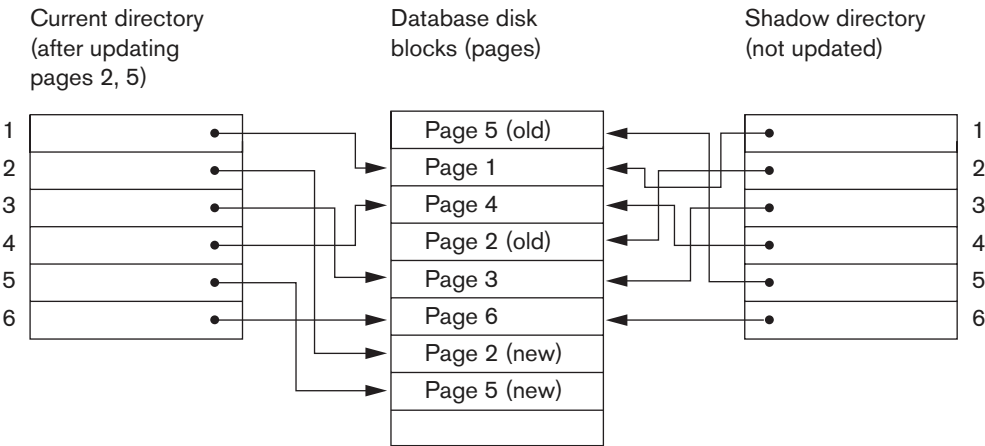
the beginning of the log). Whenever an item is undone, it is added to a list of undone items and is not undone again.

22.4 Shadow Paging

This recovery scheme does not require the use of a log in a single-user environment. In a multiuser environment, a log may be needed for the concurrency control method. Shadow paging considers the database to be made up of a number of fixed-size disk pages (or disk blocks)—say, n —for recovery purposes. A **directory** with n entries⁵ is constructed, where the i th entry points to the i th database page on disk. The directory is kept in main memory if it is not too large, and all references—reads or writes—to database pages on disk go through it. When a transaction begins executing, the **current directory**—whose entries point to the most recent or current database pages on disk—is copied into a **shadow directory**. The shadow directory is then saved on disk while the current directory is used by the transaction.

During transaction execution, the shadow directory is *never* modified. When a write_item operation is performed, a new copy of the modified database page is created, but the old copy of that page is *not overwritten*. Instead, the new page is written elsewhere—on some previously unused disk block. The current directory entry is modified to point to the new disk block, whereas the shadow directory is not modified and continues to point to the old unmodified disk block. Figure 22.4 illustrates the concepts of shadow and current directories. For pages updated by the transaction, two versions are kept. The old version is referenced by the shadow directory and the new version by the current directory.

Figure 22.4
An example of shadow paging.



⁵The directory is similar to the page table maintained by the operating system for each process.

To recover from a failure during transaction execution, it is sufficient to free the modified database pages and to discard the current directory. The state of the database before transaction execution is available through the shadow directory, and that state is recovered by reinstating the shadow directory. The database thus is returned to its state prior to the transaction that was executing when the crash occurred, and any modified pages are discarded. Committing a transaction corresponds to discarding the previous shadow directory. Since recovery involves neither undoing nor redoing data items, this technique can be categorized as a NO-UNDO/NO-REDO technique for recovery.

In a multiuser environment with concurrent transactions, logs and checkpoints must be incorporated into the shadow paging technique. One disadvantage of shadow paging is that the updated database pages change location on disk. This makes it difficult to keep related database pages close together on disk without complex storage management strategies. Furthermore, if the directory is large, the overhead of writing shadow directories to disk as transactions commit is significant. A further complication is how to handle **garbage collection** when a transaction commits. The old pages referenced by the shadow directory that have been updated must be released and added to a list of free pages for future use. These pages are no longer needed after the transaction commits. Another issue is that the operation to migrate between current and shadow directories must be implemented as an atomic operation.

22.5 The ARIES Recovery Algorithm

We now describe the ARIES algorithm as an example of a recovery algorithm used in database systems. It is used in many relational database-related products of IBM. ARIES uses a steal/no-force approach for writing, and it is based on three concepts: write-ahead logging, repeating history during redo, and logging changes during undo. We discussed write-ahead logging in Section 22.1.3. The second concept, **repeating history**, means that ARIES will retrace all actions of the database system prior to the crash to reconstruct the database state *when the crash occurred*. Transactions that were uncommitted at the time of the crash (active transactions) are undone. The third concept, **logging during undo**, will prevent ARIES from repeating the completed undo operations if a failure occurs during recovery, which causes a restart of the recovery process.

