Chapter 17

Coping With System Failures

Starting with this chapter, we focus our attention on those parts of a DBMS that control access to data. There are two major issues to address:

1. Data must be protected in the face of a system failure. This chapter deals with techniques for supporting the goal of *resilience*, that is, integrity of the data when the system fails in some way.

2. Data must not be corrupted simply because several error-free queries or database modifications are being done at once. This matter is addressed in Chapters 18 and 19.

The principal technique for supporting resilience is a *log*, which records securely the history of database changes. We shall discuss three different styles of logging, called "undo," "redo," and "undo/redo." We also discuss *recovery*, the process whereby the log is used to reconstruct what has happened to the database when there has been a failure. An important aspect of logging and recovery is avoidance of the situation where the log must be examined into the distant past. Thus, we shall learn about "checkpointing," which limits the length of log that must be examined during recovery.

In a final section, we discuss "archiving," which allows the database to survive not only temporary system failures, but situations where the entire database is lost. Then, we must rely on a recent copy of the database (the archive) plus whatever log information survives, to reconstruct the database as it existed at some point in the recent past.

17.1 Issues and Models for Resilient Operation

We begin our discussion of coping with failures by reviewing the kinds of things that can go wrong, and what a DBMS can and should do about them. We initially focus on "system failures" or "crashes," the kinds of errors that the logging and recovery methods are designed to fix. We also introduce in Section 17.1.4 the model for buffer management that underlies all discussions of recovery from system errors. The same model is needed in the next chapter as we discuss concurrent access to the database by several transactions.

17.1.1 Failure Modes

There are many things that can go wrong as a database is queried and modified. Problems range from the keyboard entry of incorrect data to an explosion in the room where the database is stored on disk. The following items are a catalog of the most important failure modes and what the DBMS can do about them.

Erroneous Data Entry

Some data errors are impossible to detect. For example, if a clerk mistypes one digit of your phone number, the data will still look like a phone number that *could* be yours. On the other hand, if the clerk omits a digit from your phone number, then the data is evidently in error, since it does not have the form of a phone number. The principal technique for addressing data-entry errors is to write constraints and triggers that detect data believed to be erroneous.

Media Failures

A local failure of a disk, one that changes only a bit or a few bits, can normally be detected by parity checks associated with the sectors of the disk, as we discussed in Section 13.4.2. Head crashes, where the entire disk becomes unreadable, are generally handled by one or both of the following approaches:

1. Use one of the RAID schemes discussed in Section 13.4, so the lost disk can be restored.

2. Maintain an ***archive***, a copy of the database on a medium such as tape or optical disk. The archive is periodically created, either fully or incrementally, and stored at a safe distance from the database itself. We shall discuss archiving in Section 17.5.

3. Instead of an archive, one could keep redundant copies of the database on-line, distributed among several sites. These copies are kept consistent by mechanisms we shall discuss in Section 20.6.

Catastrophic Failure

In this category are a number of situations in which the media holding the database is completely destroyed. Examples include explosions, fires, or vandalism at the site of the database. RAID will not help, since all the data disks and their parity check disks become useless simultaneously. However, the other approaches that can be used to protect against media failure — archiving and redundant, distributed copies — will also protect against a catastrophic failure.

System Failures

The processes that query and modify the database are called *transactions*. A transaction, like any program, executes a number of steps in sequence; often, several of these steps will modify the database. Each transaction has a *state*, which represents what has happened so far in the transaction. The state includes the current place in the transaction's code being executed and the values of any local variables of the transaction that will be needed later on.

System failures are problems that cause the state of a transaction to be lost. Typical system failures are power loss and software errors. Since main memory is "volatile," as we discussed in Section 13.1.3, a power failure will cause the contents of main memory to disappear, along with the result of any transaction step that was kept only in main memory, rather than on (nonvolatile) disk. Similarly, a software error may overwrite part of main memory, possibly including values that were part of the state of the program.

When main memory is lost, the transaction state is lost; that is, we can no longer tell what parts of the transaction, including its database modifications, were made. Running the transaction again may not fix the problem. For example, if the transaction must add 1 to a value in the database, we do not know whether to repeat the addition of 1 or not. The principal remedy for the problems that arise due to a system error is logging of all database changes in a separate, nonvolatile log, coupled with recovery when necessary. However, the mechanisms whereby such logging can be done in a fail-safe manner are surprisingly intricate, as we shall see starting in Section 17.2.

17.1.2 More About Transactions

We introduced the idea of transactions from the point of view of the SQL programmer in Section 6.6. Before proceeding to our study of database resilience and recovery from failures, we need to discuss the fundamental notion of a transaction in more detail.

The transaction is the unit of execution of database operations. For example, if we are issuing ad-hoc commands to a SQL system, then each query or database modification statement (plus any resulting trigger actions) is a transaction. When using an embedded SQL interface, the programmer controls the extent of a transaction, which may include several queries or modifications, as well as operations performed in the host language. In the typical embedded SQL system, transactions begin as soon as operations on the database are executed and end with an explicit `COMMIT` or `ROLLBACK` ("abort") command.

As we shall discuss in Section 17.1.3, a transaction must execute atomically, that is, all-or-nothing and as if it were executed at an instant in time. Assuring that transactions are executed correctly is the job of a *transaction manager*, a subsystem that performs several functions, including:

1. Issuing signals to the log manager (described below) so that necessary information in the form of "log records" can be stored on the log.

2. Assuring that concurrently executing transactions do not interfere with each other in ways that introduce errors ("scheduling"; see Section 18.1).

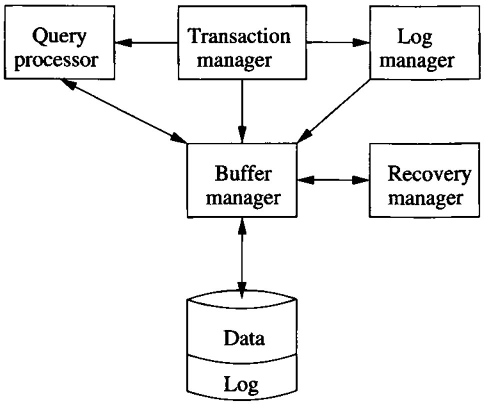

Figure 17.1: The log manager and transaction manager

The transaction manager and its interactions are suggested by Fig. 17.1. The transaction manager will send messages about actions of transactions to the log manager, to the buffer manager about when it is possible or necessary to copy the buffer back to disk, and to the query processor to execute the queries and other database operations that comprise the transaction.

The log manager maintains the log. It must deal with the buffer manager, since space for the log initially appears in main-memory buffers, and at certain times these buffers must be copied to disk. The log, as well as the data, occupies space on the disk, as we suggest in Fig. 17.1.

Finally, we show a recovery manager in Fig. 17.1. When there is a crash, the recovery manager is activated. It examines the log and uses it to repair the data, if necessary. As always, access to the disk is through the buffer manager.

17.1.3 Correct Execution of Transactions

Before we can deal with correcting system errors, we need to understand what it means for a transaction to be executed "correctly." To begin, we assume that the database is composed of "elements." We shall not specify precisely what an "element" is, except to say it has a value and can be accessed or modified by transactions. Different database systems use different notions of elements, but they are usually chosen from one or more of the following:

1. Relations.

2. Disk blocks or pages.

3. Individual tuples or objects.

In examples to follow, one can imagine that database elements are tuples, or in many examples, simply integers. However, there are several good reasons in practice to use choice (2) — disk blocks or pages — as the database element. In this way, buffer-contents become single elements, allowing us to avoid some serious problems with logging and transactions that we shall explore periodically as we learn various techniques. Avoiding database elements that are bigger than disk blocks also prevents a situation where part but not all of an element has been placed in nonvolatile storage when a crash occurs.

A database has a *state*, which is a value for each of its elements.[1] Intuitively, we regard certain states as *consistent*, and others as inconsistent. Consistent states satisfy all constraints of the database schema, such as key constraints and constraints on values. However, consistent states must also satisfy implicit constraints that are in the mind of the database designer. The implicit constraints may be maintained by triggers that are part of the database schema, but they might also be maintained only by policy statements concerning the database, or warnings associated with the user interface through which updates are made.

A fundamental assumption about transactions is:

- *The Correctness Principle*: If a transaction executes in the absence of any other transactions or system errors, and it starts with the database in a consistent state, then the database is also in a consistent state when the transaction ends.

There is a converse to the correctness principle that forms the motivation for both the logging techniques discussed in this chapter and the concurrency control mechanisms discussed in Chapter 18. This converse involves two points:

1. A transaction is *atomic*; that is, it must be executed as a whole or not at all. If only part of a transaction executes, then the resulting database state may not be consistent.

