CHAPTER 16

Recovery System

A computer system, like any other device, is subject to failure from a variety of causes: disk crash, power outage, software error, a fire in the machine room, even sabotage. In any failure, information may be lost. Therefore, the database system must take actions in advance to ensure that the atomicity and durability properties of transactions, introduced in Chapter 14, are preserved. An integral part of a database system is a **recovery scheme** that can restore the database to the consistent state that existed before the failure. The recovery scheme must also provide **high availability**; that is, it must minimize the time for which the database is not usable after a failure.

16.1 Failure Classification

There are various types of failure that may occur in a system, each of which needs to be dealt with in a different manner. In this chapter, we shall consider only the following types of failure:

- **Transaction failure**. There are two types of errors that may cause a transaction to fail:
  - **Logical error**. The transaction can no longer continue with its normal execution because of some internal condition, such as bad input, data not found, overflow, or resource limit exceeded.
  - **System error**. The system has entered an undesirable state (for example, deadlock), as a result of which a transaction cannot continue with its normal execution. The transaction, however, can be reexecuted at a later time.
- **System crash**. There is a hardware malfunction, or a bug in the database software or the operating system, that causes the loss of the content of volatile storage, and brings transaction processing to a halt. The content of nonvolatile storage remains intact, and is not corrupted.

The assumption that hardware errors and bugs in the software bring the system to a halt, but do not corrupt the nonvolatile storage contents, is known as the **fail-stop assumption**. Well-designed systems have numerous internal checks, at the hardware and the software level, that bring the system to a halt when there is an error. Hence, the fail-stop assumption is a reasonable one.

- **Disk failure**. A disk block loses its content as a result of either a head crash or failure during a data-transfer operation. Copies of the data on other disks, or archival backups on tertiary media, such as DVD or tapes, are used to recover from the failure.

To determine how the system should recover from failures, we need to identify the failure modes of those devices used for storing data. Next, we must consider how these failure modes affect the contents of the database. We can then propose algorithms to ensure database consistency and transaction atomicity despite failures. These algorithms, known as recovery algorithms, have two parts:

1. Actions taken during normal transaction processing to ensure that enough information exists to allow recovery from failures.
2. Actions taken after a failure to recover the database contents to a state that ensures database consistency, transaction atomicity, and durability.

16.2 Storage

As we saw in Chapter 10, the various data items in the database may be stored and accessed in a number of different storage media. In Section 14.3, we saw that storage media can be distinguished by their relative speed, capacity, and resilience to failure. We identified three categories of storage:

- **Volatile storage**
- **Nonvolatile storage**
- **Stable storage**

Stable storage or, more accurately, an approximation thereof, plays a critical role in recovery algorithms.

16.2.1 Stable-Storage Implementation

To implement stable storage, we need to replicate the needed information in several nonvolatile storage media (usually disk) with independent failure modes, and to update the information in a controlled manner to ensure that failure during data transfer does not damage the needed information.

Recall (from Chapter 10) that RAID systems guarantee that the failure of a single disk (even during data transfer) will not result in loss of data. The simplest and fastest form of RAID is the mirrored disk, which keeps two copies of each block, on separate disks. Other forms of RAID offer lower costs, but at the expense of lower performance.

RAID systems, however, cannot guard against data loss due to disasters such as fires or flooding. Many systems store archival backups of tapes off site to guard against such disasters. However, since tapes cannot be carried off site continually, updates since the most recent time that tapes were carried off site could be lost in such a disaster. More secure systems keep a copy of each block of stable storage at a remote site, writing it out over a computer network, in addition to storing the block on a local disk system. Since the blocks are output to a remote system as and when they are output to local storage, once an output operation is complete, the output is not lost, even in the event of a disaster such as a fire or flood. We study such *remote backup* systems in Section 16.9.

In the remainder of this section, we discuss how storage media can be protected from failure during data transfer. Block transfer between memory and disk storage can result in:

- **Successful completion**. The transferred information arrived safely at its destination.
- **Partial failure**. A failure occurred in the midst of transfer, and the destination block has incorrect information.
- **Total failure**. The failure occurred sufficiently early during the transfer that the destination block remains intact.

We require that, if a **data-transfer failure** occurs, the system detects it and invokes a recovery procedure to restore the block to a consistent state. To do so, the system must maintain two physical blocks for each logical database block; in the case of mirrored disks, both blocks are at the same location; in the case of remote backup, one of the blocks is local, whereas the other is at a remote site. An output operation is executed as follows:

1. Write the information onto the first physical block.
2. When the first write completes successfully, write the same information onto the second physical block.
3. The output is completed only after the second write completes successfully.