The ARIES recovery procedure consists of three main steps: analysis, REDO, and UNDO. The **analysis step** identifies the dirty (updated) pages in the buffer⁶ and the set of transactions active at the time of the crash. The appropriate point in the log where the REDO operation should start is also determined. The **REDO phase** actually reapplies updates from the log to the database. Generally, the REDO operation is applied only to committed transactions. However, this is not the case in ARIES.

⁶The actual buffers may be lost during a crash, since they are in main memory. Additional tables stored in the log during checkpointing (Dirty Page Table, Transaction Table) allow ARIES to identify this information (as discussed later in this section).

Certain information in the ARIES log will provide the start point for REDO, from which REDO operations are applied until the end of the log is reached. Additionally, information stored by ARIES and in the data pages will allow ARIES to determine whether the operation to be redone has actually been applied to the database and therefore does not need to be reapplied. Thus, *only the necessary REDO operations* are applied during recovery. Finally, during the **UNDO phase**, the log is scanned backward and the operations of transactions that were active at the time of the crash are undone in reverse order. The information needed for ARIES to accomplish its recovery procedure includes the log, the Transaction Table, and the Dirty Page Table. Additionally, checkpointing is used. These tables are maintained by the transaction manager and written to the log during checkpointing.

In ARIES, every log record has an associated **log sequence number (LSN)** that is monotonically increasing and indicates the address of the log record on disk. Each LSN corresponds to a *specific change* (action) of some transaction. Also, each data page will store the LSN of the *latest log record corresponding to a change for that page*. A log record is written for any of the following actions: updating a page (write), committing a transaction (commit), aborting a transaction (abort), undoing an update (undo), and ending a transaction (end). The need for including the first three actions in the log has been discussed, but the last two need some explanation. When an update is undone, a *compensation log record* is written in the log so that the undo does not have to be repeated. When a transaction ends, whether by committing or aborting, an *end log record* is written.

Common fields in all log records include the previous LSN for that transaction, the transaction ID, and the type of log record. The previous LSN is important because it links the log records (in reverse order) for each transaction. For an update (write) action, additional fields in the log record include the page ID for the page that contains the item, the length of the updated item, its offset from the beginning of the page, the before image of the item, and its after image.

In addition to the log, two tables are needed for efficient recovery: the **Transaction Table** and the **Dirty Page Table**, which are maintained by the transaction manager. When a crash occurs, these tables are rebuilt in the analysis phase of recovery. The Transaction Table contains an entry for *each active transaction*, with information such as the transaction ID, transaction status, and the LSN of the most recent log record for the transaction. The Dirty Page Table contains an entry for each dirty page in the DBMS cache, which includes the page ID and the LSN corresponding to the earliest update to that page.

Checkpointing in ARIES consists of the following: writing a `begin_checkpoint` record to the log, writing an `end_checkpoint` record to the log, and writing the LSN of the `begin_checkpoint` record to a special file. This special file is accessed during recovery to locate the last checkpoint information. With the `end_checkpoint` record, the contents of both the Transaction Table and Dirty Page Table are appended to the end of the log. To reduce the cost, **fuzzy checkpointing** is used so that the DBMS can continue to execute transactions during checkpointing (see Section 22.1.4). Additionally, the contents of the DBMS cache do not have to be flushed

to disk during checkpoint, since the Transaction Table and Dirty Page Table—which are appended to the log on disk—contain the information needed for recovery. Note that if a crash occurs during checkpointing, the special file will refer to the previous checkpoint, which would be used for recovery.

After a crash, the ARIES recovery manager takes over. Information from the last checkpoint is first accessed through the special file. The **analysis phase** starts at the `begin_checkpoint` record and proceeds to the end of the log. When the `end_checkpoint` record is encountered, the Transaction Table and Dirty Page Table are accessed (recall that these tables were written in the log during checkpointing). During analysis, the log records being analyzed may cause modifications to these two tables. For instance, if an end log record was encountered for a transaction T in the Transaction Table, then the entry for T is deleted from that table. If some other type of log record is encountered for a transaction T' , then an entry for T' is inserted into the Transaction Table, if not already present, and the last LSN field is modified. If the log record corresponds to a change for page P , then an entry would be made for page P (if not present in the table) and the associated LSN field would be modified. When the analysis phase is complete, the necessary information for REDO and UNDO has been compiled in the tables.