2. Transactions that execute simultaneously are likely to lead to an inconsistent state unless we take steps to control their interactions, as we shall in Chapter 18.

[1]We should not confuse the database state with the state of a transaction; the latter is values for the transaction's local variables, not database elements.

Is the Correctness Principle Believable?

Given that a database transaction could be an ad-hoc modification command issued at a terminal, perhaps by someone who doesn't understand the implicit constraints in the mind of the database designer, is it plausible to assume all transactions take the database from a consistent state to another consistent state? Explicit constraints are enforced by the database, so any transaction that violates them will be rejected by the system and not change the database at all. As for implicit constraints, one cannot characterize them exactly under any circumstances. Our position, justifying the correctness principle, is that if someone is given authority to modify the database, then they also have the authority to judge what the implicit constraints are.

17.1.4 The Primitive Operations of Transactions

Let us now consider in detail how transactions interact with the database. There are three address spaces that interact in important ways:

1. The space of disk blocks holding the database elements.
2. The virtual or main memory address space that is managed by the buffer manager.
3. The local address space of the transaction.

For a transaction to read a database element, that element must first be brought to a main-memory buffer or buffers, if it is not already there. Then, the contents of the buffer(s) can be read by the transaction into its own address space. Writing a new value for a database element by a transaction follows the reverse route. The new value is first created by the transaction in its own space. Then, this value is copied to the appropriate buffer(s).

The buffer may or may not be copied to disk immediately; that decision is the responsibility of the buffer manager in general. As we shall soon see, one of the principal tools for assuring resilience is forcing the buffer manager to write the block in a buffer back to disk at appropriate times. However, in order to reduce the number of disk I/O's, database systems can and will allow a change to exist only in volatile main-memory storage, at least for certain periods of time and under the proper set of conditions.

In order to study the details of logging algorithms and other transaction-management algorithms, we need a notation that describes all the operations that move data between address spaces. The primitives we shall use are:

1. `INPUT(X)`: Copy the disk block containing database element X to a memory buffer.

Buffers in Query Processing and in Transactions

If you got used to the analysis of buffer utilization in the chapters on query processing, you may notice a change in viewpoint here. In Chapters 15 and 16 we were interested in buffers principally as they were used to compute temporary relations during the evaluation of a query. That is one important use of buffers, but there is never a need to preserve a temporary value, so these buffers do not generally have their values logged. On the other hand, those buffers that hold data retrieved from the database *do* need to have those values preserved, especially when the transaction updates them.

2. `READ(X,t)`: Copy the database element X to the transaction's local variable t. More precisely, if the block containing database element X is not in a memory buffer then first execute `INPUT(X)`. Next, assign the value of X to local variable t.

3. `WRITE(X,t)`: Copy the value of local variable t to database element X in a memory buffer. More precisely, if the block containing database element X is not in a memory buffer then execute `INPUT(X)`. Next, copy the value of t to X in the buffer.

4. `OUTPUT(X)`: Copy the block containing X from its buffer to disk.

The above operations make sense as long as database elements reside within a single disk block, and therefore within a single buffer. If a database element occupies several blocks, we shall imagine that each block-sized portion of the element is an element by itself. The logging mechanism to be used will assure that the transaction cannot complete without the write of X being atomic; i.e., either all blocks of X are written to disk, or none are. Thus, we shall assume for the entire discussion of logging that

- A database element is no larger than a single block.

Different DBMS components issue the various commands we just introduced. `READ` and `WRITE` are issued by transactions. `INPUT` and `OUTPUT` are normally issued by the buffer manager. `OUTPUT` can also be initiated by the log manager under certain conditions, as we shall see.

Example 17.1: To see how the above primitive operations relate to what a transaction might do, let us consider a database that has two elements, A and B, with the constraint that they must be equal in all consistent states.[2]

Transaction T consists logically of the following two steps:

[2]One reasonably might ask why we should bother to have two different elements that are constrained to be equal, rather than maintaining only one element. However, this simple

```
A := A*2;
B := B*2;
```

If T starts in a consistent state (i.e., $A = B$) and completes its activities without interference from another transaction or system error, then the final state must also be consistent. That is, T doubles two equal elements to get new, equal elements.

Execution of T involves reading A and B from disk, performing arithmetic in the local address space of T, and writing the new values of A and B to their buffers. The relevant steps of T are thus:

```
READ(A,t); t := t*2; WRITE(A,t); READ(B,t); t := t*2; WRITE(B,t);
```

In addition, the buffer manager will eventually execute the OUTPUT steps to write these buffers back to disk. Figure 17.2 shows the primitive steps of T, followed by the two OUTPUT commands from the buffer manager. We assume that initially $A = B = 8$. The values of the memory and disk copies of A and B and the local variable t in the address space of transaction T are indicated for each step.

Action	t	Mem A	Mem B	Disk A	Disk B
READ(A,t)	8	8		8	8
t := t*2	16	8		8	8
WRITE(A,t)	16	16		8	8
READ(B,t)	8	16	8	8	8
t := t*2	16	16	8	8	8
WRITE(B,t)	16	16	16	8	8
OUTPUT(A)	16	16	16	16	8
OUTPUT(B)	16	16	16	16	16

Figure 17.2: Steps of a transaction and its effect on memory and disk

At the first step, T reads A, which generates an INPUT(A) command for the buffer manager if A's block is not already in a buffer. The value of A is also copied by the READ command into local variable t of T's address space. The second step doubles t; it has no affect on A, either in a buffer or on disk. The third step writes t into A of the buffer; it does not affect A on disk. The next three steps do the same for B, and the last two steps copy A and B to disk.

Observe that as long as all these steps execute, consistency of the database is preserved. If a system error occurs before OUTPUT(A) is executed, then there is no effect to the database stored on disk; it is as if T never ran, and consistency is preserved. However, if there is a system error after OUTPUT(A) but before

numerical constraint captures the spirit of many more realistic constraints, e.g., the number of seats sold on a flight must not exceed the number of seats on the plane by more than 10%, or the sum of the loan balances at a bank must equal the total debt of the bank.

`OUTPUT(B)`, then the database is left in an inconsistent state. We cannot prevent this situation from ever occurring, but we can arrange that when it does occur, the problem can be repaired — either both A and B will be reset to 8, or both will be advanced to 16. □

17.1.5 Exercises for Section 17.1

Exercise 17.1.1: Suppose that the consistency constraint on the database is $0 \leq A \leq B$. Tell whether each of the following transactions preserves consistency.

a) `A := A+B; B := A+B;`

b) `B := A+B; A := A+B;`

c) `A := B+1; B := A+1;`

Exercise 17.1.2: For each of the transactions of Exercise 17.1.1, add the read- and write-actions to the computation and show the effect of the steps on main memory and disk. Assume that initially $A = 5$ and $B = 10$. Also, tell whether it is possible, with the appropriate order of `OUTPUT` actions, to assure that consistency is preserved even if there is a crash while the transaction is executing.

17.2 Undo Logging

A *log* is a file of *log records*, each telling something about what some transaction has done. If log records appear in nonvolatile storage, we can use them to restore the database to a consistent state after a system crash. Our first style of logging — *undo logging* — makes repairs to the database state by undoing the effects of transactions that may not have completed before the crash.

Additionally, in this section we introduce the basic idea of log records, including the *commit* (successful completion of a transaction) action and its effect on the database state and log. We shall also consider how the log itself is created in main memory and copied to disk by a "flush-log" operation. Finally, we examine the undo log specifically, and learn how to use it in recovery from a crash. In order to avoid having to examine the entire log during recovery, we introduce the idea of "checkpointing," which allows old portions of the log to be thrown away.

17.2.1 Log Records

Imagine the log as a file opened for appending only. As transactions execute, the *log manager* has the job of recording in the log each important event. One block of the log at a time is filled with log records, each representing one of these events. Log blocks are initially created in main memory and are allocated

Why Might a Transaction Abort?

One might wonder why a transaction would abort rather than commit. There are actually several reasons. The simplest is when there is some error condition in the code of the transaction itself, e.g., an attempted division by zero. The DBMS may also abort a transaction for one of several reasons. For instance, a transaction may be involved in a deadlock, where it and one or more other transactions each hold some resource that the other needs. Then, one or more transactions must be forced by the system to abort (see Section 19.2).

by the buffer manager like any other blocks that the DBMS needs. The log blocks are written to nonvolatile storage on disk as soon as is feasible; we shall have more to say about this matter in Section 17.2.2.

There are several forms of log record that are used with each of the types of logging we discuss in this chapter. These are:

1. `<START` T`>`: This record indicates that transaction T has begun.