If the system fails while blocks are being written, it is possible that the two copies of a block are inconsistent with each other. During recovery, for each block, the system would need to examine two copies of the blocks. If both are the same and no detectable error exists, then no further actions are necessary. (Recall that errors in a disk block, such as a partial write to the block, are detected by storing a checksum with each block.) If the system detects an error in one block, then it

replaces its content with the content of the other block. If both blocks contain no detectable error, but they differ in content, then the system replaces the content of the first block with the value of the second. This recovery procedure ensures that a write to stable storage either succeeds completely (that is, updates all copies) or results in no change.

The requirement of comparing every corresponding pair of blocks during recovery is expensive to meet. We can reduce the cost greatly by keeping track of block writes that are in progress, using a small amount of nonvolatile RAM. On recovery, only blocks for which writes were in progress need to be compared.

The protocols for writing out a block to a remote site are similar to the protocols for writing blocks to a mirrored disk system, which we examined in Chapter 10, and particularly in Practice Exercise 10.3.

We can extend this procedure easily to allow the use of an arbitrarily large number of copies of each block of stable storage. Although a large number of copies reduces the probability of a failure to even lower than two copies do, it is usually reasonable to simulate stable storage with only two copies.

16.2.2 Data Access

As we saw in Chapter 10, the database system resides permanently on nonvolatile storage (usually disks) and only parts of the database are in memory at any time.[1] The database is partitioned into fixed-length storage units called **blocks**. Blocks are the units of data transfer to and from disk, and may contain several data items. We shall assume that no data item spans two or more blocks. This assumption is realistic for most data-processing applications, such as a bank or a university.

Transactions input information from the disk to main memory, and then output the information back onto the disk. The input and output operations are done in block units. The blocks residing on the disk are referred to as **physical blocks**; the blocks residing temporarily in main memory are referred to as **buffer blocks**. The area of memory where blocks reside temporarily is called the **disk buffer**.

Block movements between disk and main memory are initiated through the following two operations:

1. input(B) transfers the physical block B to main memory.
2. output(B) transfers the buffer block B to the disk, and replaces the appropriate physical block there.

Figure 16.1 illustrates this scheme.

Conceptually, each transaction T_i has a private work area in which copies of data items accessed and updated by T_i are kept. The system creates this work area when the transaction is initiated; the system removes it when the transaction

[1] There is a special category of database system, called *main-memory database systems*, where the entire database can be loaded into memory at once. We consider such systems in Section 26.4.

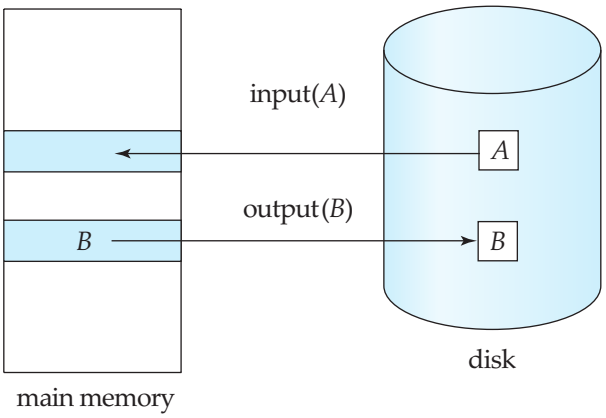

Figure 16.1 Block storage operations.

either commits or aborts. Each data item X kept in the work area of transaction T_i is denoted by x_i. Transaction T_i interacts with the database system by transferring data to and from its work area to the system buffer. We transfer data by these two operations:

1. read(X) assigns the value of data item X to the local variable x_i. It executes this operation as follows:
   a. If block B_X on which X resides is not in main memory, it issues input(B_X).
   b. It assigns to x_i the value of X from the buffer block.
2. write(X) assigns the value of local variable x_i to data item X in the buffer block. It executes this operation as follows:
   a. If block B_X on which X resides is not in main memory, it issues input(B_X).
   b. It assigns the value of x_i to X in buffer B_X.

Note that both operations may require the transfer of a block from disk to main memory. They do not, however, specifically require the transfer of a block from main memory to disk.

A buffer block is eventually written out to the disk either because the buffer manager needs the memory space for other purposes or because the database system wishes to reflect the change to B on the disk. We shall say that the database system performs a **force-output** of buffer B if it issues an output(B).

When a transaction needs to access a data item X for the first time, it must execute read(X). The system then performs all updates to X on x_i. At any point during its execution a transaction may execute write(X) to reflect the change to X in the database itself; write(X) must certainly be done after the final write to X.