The **REDO phase** follows next. To reduce the amount of unnecessary work, ARIES starts redoing at a point in the log where it knows (for sure) that previous changes to dirty pages *have already been applied to the database on disk*. It can determine this by finding the smallest LSN, M , of all the dirty pages in the Dirty Page Table, which indicates the log position where ARIES needs to start the REDO phase. Any changes corresponding to an LSN $< M$, for redoable transactions, must have already been propagated to disk or already been overwritten in the buffer; otherwise, those dirty pages with that LSN would be in the buffer (and the Dirty Page Table). So, REDO starts at the log record with LSN = M and scans forward to the end of the log.

For each change recorded in the log, the REDO algorithm would verify whether or not the change has to be reapplied. For example, if a change recorded in the log pertains to page P that is not in the Dirty Page Table, then this change is already on disk and does not need to be reapplied. Or, if a change recorded in the log (with LSN = N , say) pertains to page P and the Dirty Page Table contains an entry for P with LSN greater than N , then the change is already present. If neither of these two conditions holds, page P is read from disk and the LSN stored on that page, $LSN(P)$, is compared with N . If $N < LSN(P)$, then the change has been applied and the page does not need to be rewritten to disk.

Once the REDO phase is finished, the database is in the exact state that it was in when the crash occurred. The set of active transactions—called the `undo_set`—has been identified in the Transaction Table during the analysis phase. Now, the **UNDO phase** proceeds by scanning backward from the end of the log and undoing the appropriate actions. A compensating log record is written for each action that is undone. The UNDO reads backward in the log until every action of the set of transactions in the `undo_set` has been undone. When this is completed, the recovery process is finished and normal processing can begin again.

(a)

Lsn	Last_Lsn	Tran_id	Type	Page_id	Other_information
1	0	T_1	update	C	...
2	0	T_2	update	B	...
3	1	T_1	commit		...
4	begin checkpoint				
5	end checkpoint				
6	0	T_3	update	A	...
7	2	T_2	update	C	...
8	7	T_2	commit		...

(b)	TRANSACTION TABLE			DIRTY PAGE TABLE	
	Transaction_id	Last_Lsn	Status	Page_id	Lsn
	T_1	3	commit	C	1
	T_2	2	in progress	B	2

(c)	TRANSACTION TABLE			DIRTY PAGE TABLE	
	Transaction_id	Last_Lsn	Status	Page_id	Lsn
	T_1	3	commit	C	7
	T_2	8	commit	B	2
	T_3	6	in progress	A	6

Figure 22.5

An example of recovery in ARIES. (a) The log at point of crash. (b) The Transaction and Dirty Page Tables at time of checkpoint. (c) The Transaction and Dirty Page Tables after the analysis phase.

Consider the recovery example shown in Figure 22.5. There are three transactions: T_1 , T_2 , and T_3 . T_1 updates page C, T_2 updates pages B and C, and T_3 updates page A. Figure 22.5(a) shows the partial contents of the log, and Figure 22.5(b) shows the contents of the Transaction Table and Dirty Page Table. Now, suppose that a crash occurs at this point. Since a checkpoint has occurred, the address of the associated begin_checkpoint record is retrieved, which is location 4. The analysis phase starts from location 4 until it reaches the end. The end_checkpoint record contains the Transaction Table and Dirty Page Table in Figure 22.5(b), and the analysis phase will further reconstruct these tables. When the analysis phase encounters log record 6, a new entry for transaction T_3 is made in the Transaction Table and a new entry for page A is made in the Dirty Page Table. After log record 8 is analyzed, the status of transaction T_2 is changed to committed in the Transaction Table. Figure 22.5(c) shows the two tables after the analysis phase.

For the REDO phase, the smallest LSN in the Dirty Page Table is 1. Hence the REDO will start at log record 1 and proceed with the REDO of updates. The LSNs {1, 2, 6, 7} corresponding to the updates for pages C, B, A, and C, respectively, are not less than the LSNs of those pages (as shown in the Dirty Page Table). So those data pages will be read again and the updates reapplied from the log (assuming the actual LSNs stored on those data pages are less than the corresponding log entry). At this point, the REDO phase is finished and the UNDO phase starts. From the Transaction Table (Figure 22.5(c)), UNDO is applied only to the active transaction T_3 . The UNDO phase starts at log entry 6 (the last update for T_3) and proceeds backward in the log. The backward chain of updates for transaction T_3 (only log record 6 in this example) is followed and undone.