2. `<COMMIT` T`>`: Transaction T has completed successfully and will make no more changes to database elements. Any changes to the database made by T should appear on disk. However, because we cannot control when the buffer manager chooses to copy blocks from memory to disk, we cannot in general be sure that the changes are already on disk when we see the `<COMMIT` T`>` log record. If we insist that the changes already be on disk, this requirement must be enforced by the log manager (as is the case for undo logging).

3. `<ABORT` T`>`: Transaction T could not complete successfully. If transaction T aborts, no changes it made can have been copied to disk, and it is the job of the transaction manager to make sure that such changes never appear on disk, or that their effect on disk is cancelled if they do. We shall discuss the matter of repairing the effect of aborted transactions in Section 19.1.1.

For an undo log, the only other kind of log record we need is an *update record*, which is a triple $<T, X, v>$. The meaning of this record is: transaction T has changed database element X, and its former value was v. The change reflected by an update record normally occurs in memory, not disk; i.e., the log record is a response to a `WRITE` action into memory, not an `OUTPUT` action to disk. Notice also that an undo log does not record the new value of a database element, only the old value. As we shall see, should recovery be necessary in a system using undo logging, the only thing the recovery manager will do is cancel the possible effect of a transaction on disk by restoring the old value.

Preview of Other Logging Methods

In "redo logging" (Section 17.3), on recovery we redo any transaction that has a `COMMIT` record, and we ignore all others. Rules for redo logging assure that we may ignore transactions whose `COMMIT` records never reached the log on disk. "Undo/redo logging" (Section 17.4) will, on recovery, undo any transaction that has not committed, and will redo those transactions that have committed. Again, log-management and buffering rules will assure that these steps successfully repair any damage to the database.

17.2.2 The Undo-Logging Rules

An undo log is sufficient to allow recovery from a system failure, provided transactions and the buffer manager obey two rules:

U_1: If transaction T modifies database element X, then the log record of the form $<T, X, v>$ must be written to disk *before* the new value of X is written to disk.

U_2: If a transaction commits, then its `COMMIT` log record must be written to disk only *after* all database elements changed by the transaction have been written to disk, but as soon thereafter as possible.

To summarize rules U_1 and U_2, material associated with one transaction must be written to disk in the following order:

a) The log records indicating changed database elements.

b) The changed database elements themselves.

c) The `COMMIT` log record.

However, the order of (a) and (b) applies to each database element individually, not to the group of update records for a transaction as a whole.

In order to force log records to disk, the log manager needs a *flush-log* command that tells the buffer manager to copy to disk any log blocks that have not previously been copied to disk or that have been changed since they were last copied. In sequences of actions, we shall show `FLUSH LOG` explicitly. The transaction manager also needs to have a way to tell the buffer manager to perform an `OUTPUT` action on a database element. We shall continue to show the `OUTPUT` action in sequences of transaction steps.

Example 17.2: Let us reconsider the transaction of Example 17.1 in the light of undo logging. Figure 17.3 expands on Fig. 17.2 to show the log entries and flush-log actions that have to take place along with the actions of the transaction

Step	Action	t	M-A	M-B	D-A	D-B	Log
1)							<START T>
2)	READ(A,t)	8	8		8	8	
3)	t := t*2	16	8		8	8	
4)	WRITE(A,t)	16	16		8	8	<$T, A, 8$>
5)	READ(B,t)	8	16	8	8	8	
6)	t := t*2	16	16	8	8	8	
7)	WRITE(B,t)	16	16	16	8	8	<$T, B, 8$>
8)	FLUSH LOG						
9)	OUTPUT(A)	16	16	16	16	8	
10)	OUTPUT(B)	16	16	16	16	16	
11)							<COMMIT T>
12)	FLUSH LOG						

Figure 17.3: Actions and their log entries

T. Note we have shortened the headers to M-A for "the copy of A in a memory buffer" or D-B for "the copy of B on disk," and so on.

In line (1) of Fig. 17.3, transaction T begins. The first thing that happens is that the <START T> record is written to the log. Line (2) represents the read of A by T. Line (3) is the local change to t, which affects neither the database stored on disk nor any portion of the database in a memory buffer. Neither lines (2) nor (3) require any log entry, since they have no affect on the database.

Line (4) is the write of the new value of A to the buffer. This modification to A is reflected by the log entry <$T, A, 8$> which says that A was changed by T and its former value was 8. Note that the new value, 16, is not mentioned in an undo log.

Lines (5) through (7) perform the same three steps with B instead of A. At this point, T has completed and must commit. The changed A and B must migrate to disk, but in order to follow the two rules for undo logging, there is a fixed sequence of events that must happen.

First, A and B cannot be copied to disk until the log records for the changes are on disk. Thus, at step (8) the log is flushed, assuring that these records appear on disk. Then, steps (9) and (10) copy A and B to disk. The transaction manager requests these steps from the buffer manager in order to commit T.

Now, it is possible to commit T, and the <COMMIT T> record is written to the log, which is step (11). Finally, we must flush the log again at step (12) to make sure that the <COMMIT T> record of the log appears on disk. Notice that without writing this record to disk, we could have a situation where a transaction has committed, but for a long time a review of the log does not tell us that it has committed. That situation could cause strange behavior if there were a crash, because, as we shall see in Section 17.2.3, a transaction that appeared to the user to have completed long ago would then be undone and effectively aborted. □

Background Activity Affects the Log and Buffers

As we look at a sequence of actions and log entries like Fig. 17.3, it is tempting to imagine that these actions occur in isolation. However, the DBMS may be processing many transactions simultaneously. Thus, the four log records for transaction T may be interleaved on the log with records for other transactions. Moreover, if one of these transactions flushes the log, then the log records from T may appear on disk earlier than is implied by the flush-log actions of Fig. 17.3. There is no harm if log records reflecting a database modification appear earlier than necessary. The essential policy for undo logging is that we don't write the `<COMMIT` T`>` record until the `OUTPUT` actions for T are completed.

A trickier situation occurs if two database elements A and B share a block. Then, writing one of them to disk writes the other as well. In the worst case, we can violate rule U_1 by writing one of these elements prematurely. It may be necessary to adopt additional constraints on transactions in order to make undo logging work. For instance, we might use a locking scheme where database elements are disk blocks, as described in Section 18.3, to prevent two transactions from accessing the same block at the same time. This and other problems that appear when database elements are fractions of a block motivate our suggestion that blocks *be* the database elements.

17.2.3 Recovery Using Undo Logging

Suppose now that a system failure occurs. It is possible that certain database changes made by a given transaction were written to disk, while other changes made by the same transaction never reached the disk. If so, the transaction was not executed atomically, and there may be an inconsistent database state. The *recovery manager* must use the log to restore the database to some consistent state.

In this section we consider only the simplest form of recovery manager, one that looks at the entire log, no matter how long, and makes database changes as a result of its examination. In Section 17.2.4 we consider a more sensible approach, where the log is periodically "checkpointed," to limit the distance back in history that the recovery manager must go.

The first task of the recovery manager is to divide the transactions into committed and uncommitted transactions. If there is a log record `<COMMIT` T`>`, then by undo rule U_2 all changes made by transaction T were previously written to disk. Thus, T by itself could not have left the database in an inconsistent state when the system failure occurred.

However, suppose that we find a `<START` T`>` record on the log but no `<COMMIT` T`>` record. Then there could have been some changes to the database made by T that were written to disk before the crash, while other changes by T either were not made, or were made in the main-memory buffers but not copied to disk. In this case, T is an *incomplete transaction* and must be *undone*. That is, whatever changes T made must be reset to their previous value. Fortunately, rule U_1 assures us that if T changed X on disk before the crash, then there will be a $<T, X, v>$ record on the log, and that record will have been copied to disk before the crash. Thus, during the recovery, we must write the value v for database element X. Note that this rule begs the question whether X had value v in the database anyway; we don't even bother to check.

Since there may be several uncommitted transactions in the log, and there may even be several uncommitted transactions that modified X, we have to be systematic about the order in which we restore values. Thus, the recovery manager must scan the log from the end (i.e., from the most recently written record to the earliest written). As it travels, it remembers all those transactions T for which it has seen a <COMMIT T> record or an <ABORT T> record. Also as it travels backward, if it sees a record $<T, X, v>$, then:

1. If T is a transaction whose COMMIT record has been seen, then do nothing. T is committed and must not be undone.

2. Otherwise, T is an incomplete transaction, or an aborted transaction. The recovery manager must change the value of X in the database to v, in case X had been altered just before the crash.

After making these changes, the recovery manager must write a log record <ABORT T> for each incomplete transaction T that was not previously aborted, and then flush the log. Now, normal operation of the database may resume, and new transactions may begin executing.

Example 17.3: Let us consider the sequence of actions from Fig. 17.3 and Example 17.2. There are several different times that the system crash could have occurred; let us consider each significantly different one.