The output(B_X) operation for the buffer block B_X on which X resides does not need to take effect immediately after write(X) is executed, since the block B_X may contain other data items that are still being accessed. Thus, the actual output may take place later. Notice that, if the system crashes after the write(X) operation was executed but before output(B_X) was executed, the new value of X is never written to disk and, thus, is lost. As we shall see shortly, the database system executes extra actions to ensure that updates performed by committed transactions are not lost even if there is a system crash.

16.3 Recovery and Atomicity

Consider again our simplified banking system and a transaction T_i that transfers \$50 from account A to account B, with initial values of A and B being \$1000 and \$2000, respectively. Suppose that a system crash has occurred during the execution of T_i, after output(B_A) has taken place, but before output(B_B) was executed, where B_A and B_B denote the buffer blocks on which A and B reside. Since the memory contents were lost, we do not know the fate of the transaction.

When the system restarts, the value of A would be \$950, while that of B would be \$2000, which is clearly inconsistent with the atomicity requirement for transaction T_i. Unfortunately, there is no way to find out by examining the database state what blocks had been output, and what had not, before the crash. It is possible that the transaction completed, updating the database on stable storage from an initial state with the values of A and B being \$1000 and \$1950; it is also possible that the transaction did not affect the stable storage at all, and the values of A and B were \$950 and \$2000 initially; or that the updated B was output but not the updated A; or that the updated A was output but the updated B was not.

Our goal is to perform either all or no database modifications made by T_i. However, if T_i performed multiple database modifications, several output operations may be required, and a failure may occur after some of these modifications have been made, but before all of them are made.

To achieve our goal of atomicity, we must first output to stable storage information describing the modifications, without modifying the database itself. As we shall see, this information can help us ensure that all modifications performed by committed transactions are reflected in the database (perhaps during the course of recovery actions after a crash). This information can also help us ensure that no modifications made by an aborted transaction persist in the database.

16.3.1 Log Records

The most widely used structure for recording database modifications is the **log**. The log is a sequence of **log records**, recording all the update activities in the database.

There are several types of log records. An **update log record** describes a single database write. It has these fields:

SHADOW COPIES AND SHADOW PAGING

In the **shadow-copy** scheme, a transaction that wants to update the database first creates a complete copy of the database. All updates are done on the new database copy, leaving the original copy, the **shadow copy**, untouched. If at any point the transaction has to be aborted, the system merely deletes the new copy. The old copy of the database has not been affected. The current copy of the database is identified by a pointer, called db-pointer, which is stored on disk.

If the transaction partially commits (that is, executes its final statement) it is committed as follows: First, the operating system is asked to make sure that all pages of the new copy of the database have been written out to disk. (Unix systems use the fsync command for this purpose.) After the operating system has written all the pages to disk, the database system updates the pointer db-pointer to point to the new copy of the database; the new copy then becomes the current copy of the database. The old copy of the database is then deleted. The transaction is said to have been *committed* at the point where the updated db-pointer is written to disk.

The implementation actually depends on the write to db-pointer being atomic; that is, either all its bytes are written or none of its bytes are written. Disk systems provide atomic updates to entire blocks, or at least to a disk sector. In other words, the disk system guarantees that it will update db-pointer atomically, as long as we make sure that db-pointer lies entirely in a single sector, which we can ensure by storing db-pointer at the beginning of a block.

Shadow copy schemes are commonly used by text editors (saving the file is equivalent to transaction commit, while quitting without saving the file is equivalent to transaction abort). Shadow copying can be used for small databases, but copying a large database would be extremely expensive. A variant of shadow-copying, called **shadow-paging**, reduces copying as follows: the scheme uses a page table containing pointers to all pages; the page table itself and all updated pages are copied to a new location. Any page which is not updated by a transaction is not copied, but instead the new page table just stores a pointer to the original page. When a transaction commits, it atomically updates the pointer to the page table, which acts as db-pointer, to point to the new copy.

Shadow paging unfortunately does not work well with concurrent transactions and is not widely used in databases.

- **Transaction identifier**, which is the unique identifier of the transaction that performed the write operation.
- **Data-item identifier**, which is the unique identifier of the data item written. Typically, it is the location on disk of the data item, consisting of the block identifier of the block on which the data item resides, and an offset within the block.
- **Old value**, which is the value of the data item prior to the write.

- **New value**, which is the value that the data item will have after the write.