22.6 Recovery in Multidatabase Systems

So far, we have implicitly assumed that a transaction accesses a single database. In some cases, a single transaction, called a **multidatabase transaction**, may require access to multiple databases. These databases may even be stored on different types of DBMSs; for example, some DBMSs may be relational, whereas others are object-oriented, hierarchical, or network DBMSs. In such a case, each DBMS involved in the multidatabase transaction may have its own recovery technique and transaction manager separate from those of the other DBMSs. This situation is somewhat similar to the case of a distributed database management system (see Chapter 23), where parts of the database reside at different sites that are connected by a communication network.

To maintain the atomicity of a multidatabase transaction, it is necessary to have a two-level recovery mechanism. A **global recovery manager**, or **coordinator**, is needed to maintain information needed for recovery, in addition to the local recovery managers and the information they maintain (log, tables). The coordinator usually follows a protocol called the **two-phase commit protocol**, whose two phases can be stated as follows:

- **Phase 1.** When all participating databases signal the coordinator that the part of the multidatabase transaction involving each has concluded, the coordinator sends a message *prepare for commit* to each participant to get ready for committing the transaction. Each participating database receiving that message will force-write all log records and needed information for local recovery to disk and then send a *ready to commit* or *OK* signal to the coordinator. If the force-writing to disk fails or the local transaction cannot commit for some reason, the participating database sends a *cannot commit* or *not OK* signal to the coordinator. If the coordinator does not receive a reply from the database within a certain time out interval, it assumes a *not OK* response.
- **Phase 2.** If *all* participating databases reply *OK*, and the coordinator's vote is also *OK*, the transaction is successful, and the coordinator sends a *commit* signal for the transaction to the participating databases. Because all the local effects of the transaction and information needed for local recovery have

been recorded in the logs of the participating databases, local recovery from failure is now possible. Each participating database completes transaction commit by writing a [commit] entry for the transaction in the log and permanently updating the database if needed. Conversely, if one or more of the participating databases or the coordinator have a *not OK* response, the transaction has failed, and the coordinator sends a message to *roll back* or *UNDO* the local effect of the transaction to each participating database. This is done by undoing the local transaction operations, using the log.

The net effect of the two-phase commit protocol is that either all participating databases commit the effect of the transaction or none of them do. In case any of the participants—or the coordinator—fails, it is always possible to recover to a state where either the transaction is committed or it is rolled back. A failure during or before phase 1 usually requires the transaction to be rolled back, whereas a failure during phase 2 means that a successful transaction can recover and commit.

22.7 Database Backup and Recovery from Catastrophic Failures

So far, all the techniques we have discussed apply to noncatastrophic failures. A key assumption has been that the system log is maintained on the disk and is not lost as a result of the failure. Similarly, the shadow directory must be stored on disk to allow recovery when shadow paging is used. The recovery techniques we have discussed use the entries in the system log or the shadow directory to recover from failure by bringing the database back to a consistent state.

The recovery manager of a DBMS must also be equipped to handle more catastrophic failures such as disk crashes. The main technique used to handle such crashes is a **database backup**, in which the whole database and the log are periodically copied onto a cheap storage medium such as magnetic tapes or other large capacity offline storage devices. In case of a catastrophic system failure, the latest backup copy can be reloaded from the tape to the disk, and the system can be restarted.

Data from critical applications such as banking, insurance, stock market, and other databases is periodically backed up in its entirety and moved to physically separate safe locations. Subterranean storage vaults have been used to protect such data from flood, storm, earthquake, or fire damage. Events like the 9/11 terrorist attack in New York (in 2001) and the Katrina hurricane disaster in New Orleans (in 2005) have created a greater awareness of *disaster recovery of critical databases*.

To avoid losing all the effects of transactions that have been executed since the last backup, it is customary to back up the system log at more frequent intervals than full database backup by periodically copying it to magnetic tape. The system log is usually substantially smaller than the database itself and hence can be backed up more frequently. Therefore, users do not lose all transactions they have performed since the last database backup. All committed transactions recorded in the portion of the system log that has been backed up to tape can have their effect on the database

redone. A new log is started after each database backup. Hence, to recover from disk failure, the database is first recreated on disk from its latest backup copy on tape. Following that, the effects of all the committed transactions whose operations have been recorded in the backed-up copies of the system log are reconstructed.