1. The crash occurs after step (12). Then the <COMMIT T> record reached disk before the crash. When we recover, we do not undo the results of T, and all log records concerning T are ignored by the recovery manager.

2. The crash occurs between steps (11) and (12). It is possible that the log record containing the COMMIT got flushed to disk; for instance, the buffer manager may have needed the buffer containing the end of the log for another transaction, or some other transaction may have asked for a log flush. If so, then the recovery is the same as in case (1) as far as T is concerned. However, if the COMMIT record never reached disk, then the recovery manager considers T incomplete. When it scans the log backward, it comes first to the record $<T, B, 8>$. It therefore stores 8 as the value of B on disk. It then comes to the record $<T, A, 8>$ and makes A have value 8 on disk. Finally, the record <ABORT T> is written to the log, and the log is flushed.

Crashes During Recovery

Suppose the system again crashes while we are recovering from a previous crash. Because of the way undo-log records are designed, giving the old value rather than, say, the change in the value of a database element, the recovery steps are *idempotent*; that is, repeating them many times has exactly the same effect as performing them once. We already observed that if we find a record $<T, X, v>$, it does not matter whether the value of X is already v — we may write v for X regardless. Similarly, if we repeat the recovery process, it does not matter whether the first recovery attempt restored some old values; we simply restore them again. The same reasoning holds for the other logging methods we discuss in this chapter. Since the recovery operations are idempotent, we can recover a second time without worrying about changes made the first time.

3. The crash occurs between steps (10) and (11). Now, the `COMMIT` record surely was not written, so T is incomplete and is undone as in case (2).

4. The crash occurs between steps (8) and (10). Again, T is undone. In this case the change to A and/or B may not have reached disk. Nevertheless, the proper value, 8, is restored for each of these database elements.

5. The crash occurs prior to step (8). Now, it is not certain whether any of the log records concerning T have reached disk. However, we know by rule U_1 that if the change to A and/or B reached disk, then the corresponding log record reached disk. Therefore if there were changes to A and/or B made on disk by T, then the corresponding log record will cause the recovery manager to undo those changes.

□

17.2.4 Checkpointing

As we observed, recovery requires that the entire log be examined, in principle. When logging follows the undo style, once a transaction has its `COMMIT` log record written to disk, the log records of that transaction are no longer needed during recovery. We might imagine that we could delete the log prior to a `COMMIT`, but sometimes we cannot. The reason is that often many transactions execute at once. If we truncated the log after one transaction committed, log records pertaining to some other active transaction T might be lost and could not be used to undo T if recovery were necessary.

The simplest way to untangle potential problems is to *checkpoint* the log periodically. In a simple checkpoint, we:

1. Stop accepting new transactions.
2. Wait until all currently active transactions commit or abort and have written a `COMMIT` or `ABORT` record on the log.
3. Flush the log to disk.
4. Write a log record <`CKPT`>, and flush the log again.
5. Resume accepting transactions.

Any transaction that executed prior to the checkpoint will have finished, and by rule U_2 its changes will have reached the disk. Thus, there will be no need to undo any of these transactions during recovery. During a recovery, we scan the log backwards from the end, identifying incomplete transactions as in Section 17.2.3. However, when we find a <`CKPT`> record, we know that we have seen all the incomplete transactions. Since no transactions may begin until the checkpoint ends, we must have seen every log record pertaining to the incomplete transactions already. Thus, there is no need to scan prior to the <`CKPT`>, and in fact the log before that point can be deleted or overwritten safely.

Example 17.4: Suppose the log begins:

$$
\begin{array}{l}
<\texttt{START}\ T_1> \\
<T_1, A, 5> \\
<\texttt{START}\ T_2> \\
<T_2, B, 10>
\end{array}
$$

At this time, we decide to do a checkpoint. Since T_1 and T_2 are the active (incomplete) transactions, we shall have to wait until they complete before writing the <`CKPT`> record on the log.

A possible extension of the log is shown in Fig. 17.4. Suppose a crash occurs at this point. Scanning the log from the end, we identify T_3 as the only incomplete transaction, and restore E and F to their former values 25 and 30, respectively. When we reach the <`CKPT`> record, we know there is no need to examine prior log records and the restoration of the database state is complete. □

17.2.5 Nonquiescent Checkpointing

A problem with the checkpointing technique described in Section 17.2.4 is that effectively we must shut down the system while the checkpoint is being made. Since the active transactions may take a long time to commit or abort, the system may appear to users to be stalled. Thus, a more complex technique known as *nonquiescent checkpointing*, which allows new transactions to enter the system during the checkpoint, is usually preferred. The steps in a nonquiescent checkpoint are:

```
<START $T_1$>
<$T_1, A, 5$>
<START $T_2$>
<$T_2, B, 10$>
<$T_2, C, 15$>
<$T_1, D, 20$>
<COMMIT $T_1$>
<COMMIT $T_2$>
<CKPT>
<START $T_3$>
<$T_3, E, 25$>
<$T_3, F, 30$>
```

Figure 17.4: An undo log

1. Write a log record <START CKPT ($T_1, \ldots, T_k$)> and flush the log. Here, $T_1, \ldots, T_k$ are the names or identifiers for all the *active* transactions (i.e., transactions that have not yet committed and written their changes to disk).

2. Wait until all of $T_1, \ldots, T_k$ commit or abort, but do not prohibit other transactions from starting.

3. When all of $T_1, \ldots, T_k$ have completed, write a log record <END CKPT> and flush the log.

With a log of this type, we can recover from a system crash as follows. As usual, we scan the log from the end, finding all incomplete transactions as we go, and restoring old values for database elements changed by these transactions. There are two cases, depending on whether, scanning backwards, we first meet an <END CKPT> record or a <START CKPT ($T_1, \ldots, T_k$)> record.

- If we first meet an <END CKPT> record, then we know that all incomplete transactions began after the previous <START CKPT ($T_1, \ldots, T_k$)> record. We may thus scan backwards as far as the next START CKPT, and then stop; previous log is useless and may as well have been discarded.

- If we first meet a record <START CKPT ($T_1, \ldots, T_k$)>, then the crash occurred during the checkpoint. However, the only incomplete transactions are those we met scanning backwards before we reached the START CKPT and those of $T_1, \ldots, T_k$ that did not complete before the crash. Thus, we need scan no further back than the start of the earliest of these incomplete transactions. The previous START CKPT record is certainly prior to any of these transaction starts, but often we shall find the starts of the

Finding the Last Log Record

It is common to recycle blocks of the log file on disk, since checkpoints allow us to drop old portions of the log. However, if we overwrite old log records, then we need to keep a serial number, which may only increase, as suggested by:

$\cancel{1}$ 9	$\cancel{2}$ 10	$\cancel{3}$ 11	4	5	6	7	8

Then, we can find the record whose serial number is greater than that of the next record; the latter record will be the current end of the log, and the entire log is found by ordering the current records by their present serial numbers.

In practice, a large log may be composed of many files, with a "top" file whose records indicate the files that comprise the log. Then, to recover, we find the last record of the top file, go to the file indicated, and find the last record there.

incomplete transactions long before we reach the previous checkpoint.[3] Moreover, if we use pointers to chain together the log records that belong to the same transaction, then we need not search the whole log for records belonging to active transactions; we just follow their chains back through the log.

As a general rule, once an `<END CKPT>` record has been written to disk, we can delete the log prior to the previous `START CKPT` record.

Example 17.5: Suppose that, as in Example 17.4, the log begins:

$$\begin{array}{l}<\texttt{START}\ T_1>\\ <T_1, A, 5>\\ <\texttt{START}\ T_2>\\ <T_2, B, 10>\end{array}$$

Now, we decide to do a nonquiescent checkpoint. Since T_1 and T_2 are the active (incomplete) transactions at this time, we write a log record

$$<\texttt{START CKPT}\ (T_1, T_2)>$$

Suppose that while waiting for T_1 and T_2 to complete, another transaction, T_3, initiates. A possible continuation of the log is shown in Fig. 17.5.

Suppose that at this point there is a system crash. Examining the log from the end, we find that T_3 is an incomplete transaction and must be undone.