We represent an update log record as $<T_i, X_j, V_1, V_2>$, indicating that transaction T_i has performed a write on data item X_j. X_j had value V_1 before the write, and has value V_2 after the write. Other special log records exist to record significant events during transaction processing, such as the start of a transaction and the commit or abort of a transaction. Among the types of log records are:

- $<T_i$ start$>$. Transaction T_i has started.
- $<T_i$ commit$>$. Transaction T_i has committed.
- $<T_i$ abort$>$. Transaction T_i has aborted.

We shall introduce several other types of log records later.

Whenever a transaction performs a write, it is essential that the log record for that write be created and added to the log, before the database is modified. Once a log record exists, we can output the modification to the database if that is desirable. Also, we have the ability to *undo* a modification that has already been output to the database. We undo it by using the old-value field in log records.

For log records to be useful for recovery from system and disk failures, the log must reside in stable storage. For now, we assume that every log record is written to the end of the log on stable storage as soon as it is created. In Section 16.5, we shall see when it is safe to relax this requirement so as to reduce the overhead imposed by logging. Observe that the log contains a complete record of all database activity. As a result, the volume of data stored in the log may become unreasonably large. In Section 16.3.6, we shall show when it is safe to erase log information.

16.3.2 Database Modification

As we noted earlier, a transaction creates a log record prior to modifying the database. The log records allow the system to undo changes made by a transaction in the event that the transaction must be aborted; they allow the system also to redo changes made by a transaction if the transaction has committed but the system crashed before those changes could be stored in the database on disk. In order for us to understand the role of these log records in recovery, we need to consider the steps a transaction takes in modifying a data item:

1. The transaction performs some computations in its own private part of main memory.
2. The transaction modifies the data block in the disk buffer in main memory holding the data item.
3. The database system executes the output operation that writes the data block to disk.

We say a transaction *modifies the database* if it performs an update on a disk buffer, or on the disk itself; updates to the private part of main memory do not count as database modifications. If a transaction does not modify the database until it has committed, it is said to use the **deferred-modification** technique. If database modifications occur while the transaction is still active, the transaction is said to use the **immediate-modification** technique. Deferred modification has the overhead that transactions need to make local copies of all updated data items; further, if a transaction reads a data item that it has updated, it must read the value from its local copy.

The recovery algorithms we describe in this chapter support immediate modification. As described, they work correctly even with deferred modification, but can be optimized to reduce overhead when used with deferred modification; we leave details as an exercise.

A recovery algorithm must take into account a variety of factors, including:

- The possibility that a transaction may have committed although some of its database modifications exist only in the disk buffer in main memory and not in the database on disk.
- The possibility that a transaction may have modified the database while in the active state and, as a result of a subsequent failure, may need to abort.

Because all database modifications must be preceded by the creation of a log record, the system has available both the old value prior to the modification of the data item and the new value that is to be written for the data item. This allows the system to perform *undo* and *redo* operations as appropriate.

- **Undo** using a log record sets the data item specified in the log record to the old value.
- **Redo** using a log record sets the data item specified in the log record to the new value.

16.3.3 Concurrency Control and Recovery

If the concurrency control scheme allows a data item X that has been modified by a transaction T_1 to be further modified by another transaction T_2 before T_1 commits, then undoing the effects of T_1 by restoring the old value of X (before T_1 updated X) would also undo the effects of T_2. To avoid such situations, recovery algorithms usually require that if a data item has been modified by a transaction, no other transaction can modify the data item until the first transaction commits or aborts.

This requirement can be ensured by acquiring an exclusive lock on any updated data item and holding the lock until the transaction commits; in other words, by using strict two-phase locking. Snapshot-isolation and validation-

based concurrency-control techniques also acquire exclusive locks on data items at the time of validation, before modifying the data items, and hold the locks until the transaction is committed; as a result the above requirement is satisfied even by these concurrency control protocols.

We discuss later, in Section 16.7, how the above requirement can be relaxed in certain cases.

When either snapshot-isolation or validation is used for concurrency control, database updates of a transaction are (conceptually) deferred until the transaction is partially committed; the deferred-modification technique is a natural fit with these concurrency control schemes. However, it is worth noting that some implementations of snapshot isolation use immediate modification, but provide a logical snapshot on demand: when a transaction needs to read an item that a concurrent transaction has updated, a copy of the (already updated) item is made, and updates made by concurrent transactions are rolled back on the copy of the item. Similarly, immediate modification of the database is a natural fit with two-phase locking, but deferred modification can also be used with two-phase locking.