22.8 Summary

In this chapter, we discussed the techniques for recovery from transaction failures. The main goal of recovery is to ensure the atomicity property of a transaction. If a transaction fails before completing its execution, the recovery mechanism has to make sure that the transaction has no lasting effects on the database. First in Section 22.1 we gave an informal outline for a recovery process, and then we discussed system concepts for recovery. These included a discussion of caching, in-place updating versus shadowing, before and after images of a data item, UNDO versus REDO recovery operations, steal/no-steal and force/no-force policies, system checkpointing, and the write-ahead logging protocol.

Next we discussed two different approaches to recovery: deferred update (Section 22.2) and immediate update (Section 22.3). Deferred update techniques postpone any actual updating of the database on disk until a transaction reaches its commit point. The transaction force-writes the log to disk before recording the updates in the database. This approach, when used with certain concurrency control methods, is designed never to require transaction rollback, and recovery simply consists of redoing the operations of transactions committed after the last checkpoint from the log. The disadvantage is that too much buffer space may be needed, since updates are kept in the buffers and are not applied to disk until a transaction commits. Deferred update can lead to a recovery algorithm known as NO-UNDO/REDO. Immediate update techniques may apply changes to the database on disk before the transaction reaches a successful conclusion. Any changes applied to the database must first be recorded in the log and force-written to disk so that these operations can be undone if necessary. We also gave an overview of a recovery algorithm for immediate update known as UNDO/REDO. Another algorithm, known as UNDO/NO-REDO, can also be developed for immediate update if all transaction actions are recorded in the database before commit.

We discussed the shadow paging technique for recovery in Section 22.4, which keeps track of old database pages by using a shadow directory. This technique, which is classified as NO-UNDO/NO-REDO, does not require a log in single-user systems but still needs the log for multiuser systems. We also presented ARIES in Section 22.5, which is a specific recovery scheme used in many of IBM's relational database products. Then in Section 22.6 we discussed the two-phase commit protocol, which is used for recovery from failures involving multidatabase transactions. Finally, we discussed recovery from catastrophic failures in Section 22.7, which is typically done by backing up the database and the log to tape. The log can be backed up more frequently than the database, and the backup log can be used to redo operations starting from the last database backup.

Review Questions

- 22.1. Discuss the different types of transaction failures. What is meant by *catastrophic failure*?
- 22.2. Discuss the actions taken by the `read_item` and `write_item` operations on a database.
- 22.3. What is the system log used for? What are the typical kinds of entries in a system log? What are checkpoints, and why are they important? What are transaction commit points, and why are they important?
- 22.4. How are buffering and caching techniques used by the recovery subsystem?
- 22.5. What are the before image (BFIM) and after image (AFIM) of a data item? What is the difference between in-place updating and shadowing, with respect to their handling of BFIM and AFIM?
- 22.6. What are UNDO-type and REDO-type log entries?
- 22.7. Describe the write-ahead logging protocol.
- 22.8. Identify three typical lists of transactions that are maintained by the recovery subsystem.
- 22.9. What is meant by *transaction rollback*? What is meant by *cascading rollback*? Why do practical recovery methods use protocols that do not permit cascading rollback? Which recovery techniques do not require any rollback?
- 22.10. Discuss the UNDO and REDO operations and the recovery techniques that use each.
- 22.11. Discuss the deferred update technique of recovery. What are the advantages and disadvantages of this technique? Why is it called the NO-UNDO/REDO method?
- 22.12. How can recovery handle transaction operations that do not affect the database, such as the printing of reports by a transaction?
- 22.13. Discuss the immediate update recovery technique in both single-user and multiuser environments. What are the advantages and disadvantages of immediate update?
- 22.14. What is the difference between the UNDO/REDO and the UNDO/NO-REDO algorithms for recovery with immediate update? Develop the outline for an UNDO/NO-REDO algorithm.
- 22.15. Describe the shadow paging recovery technique. Under what circumstances does it not require a log?
- 22.16. Describe the three phases of the ARIES recovery method.
- 22.17. What are log sequence numbers (LSNs) in ARIES? How are they used? What information do the Dirty Page Table and Transaction Table contain? Describe how fuzzy checkpointing is used in ARIES.

- 22.18.** What do the terms *steal/no-steal* and *force/no-force* mean with regard to buffer management for transaction processing?
- 22.19.** Describe the two-phase commit protocol for multidatabase transactions.
- 22.20.** Discuss how disaster recovery from catastrophic failures is handled.