[3]Notice, however, that because the checkpoint is nonquiescent, one of the incomplete transactions could have begun between the start and end of the previous checkpoint.

```
<START T1>
<T1, A, 5>
<START T2>
<T2, B, 10>
<START CKPT (T1, T2)>
<T2, C, 15>
<START T3>
<T1, D, 20>
<COMMIT T1>
<T3, E, 25>
<COMMIT T2>
<END CKPT>
<T3, F, 30>
```

Figure 17.5: An undo log using nonquiescent checkpointing

The final log record tells us to restore database element F to the value 30. When we find the <END CKPT> record, we know that all incomplete transactions began after the previous START CKPT. Scanning further back, we find the record <$T_3, E, 25$>, which tells us to restore E to value 25. Between that record, and the START CKPT there are no other transactions that started but did not commit, so no further changes to the database are made.

```
<START T1>
<T1, A, 5>
<START T2>
<T2, B, 10>
<START CKPT (T1, T2)>
<T2, C, 15>
<START T3>
<T1, D, 20>
<COMMIT T1>
<T3, E, 25>
```

Figure 17.6: Undo log with a system crash during checkpointing

Now suppose the crash occurs during the checkpoint, and the end of the log after the crash is as shown in Fig. 17.6. Scanning backwards, we identify T_3 and then T_2 as incomplete transactions and undo changes they have made. When we find the <START CKPT (T_1, T_2)> record, we know that the only other possible incomplete transaction is T_1. However, we have already scanned the <COMMIT T_1> record, so we know that T_1 is *not* incomplete. Also, we have already seen the <START T_3> record. Thus, we need only to continue backwards until we meet the START record for T_2, restoring database element B to value

10 as we go. □

17.2.6 Exercises for Section 17.2

Exercise 17.2.1: Show the undo-log records for each of the transactions (call each T) of Exercise 17.1.1, assuming that initially $A = 5$ and $B = 10$.

Exercise 17.2.2: For each of the sequences of log records representing the actions of one transaction T, tell all the sequences of events that are legal according to the rules of undo logging, where the events of interest are the writing to disk of the blocks containing database elements, and the blocks of the log containing the update and commit records. You may assume that log records are written to disk in the order shown; i.e., it is not possible to write one log record to disk while a previous record is not written to disk.

a) <START T>; <T, A, 10>; <T, B, 20>; <COMMIT T>;

b) <START T>; <T, A, 10>; <T, B, 20>; <T, C, 30><COMMIT T>;

! **Exercise 17.2.3:** The pattern introduced in Exercise 17.2.2 can be extended to a transaction that writes new values for n database elements. How many legal sequences of events are there for such a transaction, if the undo-logging rules are obeyed?

Exercise 17.2.4: The following is a sequence of undo-log records written by two transactions T and U: <START T>; <T, A, 10>; <START U>; <U, B, 20>; <T, C, 30>; <U, D, 40>; <COMMIT U>; <T, E, 50>; <COMMIT T>. Describe the action of the recovery manager, including changes to both disk and the log, if there is a crash and the last log record to appear on disk is:

(a) <START U> (b) <COMMIT U> (c) <T, E, 50> (d) <COMMIT T>.

Exercise 17.2.5: For each of the situations described in Exercise 17.2.4, what values written by T and U *must* appear on disk? Which values *might* appear on disk?

! **Exercise 17.2.6:** Suppose that the transaction U in Exercise 17.2.4 is changed so that the record <U, D, 40> becomes <U, A, 40>. What is the effect on the disk value of A if there is a crash at some point during the sequence of events? What does this example say about the ability of logging by itself to preserve atomicity of transactions?

Exercise 17.2.7: Consider the following sequence of log records: <START S>; <S, A, 60>; <COMMIT S>; <START T>; <T, A, 10>; <START U>; <U, B, 20>; <T, C, 30>; <START V>; <U, D, 40>; <V, F, 70>; <COMMIT U>; <T, E, 50>; <COMMIT T>; <V, B, 80>; <COMMIT V>. Suppose that we begin a nonquiescent checkpoint immediately after one of the following log records has been written (in memory):

(a) <$S, A, 60$> (b) <$T, A, 10$> (c) <$U, B, 20$>
(d) <$U, D, 40$> (e) <$T, E, 50$>

For each, tell:

i. When the <`END CKPT`> record is written, and

ii. For each possible point at which a crash could occur, how far back in the log we must look to find all possible incomplete transactions.

17.3 Redo Logging

Undo logging has a potential problem that we cannot commit a transaction without first writing all its changed data to disk. Sometimes, we can save disk I/O's if we let changes to the database reside only in main memory for a while. As long as there is a log to fix things up in the event of a crash, it is safe to do so.

The requirement for immediate backup of database elements to disk can be avoided if we use a logging mechanism called *redo logging*. The principal differences between redo and undo logging are:

1. While undo logging cancels the effect of incomplete transactions and ignores committed ones during recovery, redo logging ignores incomplete transactions and repeats the changes made by committed transactions.

2. While undo logging requires us to write changed database elements to disk before the `COMMIT` log record reaches disk, redo logging requires that the `COMMIT` record appear on disk before any changed values reach disk.

3. While the old values of changed database elements are exactly what we need to recover when the undo rules U_1 and U_2 are followed, to recover using redo logging, we need the new values instead.

17.3.1 The Redo-Logging Rule

In redo logging the meaning of a log record <T, X, v> is "transaction T wrote new value v for database element X." There is no indication of the old value of X in this record. Every time a transaction T modifies a database element X, a record of the form <T, X, v> must be written to the log.

For redo logging, the order in which data and log entries reach disk can be described by a single "redo rule," called the *write-ahead logging rule*.

R_1: Before modifying any database element X on disk, it is necessary that all log records pertaining to this modification of X, including both the update record <T, X, v> and the <`COMMIT` T> record, must appear on disk.

The COMMIT record for a transaction can only be written to the log when the transaction completes, so the commit record must follow all the update log records. Thus, when redo logging is in use, the order in which material associated with one transaction gets written to disk is:

1. The log records indicating changed database elements.

2. The COMMIT log record.

3. The changed database elements themselves.

Example 17.6: Let us consider the same transaction T as in Example 17.2. Figure 17.7 shows a possible sequence of events for this transaction.

Step	Action	t	M-A	M-B	D-A	D-B	Log
1)							<START T>
2)	READ(A,t)	8	8		8	8	
3)	t := t*2	16	8		8	8	
4)	WRITE(A,t)	16	16		8	8	<$T, A, 16$>
5)	READ(B,t)	8	16	8	8	8	
6)	t := t*2	16	16	8	8	8	
7)	WRITE(B,t)	16	16	16	8	8	<$T, B, 16$>
8)							<COMMIT T>
9)	FLUSH LOG						
10)	OUTPUT(A)	16	16	16	16	8	
11)	OUTPUT(B)	16	16	16	16	16	

Figure 17.7: Actions and their log entries using redo logging

The major differences between Figs. 17.7 and 17.3 are as follows. First, we note in lines (4) and (7) of Fig. 17.7 that the log records reflecting the changes have the new values of A and B, rather than the old values. Second, we see that the <COMMIT T> record comes earlier, at step (8). Then, the log is flushed, so all log records involving the changes of transaction T appear on disk. Only then can the new values of A and B be written to disk. We show these values written immediately, at steps (10) and (11), although in practice they might occur later. □

17.3.2 Recovery With Redo Logging

An important consequence of the redo rule R_1 is that unless the log has a <COMMIT T> record, we know that no changes to the database made by transaction T have been written to disk. Thus, incomplete transactions may be treated during recovery as if they had never occurred. However, the committed transactions present a problem, since we do not know which of their database changes have been written to disk. Fortunately, the redo log has exactly the

Order of Redo Matters

Since several committed transactions may have written new values for the same database element X, we have required that during a redo recovery, we scan the log from earliest to latest. Thus, the final value of X in the database will be the one written last, as it should be. Similarly, when describing undo recovery, we required that the log be scanned from latest to earliest. Thus, the final value of X will be the value that it had before any of the incomplete transactions changed it.

However, if the DBMS enforces atomicity, then we would not expect to find, in an undo log, two uncommitted transactions, each of which had written the same database element. In contrast, with redo logging we focus on the committed transactions, as these need to be redone. It is quite normal for there to be two *committed* transactions, each of which changed the same database element at different times. Thus, order of redo is always important, while order of undo might not be if the right kind of concurrency control were in effect.

information we need: the new values, which we may write to disk regardless of whether they were already there. To recover, using a redo log, after a system crash, we do the following.

1. Identify the committed transactions.

2. Scan the log forward from the beginning. For each log record $<T, X, v>$ encountered:

   (a) If T is not a committed transaction, do nothing.

   (b) If T is committed, write value v for database element X.

3. For each incomplete transaction T, write an `<ABORT` T`>` record to the log and flush the log.

Example 17.7: Let us consider the log written in Fig. 17.7 and see how recovery would be performed if the crash occurred after different steps in that sequence of actions.

1. If the crash occurs any time after step (9), then the `<COMMIT` T`>` record has been flushed to disk. The recovery system identifies T as a committed transaction. When scanning the log forward, the log records $<T, A, 16>$ and $<T, B, 16>$ cause the recovery manager to write values 16 for A and B. Notice that if the crash occurred between steps (10) and (11), then the write of A is redundant, but the write of B had not occurred and

changing B to 16 is essential to restore the database state to consistency. If the crash occurred after step (11), then both writes are redundant but harmless.