16.3.4 Transaction Commit

We say that a transaction has **committed** when its commit log record, which is the last log record of the transaction, has been output to stable storage; at that point all earlier log records have already been output to stable storage. Thus, there is enough information in the log to ensure that even if there is a system crash, the updates of the transaction can be redone. If a system crash occurs before a log record $<$ T_i commit$>$ is output to stable storage, transaction T_i will be rolled back. Thus, the output of the block containing the commit log record is the single atomic action that results in a transaction getting committed.[2]

With most log-based recovery techniques, including the ones we describe in this chapter, blocks containing the data items modified by a transaction do not have to be output to stable storage when the transaction commits, but can be output some time later. We discuss this issue further in Section 16.5.2.

16.3.5 Using the Log to Redo and Undo Transactions

We now provide an overview of how the log can be used to recover from a system crash, and to roll back transactions during normal operation. However, we postpone details of the procedures for failure recovery and rollback to Section 16.4.

Consider our simplified banking system. Let T_0 be a transaction that transfers \$50 from account A to account B:

[2] The output of a block can be made atomic by techniques for dealing with data-transfer failure, as described earlier in Section 16.2.1.

<T_0 start>
<T_0, A, 1000, 950>
<T_0, B, 2000, 2050>
<T_0 commit>
<T_1 start>
<T_1, C, 700, 600>
<T_1 commit>

Figure 16.2 Portion of the system log corresponding to T_0 and T_1.

T_0: read(A);
$A := A - 50$;
write(A);
read(B);
$B := B + 50$;
write(B).

Let T_1 be a transaction that withdraws $100 from account C:

T_1: read(C);
$C := C - 100$;
write(C).

The portion of the log containing the relevant information concerning these two transactions appears in Figure 16.2.

Figure 16.3 shows one possible order in which the actual outputs took place in both the database system and the log as a result of the execution of T_0 and T_1.[3]

Using the log, the system can handle any failure that does not result in the loss of information in nonvolatile storage. The recovery scheme uses two recovery procedures. Both these procedures make use of the log to find the set of data items updated by each transaction T_i, and their respective old and new values.

- redo(T_i) sets the value of all data items updated by transaction T_i to the new values.

  The order in which updates are carried out by redo is important; when recovering from a system crash, if updates to a particular data item are applied in an order different from the order in which they were applied originally, the final state of that data item will have a wrong value. Most recovery algorithms, including the one we describe in Section 16.4, do not perform redo of each transaction separately; instead they perform a single scan of the log, during which redo actions are performed for each log record as it is encountered. This approach ensures the order of updates is preserved,

[3]Notice that this order could not be obtained using the deferred-modification technique, because the database is modified by T_0 before it commits, and likewise for T_1.

Log	Database
<T_0 start>	
<T_0, A, 1000, 950>	
<T_0, B, 2000, 2050>	
	$A = 950$
	$B = 2050$
<T_0 commit>	
<T_1 start>	
<T_1, C, 700, 600>	
	$C = 600$
<T_1 commit>	

Figure 16.3 State of system log and database corresponding to T_0 and T_1.

and is more efficient since the log needs to be read only once overall, instead of once per transaction.

- undo(T_i) restores the value of all data items updated by transaction T_i to the old values.

  In the recovery scheme that we describe in Section 16.4:

  - The undo operation not only restores the data items to their old value, but also writes log records to record the updates performed as part of the undo process. These log records are special **redo-only** log records, since they do not need to contain the old-value of the updated data item.

    As with the redo procedure, the order in which undo operations are performed is important; again we postpone details to Section 16.4.

  - When the undo operation for transaction T_i completes, it writes a <T_i abort> log record, indicating that the undo has completed.

    As we shall see in Section 16.4, the undo(T_i) procedure is executed only once for a transaction, if the transaction is rolled back during normal processing or if on recovering from a system crash, neither a commit nor an abort record is found for transaction T_i. As a result, every transaction will eventually have either a commit or an abort record in the log.

After a system crash has occurred, the system consults the log to determine which transactions need to be redone, and which need to be undone so as to ensure atomicity.

- Transaction T_i needs to be undone if the log contains the record <T_i start>, but does not contain either the record <T_i commit> or the record <T_i abort>.
- Transaction T_i needs to be redone if the log contains the record <T_i start> and either the record <T_i commit> or the record <T_i abort>. It may seem strange to redo T_i if the record <T_i abort> is in the log. To see why this works, note

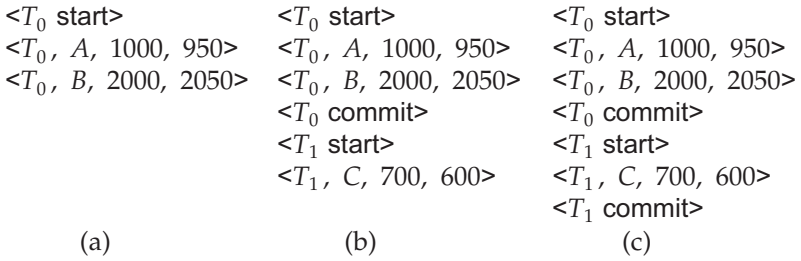

Figure 16.4 The same log, shown at three different times.

that if $<T_i$ abort$>$ is in the log, so are the redo-only records written by the undo operation. Thus, the end result will be to undo T_i's modifications in this case. This slight redundancy simplifies the recovery algorithm and enables faster overall recovery time.