Exercises

- 22.21.** Suppose that the system crashes before the $[\text{read_item}, T_3, A]$ entry is written to the log in Figure 22.1(b). Will that make any difference in the recovery process?
- 22.22.** Suppose that the system crashes before the $[\text{write_item}, T_2, D, 25, 26]$ entry is written to the log in Figure 22.1(b). Will that make any difference in the recovery process?
- 22.23.** Figure 22.6 shows the log corresponding to a particular schedule at the point of a system crash for four transactions T_1 , T_2 , T_3 , and T_4 . Suppose that we use the *immediate update protocol* with checkpointing. Describe the recovery process from the system crash. Specify which transactions are rolled back, which operations in the log are redone and which (if any) are undone, and whether any cascading rollback takes place.

$[\text{start_transaction}, T_1]$
$[\text{read_item}, T_1, A]$
$[\text{read_item}, T_1, D]$
$[\text{write_item}, T_1, D, 20, 25]$
$[\text{commit}, T_1]$
$[\text{checkpoint}]$
$[\text{start_transaction}, T_2]$
$[\text{read_item}, T_2, B]$
$[\text{write_item}, T_2, B, 12, 18]$
$[\text{start_transaction}, T_4]$
$[\text{read_item}, T_4, D]$
$[\text{write_item}, T_4, D, 25, 15]$
$[\text{start_transaction}, T_3]$
$[\text{write_item}, T_3, C, 30, 40]$
$[\text{read_item}, T_4, A]$
$[\text{write_item}, T_4, A, 30, 20]$
$[\text{commit}, T_4]$
$[\text{read_item}, T_2, D]$
$[\text{write_item}, T_2, D, 15, 25]$

← System crash

Figure 22.6

A sample schedule and its corresponding log.

- 22.24.** Suppose that we use the deferred update protocol for the example in Figure 22.6. Show how the log would be different in the case of deferred update by removing the unnecessary log entries; then describe the recovery process, using your modified log. Assume that only REDO operations are applied, and specify which operations in the log are redone and which are ignored.
- 22.25.** How does checkpointing in ARIES differ from checkpointing as described in Section 22.1.4?
- 22.26.** How are log sequence numbers used by ARIES to reduce the amount of REDO work needed for recovery? Illustrate with an example using the information shown in Figure 22.5. You can make your own assumptions as to when a page is written to disk.
- 22.27.** What implications would a no-steal/force buffer management policy have on checkpointing and recovery?

Choose the correct answer for each of the following multiple-choice questions:

- 22.28.** Incremental logging with deferred updates implies that the recovery system must
- store the old value of the updated item in the log
 - store the new value of the updated item in the log
 - store both the old and new value of the updated item in the log
 - store only the Begin Transaction and Commit Transaction records in the log
- 22.29.** The write-ahead logging (WAL) protocol simply means that
- writing of a data item should be done ahead of any logging operation
 - the log record for an operation should be written before the actual data is written
 - all log records should be written before a new transaction begins execution
 - the log never needs to be written to disk
- 22.30.** In case of transaction failure under a deferred update incremental logging scheme, which of the following will be needed?
- an undo operation
 - a redo operation
 - an undo and redo operation
 - none of the above
- 22.31.** For incremental logging with immediate updates, a log record for a transaction would contain
- a transaction name, a data item name, and the old and new value of the item
 - a transaction name, a data item name, and the old value of the item
 - a transaction name, a data item name, and the new value of the item
 - a transaction name and a data item name

- 22.32.** For correct behavior during recovery, undo and redo operations must be
- commutative
 - associative
 - idempotent
 - distributive
- 22.33.** When a failure occurs, the log is consulted and each operation is either undone or redone. This is a problem because
- searching the entire log is time consuming
 - many redos are unnecessary
 - both (a) and (b)
 - none of the above
- 22.34.** Using a log-based recovery scheme might improve performance as well as provide a recovery mechanism by
- writing the log records to disk when each transaction commits
 - writing the appropriate log records to disk during the transaction's execution
 - waiting to write the log records until multiple transactions commit and writing them as a batch
 - never writing the log records to disk
- 22.35.** There is a possibility of a cascading rollback when
- a transaction writes items that have been written only by a committed transaction
 - a transaction writes an item that is previously written by an uncommitted transaction
 - a transaction reads an item that is previously written by an uncommitted transaction
 - both (b) and (c)
- 22.36.** To cope with media (disk) failures, it is necessary
- for the DBMS to only execute transactions in a single user environment
 - to keep a redundant copy of the database
 - to never abort a transaction
 - all of the above
- 22.37.** If the shadowing approach is used for flushing a data item back to disk, then
- the item is written to disk only after the transaction commits
 - the item is written to a different location on disk
 - the item is written to disk before the transaction commits
 - the item is written to the same disk location from which it was read