2. If the crash occurs between steps (8) and (9), then although the record `<COMMIT` T`>` was written to the log, it may not have gotten to disk (depending on whether the log was flushed for some other reason). If it did get to disk, then the recovery proceeds as in case (1), and if it did not get to disk, then recovery is as in case (3), below.

3. If the crash occurs prior to step (8), then `<COMMIT` T`>` surely has not reached disk. Thus, T is treated as an incomplete transaction. No changes to A or B on disk are made on behalf of T, and eventually an `<ABORT` T`>` record is written to the log.

□

17.3.3 Checkpointing a Redo Log

Redo logs present a checkpointing problem that we do not see with undo logs. Since the database changes made by a committed transaction can be copied to disk much later than the time at which the transaction commits, we cannot limit our concern to transactions that are active at the time we decide to create a checkpoint. Regardless of whether the checkpoint is quiescent or nonquiescent, between the start and end of the checkpoint we must write to disk all database elements that have been modified by committed transactions. To do so requires that the buffer manager keep track of which buffers are *dirty*, that is, they have been changed but not written to disk. It is also required to know which transactions modified which buffers.

On the other hand, we can complete the checkpoint without waiting for the active transactions to commit or abort, since they are not allowed to write their pages to disk at that time anyway. The steps to perform a nonquiescent checkpoint of a redo log are as follows:

1. Write a log record `<START CKPT` $(T_1, \ldots, T_k)$`>`, where $T_1, \ldots, T_k$ are all the active (uncommitted) transactions, and flush the log.

2. Write to disk all database elements that were written to buffers but not yet to disk by transactions that had already committed when the `START CKPT` record was written to the log.

3. Write an `<END CKPT>` record to the log and flush the log.

Example 17.8: Figure 17.8 shows a possible redo log, in the middle of which a checkpoint occurs. When we start the checkpoint, only T_2 is active, but the value of A written by T_1 may have reached disk. If not, then we must copy A

<START T_1>
<$T_1, A, 5$>
<START T_2>
<COMMIT T_1>
<$T_2, B, 10$>
<START CKPT (T_2)>
<$T_2, C, 15$>
<START T_3>
<$T_3, D, 20$>
<END CKPT>
<COMMIT T_2>
<COMMIT T_3>

Figure 17.8: A redo log

to disk before the checkpoint can end. We suggest the end of the checkpoint occurring after several other events have occurred: T_2 wrote a value for database element C, and a new transaction T_3 started and wrote a value of D. After the end of the checkpoint, the only things that happen are that T_2 and T_3 commit. □

17.3.4 Recovery With a Checkpointed Redo Log

As for an undo log, the insertion of records to mark the start and end of a checkpoint helps us limit our examination of the log when a recovery is necessary. Also as with undo logging, there are two cases, depending on whether the last checkpoint record is START or END.

Suppose first that the last checkpoint record on the log before a crash is <END CKPT>. Now, we know that every value written by a transaction that committed before the corresponding <START CKPT ($T_1, \ldots, T_k$)> has had its changes written to disk, so we need not concern ourselves with recovering the effects of these transactions. However, any transaction that is either among the T_i's or that started after the beginning of the checkpoint can still have changes it made not yet migrated to disk, even though the transaction has committed. Thus, we must perform recovery as described in Section 17.3.2, but may limit our attention to the transactions that are either one of the T_i's mentioned in the last <START CKPT ($T_1, \ldots, T_k$)> or that started after that log record appeared in the log. In searching the log, we do not have to look further back than the earliest of the <START T_i> records. Notice, however, that these START records could appear prior to any number of checkpoints. Linking backwards all the log records for a given transaction helps us to find the necessary records, as it did for undo logging.

Now, suppose the last checkpoint record on the log is

<START CKPT $(T_1, \ldots, T_k)$>

We cannot be sure that committed transactions prior to the start of this checkpoint had their changes written to disk. Thus, we must search back to the previous <END CKPT> record, find its matching <START CKPT $(S_1, \ldots, S_m)$> record,[4] and redo all those committed transactions that either started after that START CKPT or are among the S_i's.

Example 17.9: Consider again the log of Fig. 17.8. If a crash occurs at the end, we search backwards, finding the <END CKPT> record. We thus know that it is sufficient to consider as candidates to redo all those transactions that either started after the <START CKPT (T_2)> record was written or that are on its list (i.e., T_2). Thus, our candidate set is $\{T_2, T_3\}$. We find the records <COMMIT T_2> and <COMMIT T_3>, so we know that each must be redone. We search the log as far back as the <START T_2> record, and find the update records <$T_2, B, 10$>, <$T_2, C, 15$>, and <$T_3, D, 20$> for the committed transactions. Since we don't know whether these changes reached disk, we rewrite the values 10, 15, and 20 for B, C, and D, respectively.

Now, suppose the crash occurred between the records <COMMIT T_2> and <COMMIT T_3>. The recovery is similar to the above, except that T_3 is no longer a committed transaction. Thus, its change <$T_3, D, 20$> must *not* be redone, and no change is made to D during recovery, even though that log record is in the range of records that is examined. Also, we write an <ABORT T_3> record to the log after recovery.

Finally, suppose that the crash occurs just prior to the <END CKPT> record. In principal, we must search back to the next-to-last START CKPT record and get its list of active transactions. However, in this case there is no previous checkpoint, and we must go all the way to the beginning of the log. Thus, we identify T_1 as the only committed transaction, redo its action <$T_1, A, 5$>, and write records <ABORT T_2> and <ABORT T_3> to the log after recovery. □

Since transactions may be active during several checkpoints, it is convenient to include in the <START CKPT $(T_1, \ldots, T_k)$> records not only the names of the active transactions, but pointers to the place on the log where they started. By doing so, we know when it is safe to delete early portions of the log. When we write an <END CKPT>, we know that we shall never need to look back further than the earliest of the <START T_i> records for the active transactions T_i. Thus, anything prior to that START record may be deleted.

17.3.5 Exercises for Section 17.3

Exercise 17.3.1: Show the redo-log records for each of the transactions (call each T) of Exercise 17.1.1, assuming that initially $A = 5$ and $B = 10$.

[4]There is a small technicality that there could be a START CKPT record that, because of a previous crash, has no matching <END CKPT> record. Therefore, we must look not just for the previous START CKPT, but first for an <END CKPT> and then the previous START CKPT.

Exercise 17.3.2: Repeat Exercise 17.2.2 for redo logging.

Exercise 17.3.3: Repeat Exercise 17.2.4 for redo logging.

Exercise 17.3.4: Repeat Exercise 17.2.5 for redo logging.

Exercise 17.3.5: Using the data of Exercise 17.2.7, answer for each of the positions (a) through (e) of that exercise:

i. At what points could the `<END CKPT>` record be written, and

ii. For each possible point at which a crash could occur, how far back in the log we must look to find all possible incomplete transactions. Consider both the case that the `<END CKPT>` record was or was not written prior to the crash.

17.4 Undo/Redo Logging

We have seen two different approaches to logging, differentiated by whether the log holds old values or new values when a database element is updated. Each has certain drawbacks:

- Undo logging requires that data be written to disk immediately after a transaction finishes, perhaps increasing the number of disk I/O's that need to be performed.

- On the other hand, redo logging requires us to keep all modified blocks in buffers until the transaction commits and the log records have been flushed, perhaps increasing the average number of buffers required by transactions.

- Both undo and redo logs may put contradictory requirements on how buffers are handled during a checkpoint, unless the database elements are complete blocks or sets of blocks. For instance, if a buffer contains one database element A that was changed by a committed transaction and another database element B that was changed in the same buffer by a transaction that has not yet had its `COMMIT` record written to disk, then we are required to copy the buffer to disk because of A but also forbidden to do so, because rule R_1 applies to B.

We shall now see a kind of logging called *undo/redo logging*, that provides increased flexibility to order actions, at the expense of maintaining more information on the log.

17.4.1 The Undo/Redo Rules

An undo/redo log has the same sorts of log records as the other kinds of log, with one exception. The update log record that we write when a database element changes value has four components. Record $<T, X, v, w>$ means that transaction T changed the value of database element X; its former value was v, and its new value is w. The constraints that an undo/redo logging system must follow are summarized by the following rule:

UR_1 Before modifying any database element X on disk because of changes made by some transaction T, it is necessary that the update record $<T, X, v, w>$ appear on disk.

Rule UR_1 for undo/redo logging thus enforces only the constraints enforced by *both* undo logging and redo logging. In particular, the <COMMIT T> log record can precede or follow any of the changes to the database elements on disk.