As an illustration, return to our banking example, with transaction T_0 and T_1 executed one after the other in the order T_0 followed by T_1. Suppose that the system crashes before the completion of the transactions. We shall consider three cases. The state of the logs for each of these cases appears in Figure 16.4.

First, let us assume that the crash occurs just after the log record for the step:

write(B)

of transaction T_0 has been written to stable storage (Figure 16.4a). When the system comes back up, it finds the record $<T_0$ start$>$ in the log, but no corresponding $<T_0$ commit$>$ or $<T_0$ abort$>$ record. Thus, transaction T_0 must be undone, so an undo(T_0) is performed. As a result, the values in accounts A and B (on the disk) are restored to \$1000 and \$2000, respectively.

Next, let us assume that the crash comes just after the log record for the step:

write(C)

of transaction T_1 has been written to stable storage (Figure 16.4b). When the system comes back up, two recovery actions need to be taken. The operation undo(T_1) must be performed, since the record $<T_1$ start$>$ appears in the log, but there is no record $<T_1$ commit$>$ or $<T_1$ abort$>$. The operation redo(T_0) must be performed, since the log contains both the record $<T_0$ start$>$ and the record $<T_0$ commit$>$. At the end of the entire recovery procedure, the values of accounts A, B, and C are \$950, \$2050, and \$700, respectively.

Finally, let us assume that the crash occurs just after the log record:

$<T_1$ commit$>$

has been written to stable storage (Figure 16.4c). When the system comes back up, both T_0 and T_1 need to be redone, since the records $<T_0$ start$>$ and $<T_0$ commit$>$ appear in the log, as do the records $<T_1$ start$>$ and $<T_1$ commit$>$. After the system performs the recovery procedures redo(T_0) and redo(T_1), the values in accounts A, B, and C are \$950, \$2050, and \$600, respectively.

16.3.6 Checkpoints

When a system crash occurs, we must consult the log to determine those transactions that need to be redone and those that need to be undone. In principle, we need to search the entire log to determine this information. There are two major difficulties with this approach:

1. The search process is time-consuming.
2. Most of the transactions that, according to our algorithm, need to be redone have already written their updates into the database. Although redoing them will cause no harm, it will nevertheless cause recovery to take longer.

To reduce these types of overhead, we introduce checkpoints.

We describe below a simple checkpoint scheme that (a) does not permit any updates to be performed while the checkpoint operation is in progress, and (b) outputs all modified buffer blocks to disk when the checkpoint is performed. We discuss later how to modify the checkpointing and recovery procedures to provide more flexibility by relaxing both these requirements.

A checkpoint is performed as follows:

1. Output onto stable storage all log records currently residing in main memory.
2. Output to the disk all modified buffer blocks.
3. Output onto stable storage a log record of the form $<$checkpoint $L>$, where L is a list of transactions active at the time of the checkpoint.

Transactions are not allowed to perform any update actions, such as writing to a buffer block or writing a log record, while a checkpoint is in progress. We discuss how this requirement can be enforced, later, in Section 16.5.2.

The presence of a $<$checkpoint $L>$ record in the log allows the system to streamline its recovery procedure. Consider a transaction T_i that completed prior to the checkpoint. For such a transaction, the $<T_i$ commit$>$ record (or $< T_i$ abort$>$ record) appears in the log before the $<$checkpoint$>$ record. Any database modifications made by T_i must have been written to the database either prior to the checkpoint or as part of the checkpoint itself. Thus, at recovery time, there is no need to perform a redo operation on T_i.

After a system crash has occurred, the system examines the log to find the last $<$checkpoint $L>$ record (this can be done by searching the log backward, from the end of the log, until the first $<$checkpoint $L>$ record is found).

The redo or undo operations need to be applied only to transactions in L, and to all transactions that started execution after the <checkpoint L> record was written to the log. Let us denote this set of transactions as T.

- For all transactions T_k in T that have no <T_k commit> record or <T_k abort> record in the log, execute undo(T_k).
- For all transactions T_k in T such that either the record <T_k commit> or the record <T_k abort> appears in the log, execute redo(T_k).