Example 17.10: Figure 17.9 is a variation in the order of the actions associated with the transaction T that we last saw in Example 17.6. Notice that the log records for updates now have both the old and the new values of A and B. In this sequence, we have written the <COMMIT T> log record in the middle of the output of database elements A and B to disk. Step (10) could also have appeared before step (8) or step (9), or after step (11). □

Step	Action	t	M-A	M-B	D-A	D-B	Log
1)							<START T>
2)	`READ(A,t)`	8	8		8	8	
3)	`t := t*2`	16	8		8	8	
4)	`WRITE(A,t)`	16	16		8	8	$<T, A, 8, 16>$
5)	`READ(B,t)`	8	16	8	8	8	
6)	`t := t*2`	16	16	8	8	8	
7)	`WRITE(B,t)`	16	16	16	8	8	$<T, B, 8, 16>$
8)	`FLUSH LOG`						
9)	`OUTPUT(A)`	16	16	16	16	8	
10)							<COMMIT T>
11)	`OUTPUT(B)`	16	16	16	16	16	

Figure 17.9: A possible sequence of actions and their log entries using undo/redo logging

17.4.2 Recovery With Undo/Redo Logging

When we need to recover using an undo/redo log, we have the information in the update records either to undo a transaction T by restoring the old values of

A Problem With Delayed Commitment

Like undo logging, a system using undo/redo logging can exhibit a behavior where a transaction appears to the user to have been completed (e.g., they booked an airline seat over the Web and disconnected), and yet because the `<COMMIT` T`>` record was not flushed to disk, a subsequent crash causes the transaction to be undone rather than redone. If this possibility is a problem, we suggest the use of an additional rule for undo/redo logging:

UR_2 A `<COMMIT` T`>` record must be flushed to disk as soon as it appears in the log.

For instance, we would add `FLUSH LOG` after step (10) of Fig. 17.9.

the database elements that T changed, or to redo T by repeating the changes it has made. The undo/redo recovery policy is:

1. Redo all the committed transactions in the order earliest-first, and

2. Undo all the incomplete transactions in the order latest-first.

Notice that it is necessary for us to do both. Because of the flexibility allowed by undo/redo logging regarding the relative order in which `COMMIT` log records and the database changes themselves are copied to disk, we could have either a committed transaction with some or all of its changes not on disk, or an uncommitted transaction with some or all of its changes on disk.

Example 17.11: Consider the sequence of actions in Fig. 17.9. Here are the different ways that recovery would take place on the assumption that there is a crash at various points in the sequence.

1. Suppose the crash occurs after the `<COMMIT` T`>` record is flushed to disk. Then T is identified as a committed transaction. We write the value 16 for both A and B to the disk. Because of the actual order of events, A already has the value 16, but B may not, depending on whether the crash occurred before or after step (11).

2. If the crash occurs prior to the `<COMMIT` T`>` record reaching disk, then T is treated as an incomplete transaction. The previous values of A and B, 8 in each case, are written to disk. If the crash occurs between steps (9) and (10), then the value of A was 16 on disk, and the restoration to value 8 is necessary. In this example, the value of B does not need to be undone, and if the crash occurs before step (9) then neither does the value of A. However, in general we cannot be sure whether restoration is necessary, so we always perform the undo operation.

□

Strange Behavior of Transactions During Recovery

You may have noticed that we did not specify whether undo's or redo's are done first during recovery using an undo/redo log. In fact, whether we perform the redo's or undo's first, we are open to the following situation: a transaction T has committed and is redone. However, T read a value X written by some transaction U that has not committed and is undone. The problem is not whether we redo first, and leave X with its value prior to U, or we undo first and leave X with its value written by T. The situation makes no sense either way, because the final database state does not correspond to the effect of any sequence of atomic transactions.

In reality, the DBMS must do more than log changes. It must assure that such situations do not occur at all. In Chapter 18, there is a discussion about the means to isolate transactions like T and U, so the interaction between them through database element X cannot occur. In Section 19.1, we explicitly address means for preventing this situation where T reads a "dirty" value of X — one that has not been committed.

17.4.3 Checkpointing an Undo/Redo Log

A nonquiescent checkpoint is somewhat simpler for undo/redo logging than for the other logging methods. We have only to do the following:

1. Write a `<START CKPT` $(T_1, \dots, T_k)$`>` record to the log, where $T_1, \dots, T_k$ are all the active transactions, and flush the log.

2. Write to disk all the buffers that are *dirty*; i.e., they contain one or more changed database elements. Unlike redo logging, we flush all dirty buffers, not just those written by committed transactions.

3. Write an `<END CKPT>` record to the log, and flush the log.

Notice in connection with point (2) that, because of the flexibility undo/redo logging offers regarding when data reaches disk, we can tolerate the writing to disk of data written by incomplete transactions. Therefore we can tolerate database elements that are smaller than complete blocks and thus may share buffers. The only requirement we must make on transactions is:

- A transaction must not write any values (even to memory buffers) until it is certain not to abort.

As we shall see in Section 19.1, this constraint is almost certainly needed anyway, in order to avoid inconsistent interactions between transactions. Notice that under redo logging, the above condition is not sufficient, since even if the transaction that wrote B is certain to commit, rule R_1 requires that the transaction's `COMMIT` record be written to disk before B is written to disk.

Example 17.12: Figure 17.10 shows an undo/redo log analogous to the redo log of Fig. 17.8. We have changed only the update records, giving them an old value as well as a new value. For simplicity, we have assumed that in each case the old value is one less than the new value.

<START T_1>
<$T_1, A, 4, 5$>
<START T_2>
<COMMIT T_1>
<$T_2, B, 9, 10$>
<START CKPT (T_2)>
<$T_2, C, 14, 15$>
<START T_3>
<$T_3, D, 19, 20$>
<END CKPT>
<COMMIT T_2>
<COMMIT T_3>

Figure 17.10: An undo/redo log

As in Example 17.8, T_2 is identified as the only active transaction when the checkpoint begins. Since this log is an undo/redo log, it is possible that T_2's new B-value 10 has been written to disk, which was not possible under redo logging. However, it is irrelevant whether or not that disk write has occurred. During the checkpoint, we shall surely flush B to disk if it is not already there, since we flush all dirty buffers. Likewise, we shall flush A, written by the committed transaction T_1, if it is not already on disk.

If the crash occurs at the end of this sequence of events, then T_2 and T_3 are identified as committed transactions. Transaction T_1 is prior to the checkpoint. Since we find the <END CKPT> record on the log, T_1 is correctly assumed to have both completed and had its changes written to disk. We therefore redo both T_2 and T_3, as in Example 17.8, and ignore T_1. However, when we redo a transaction such as T_2, we do not need to look prior to the <START CKPT (T_2)> record, even though T_2 was active at that time, because we know that T_2's changes prior to the start of the checkpoint were flushed to disk during the checkpoint.

For another instance, suppose the crash occurs just before the <COMMIT T_3> record is written to disk. Then we identify T_2 as committed but T_3 as incomplete. We redo T_2 by setting C to 15 on disk; it is not necessary to set B to 10 since we know that change reached disk before the <END CKPT>. However, unlike the situation with a redo log, we also undo T_3; that is, we set D to 19 on disk. If T_3 had been active at the start of the checkpoint, we would have had to look prior to the START-CKPT record to find if there were more actions by T_3 that may have reached disk and need to be undone. □

17.4.4 Exercises for Section 17.4

Exercise 17.4.1: Show the undo/redo-log records for each of the transactions (call each T) of Exercise 17.1.1, assuming that initially $A = 5$ and $B = 10$.

Exercise 17.4.2: For each of the sequences of log records representing the actions of one transaction T, tell all the sequences of events that are legal according to the rules of undo/redo logging, where the events of interest are the writing to disk of the blocks containing database elements, and the blocks of the log containing the update and commit records. You may assume that log records are written to disk in the order shown; i.e., it is not possible to write one log record to disk while a previous record is not written to disk.

a) <START T>; <$T, A, 10, 11$>; <$T, B, 20, 21$>; <COMMIT T>;

b) <START T>; <$T, A, 10, 21$>; <$T, B, 20, 21$>; <$T, C, 30, 31$>; <COMMIT T>;

Exercise 17.4.3: The following is a sequence of undo/redo-log records written by two transactions T and U: <START T>; <$T, A, 10, 11$>; <START U>; <$U, B, 20, 21$>; <$T, C, 30, 31$>; <$U, D, 40, 41$>; <COMMIT U>; <$T, E, 50, 51$>; <COMMIT T>. Describe the action of the recovery manager, including changes to both disk and the log, if there is a crash and the last log record to appear on disk is:

(a) <START U> (b) <COMMIT U> (c) <$T, E, 50, 51$> (d) <COMMIT T>.