Note that we need only examine the part of the log starting with the last checkpoint log record to find the set of transactions T, and to find out whether a commit or abort record occurs in the log for each transaction in T.

As an illustration, consider the set of transactions $\{T_0, T_1, \ldots, T_{100}\}$. Suppose that the most recent checkpoint took place during the execution of transaction T_{67} and T_{69}, while T_{68} and all transactions with subscripts lower than 67 completed before the checkpoint. Thus, only transactions $T_{67}, T_{69}, \ldots, T_{100}$ need to be considered during the recovery scheme. Each of them needs to be redone if it has completed (that is, either committed or aborted); otherwise, it was incomplete, and needs to be undone.

Consider the set of transactions L in a checkpoint log record. For each transaction T_i in L, log records of the transaction that occur prior to the checkpoint log record may be needed to undo the transaction, in case it does not commit. However, all log records prior to the earliest of the < T_i start> log records, among transactions T_i in L, are not needed once the checkpoint has completed. These log records can be erased whenever the database system needs to reclaim the space occupied by these records.

The requirement that transactions must not perform any updates to buffer blocks or to the log during checkpointing can be bothersome, since transaction processing has to halt while a checkpoint is in progress. A **fuzzy checkpoint** is a checkpoint where transactions are allowed to perform updates even while buffer blocks are being written out. Section 16.5.4 describes fuzzy-checkpointing schemes. Later in Section 16.8 we describe a checkpoint scheme that is not only fuzzy, but does not even require all modified buffer blocks to be output to disk at the time of the checkpoint.

16.4 Recovery Algorithm

Until now, in discussing recovery, we have identified transactions that need to be redone and those that need to be undone, but we have not given a precise algorithm for performing these actions. We are now ready to present the full recovery algorithm using log records for recovery from transaction failure and a combination of the most recent checkpoint and log records to recover from a system crash.

The recovery algorithm described in this section requires that a data item that has been updated by an uncommitted transaction cannot be modified by any other transaction, until the first transaction has either committed or aborted. Recall that this restriction was discussed earlier, in Section 16.3.3.

16.4.1 Transaction Rollback

First consider transaction rollback during normal operation (that is, not during recovery from a system crash). Rollback of a transaction T_i is performed as follows:

1. The log is scanned backward, and for each log record of T_i of the form $<T_i, X_j, V_1, V_2>$ that is found:
   a. The value V_1 is written to data item X_j, and
   b. A special redo-only log record $<T_i, X_j, V_1>$ is written to the log, where V_1 is the value being restored to data item X_j during the rollback. These log records are sometimes called **compensation log records**. Such records do not need undo information, since we never need to undo such an undo operation. We shall explain later how they are used.
2. Once the log record $<T_i$ start$>$ is found the backward scan is stopped, and a log record $<T_i$ abort$>$ is written to the log.

Observe that every update action performed by the transaction or on behalf of the transaction, including actions taken to restore data items to their old value, have now been recorded in the log. In Section 16.4.2 we shall see why this is a good idea.

16.4.2 Recovery After a System Crash

Recovery actions, when the database system is restarted after a crash, take place in two phases:

1. In the **redo phase**, the system replays updates of *all* transactions by scanning the log forward from the last checkpoint. The log records that are replayed include log records for transactions that were rolled back before system crash, and those that had not committed when the system crash occurred. This phase also determines all transactions that were incomplete at the time of the crash, and must therefore be rolled back. Such incomplete transactions would either have been active at the time of the checkpoint, and thus would appear in the transaction list in the checkpoint record, or would have started later; further, such incomplete transactions would have neither a $<T_i$ abort$>$ nor a $<T_i$ commit$>$ record in the log.
   The specific steps taken while scanning the log are as follows:
   a. The list of transactions to be rolled back, undo-list, is initially set to the list L in the $<$checkpoint $L>$ log record.

b. Whenever a normal log record of the form $<T_i, X_j, V_1, V_2>$, or a redo-only log record of the form $<T_i, X_j, V_2>$ is encountered, the operation is redone; that is, the value V_2 is written to data item X_j.

c. Whenever a log record of the form $<T_i$ start$>$ is found, T_i is added to undo-list.

d. Whenever a log record of the form $<T_i$ abort$>$ or $<T_i$ commit$>$ is found, T_i is removed from undo-list.

At the end of the redo phase, undo-list contains the list of all transactions that are incomplete, that is, they neither committed nor completed rollback before the crash.

2. In the **undo phase**, the system rolls back all transactions in the undo-list. It performs rollback by scanning the log backward from the end.

a. Whenever it finds a log record belonging to a transaction in the undo-list, it performs undo actions just as if the log record had been found during the rollback of a failed transaction.

b. When the system finds a $<T_i$ start$>$ log record for a transaction T_i in undo-list, it writes a $<T_i$ abort$>$ log record to the log, and removes T_i from undo-list.

c. The undo phase terminates once undo-list becomes empty, that is, the system has found $<T_i$ start$>$ log records for all transactions that were initially in undo-list.

After the undo phase of recovery terminates, normal transaction processing can resume.

Observe that the redo phase replays every log record since the most recent checkpoint record. In other words, this phase of restart recovery repeats all the update actions that were executed after the checkpoint, and whose log records reached the stable log. The actions include actions of incomplete transactions and the actions carried out to rollback failed transactions. The actions are repeated in the same order in which they were originally carried out; hence, this process is called **repeating history**. Although it may appear wasteful, repeating history even for failed transactions simplifies recovery schemes.

Figure 16.5 shows an example of actions logged during normal operation, and actions performed during failure recovery. In the log shown in the figure, transaction T_1 had committed, and transaction T_0 had been completely rolled back, before the system crashed. Observe how the value of data item B is restored during the rollback of T_0. Observe also the checkpoint record, with the list of active transactions containing T_0 and T_1.

When recovering from a crash, in the redo phase, the system performs a redo of all operations after the last checkpoint record. In this phase, the list undo-list initially contains T_0 and T_1; T_1 is removed first when its commit log record is found, while T_2 is added when its start log record is found. Transaction T_0 is

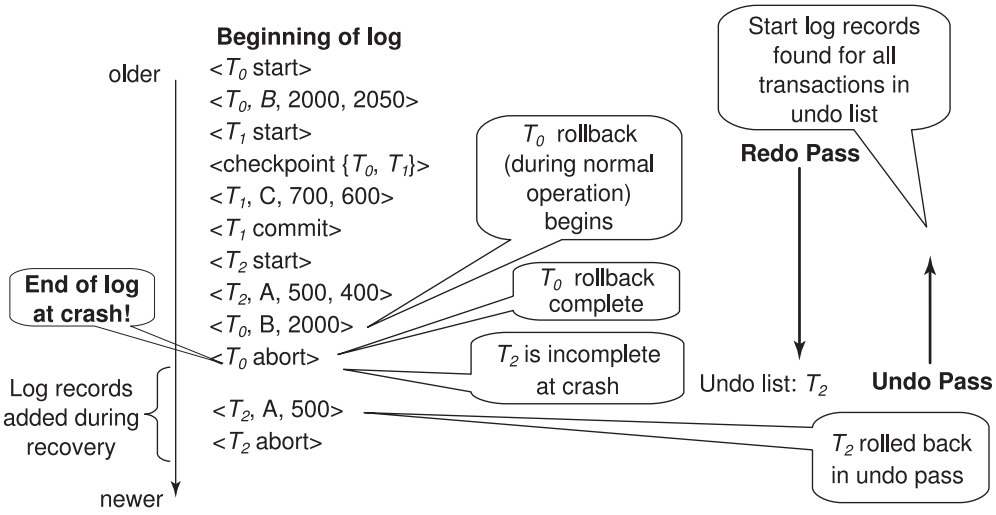

Figure 16.5 Example of logged actions, and actions during recovery.

removed from undo-list when its abort log record is found, leaving only T_2 in undo-list. The undo phase scans the log backwards from the end, and when it finds a log record of T_2 updating A, the old value of A is restored, and a redo-only log record written to the log. When the start record for T_2 is found, an abort record is added for T_2. Since undo-list contains no more transactions, the undo phase terminates, completing recovery.

16.5 Buffer Management

In this section, we consider several subtle details that are essential to the implementation of a crash-recovery scheme that ensures data consistency and imposes a minimal amount of overhead on interactions with the database.

16.5.1 Log-Record Buffering

So far, we have assumed that every log record is output to stable storage at the time it is created. This assumption imposes a high overhead on system execution for several reasons: Typically, output to stable storage is in units of blocks. In most cases, a log record is much smaller than a block. Thus, the output of each log record translates to a much larger output at the physical level. Furthermore, as we saw in Section 16.2.1, the output of a block to stable storage may involve several output operations at the physical level.

The cost of outputting a block to stable storage is sufficiently high that it is desirable to output multiple log records at once. To do so, we write log records to a log buffer in main memory, where they stay temporarily until they are output to stable storage. Multiple log records can be gathered in the log buffer and output