Exercise 17.4.4: For each of the situations described in Exercise 17.4.3, what values written by T and U *must* appear on disk? Which values *might* appear on disk?

Exercise 17.4.5: Consider the following sequence of log records: <START S>; <$S, A, 60, 61$>; <COMMIT S>; <START T>; <$T, A, 61, 62$>; <START U>; <$U, B, 20, 21$>; <$T, C, 30, 31$>; <START V>; <$U, D, 40, 41$>; <$V, F, 70, 71$>; <COMMIT U>; <$T, E, 50, 51$>; <COMMIT T>; <$V, B, 21, 22$>; <COMMIT V>. Suppose that we begin a nonquiescent checkpoint immediately after one of the following log records has been written (in memory):

(a) <$S, A, 60, 61$> (b) <$T, A, 61, 62$> (c) <$U, B, 20, 21$>
(d) <$U, D, 40, 41$> (e) <$T, E, 50, 51$>

For each, tell:

i. At what points could the <END CKPT> record be written, and

ii. For each possible point at which a crash could occur, how far back in the log we must look to find all possible incomplete transactions. Consider both the case that the <END CKPT> record was or was not written prior to the crash.

17.5 Protecting Against Media Failures

The log can protect us against system failures, where nothing is lost from disk, but temporary data in main memory is lost. However, as we discussed in Section 17.1.1, more serious failures involve the loss of one or more disks. An archiving system, which we cover next, is needed to enable a database to survive losses involving disk-resident data.

17.5.1 The Archive

To protect against media failures, we are thus led to a solution involving *archiving* — maintaining a copy of the database separate from the database itself. If it were possible to shut down the database for a while, we could make a backup copy on some storage medium such as tape or optical disk, and store the copy remote from the database, in some secure location. The backup would preserve the database state as it existed at the time of the backup, and if there were a media failure, the database could be restored to this state.

To advance to a more recent state, we could use the log, provided the log had been preserved since the archive copy was made, and the log itself survived the failure. In order to protect against losing the log, we could transmit a copy of the log, almost as soon as it is created, to the same remote site as the archive. Then, if the log as well as the data is lost, we can use the archive plus remotely stored log to recover, at least up to the point that the log was last transmitted to the remote site.

Since writing an archive is a lengthy process, we try to avoid copying the entire database at each archiving step. Thus, we distinguish between two levels of archiving:

1. A *full dump*, in which the entire database is copied.

2. An *incremental dump*, in which only those database elements changed since the previous full or incremental dump are copied.

It is also possible to have several levels of dump, with a full dump thought of as a "level 0" dump, and a "level i" dump copying everything changed since the last dump at a level less than or equal to i.

We can restore the database from a full dump and its subsequent incremental dumps, in a process much like the way a redo or undo/redo log can be used to repair damage due to a system failure. We copy the full dump back to the database, and then in an earliest-first order, make the changes recorded by the later incremental dumps.

17.5.2 Nonquiescent Archiving

The problem with the simple view of archiving in Section 17.5.1 is that most databases cannot be shut down for the period of time (possibly hours) needed

Why Not Just Back Up the Log?

We might question the need for an archive, since we have to back up the log in a secure place anyway if we are not to be stuck at the state the database was in when the previous archive was made. While it may not be obvious, the answer lies in the typical rate of change of a large database. While only a small fraction of the database may change in a day, the changes, each of which must be logged, will over the course of a year become much larger than the database itself. If we never archived, then the log could never be truncated, and the cost of storing the log would soon exceed the cost of storing a copy of the database.

to make a backup copy. We thus need to consider *nonquiescent archiving*, which is analogous to nonquiescent checkpointing. Recall that a nonquiescent checkpoint attempts to make a copy on the disk of the (approximate) database state that existed when the checkpoint started. We can rely on a small portion of the log around the time of the checkpoint to fix up any deviations from that database state, due to the fact that during the checkpoint, new transactions may have started and written to disk.

Similarly, a nonquiescent dump tries to make a copy of the database that existed when the dump began, but database activity may change many database elements on disk during the minutes or hours that the dump takes. If it is necessary to restore the database from the archive, the log entries made during the dump can be used to sort things out and get the database to a consistent state. The analogy is suggested by Fig. 17.11.

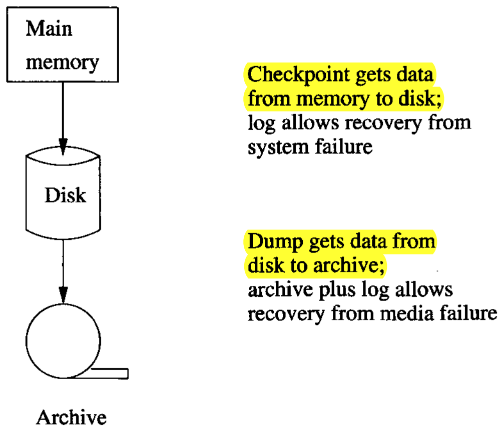
Main memory
Disk
Archive
Checkpoint gets data from memory to disk; log allows recovery from system failure
Dump gets data from disk to archive; archive plus log allows recovery from media failure

Figure 17.11: The analogy between checkpoints and dumps

A nonquiescent dump copies the database elements in some fixed order, possibly while those elements are being changed by executing transactions. As a result, the value of a database element that is copied to the archive may or may not be the value that existed when the dump began. As long as the log for the duration of the dump is preserved, the discrepancies can be corrected from the log.

Example 17.13: For a very simple example, suppose that our database consists of four elements, A, B, C, and D, which have the values 1 through 4, respectively, when the dump begins. During the dump, A is changed to 5, C is changed to 6, and B is changed to 7. However, the database elements are copied in order, and the sequence of events shown in Fig. 17.12 occurs. Then although the database at the beginning of the dump has values $(1,2,3,4)$, and the database at the end of the dump has values $(5,7,6,4)$, the copy of the database in the archive has values $(1,2,6,4)$, a database state that existed at no time during the dump. □

Disk	Archive
	Copy A
`A := 5`	
	Copy B
`C := 6`	
	Copy C
`B := 7`	
	Copy D

Figure 17.12: Events during a nonquiescent dump

In more detail, the process of making an archive can be broken into the following steps. We assume that the logging method is either redo or undo/redo; an undo log is not suitable for use with archiving.

1. Write a log record `<START DUMP>`.

2. Perform a checkpoint appropriate for whichever logging method is being used.

3. Perform a full or incremental dump of the data disk(s), as desired, making sure that the copy of the data has reached the secure, remote site.

4. Make sure that enough of the log has been copied to the secure, remote site that at least the prefix of the log up to and including the checkpoint in item (2) will survive a media failure of the database.

5. Write a log record `<END DUMP>`.

At the completion of the dump, it is safe to throw away log prior to the beginning of the checkpoint *previous* to the one performed in item (2) above.

Example 17.14: Suppose that the changes to the simple database in Example 17.13 were caused by two transactions T_1 (which writes A and B) and T_2 (which writes C) that were active when the dump began. Figure 17.13 shows a possible undo/redo log of the events during the dump.

```
<START DUMP>
<START CKPT (T1,T2)>
<T1,A,1,5>
<T2,C,3,6>
<COMMIT T2>
<T1,B,2,7>
<END CKPT>
Dump completes
<END DUMP>
```

Figure 17.13: Log taken during a dump

Notice that we did not show T_1 committing. It would be unusual that a transaction remained active during the entire time a full dump was in progress, but that possibility doesn't affect the correctness of the recovery method that we discuss next. □

17.5.3 Recovery Using an Archive and Log

Suppose that a media failure occurs, and we must reconstruct the database from the most recent archive and whatever prefix of the log has reached the remote site and has not been lost in the crash. We perform the following steps:

1. Restore the database from the archive.
   (a) Find the most recent full dump and reconstruct the database from it (i.e., copy the archive into the database).
   (b) If there are later incremental dumps, modify the database according to each, earliest first.
2. Modify the database using the surviving log. Use the method of recovery appropriate to the log method being used.

Example 17.15: Suppose there is a media failure after the dump of Example 17.14 completes, and the log shown in Fig. 17.13 survives. Assume, to make the process interesting, that the surviving portion of the log does not include a <COMMIT T_1> record, although it does include the <COMMIT T_2> record shown in that figure. The database is first restored to the values in the archive, which is, for database elements A, B, C, and D, respectively, $(1, 2, 6, 4)$.

Now, we must look at the log. Since T_2 has completed, we redo the step that sets C to 6. In this example, C already had the value 6, but it might be that:

a) The archive for C was made before T_2 changed C, or

b) The archive actually captured a later value of C, which may or may not have been written by a transaction whose commit record survived. Later in the recovery, C will be restored to the value found in the archive *if* the transaction was committed.

Since T_1 does not have a `COMMIT` record, we must undo T_1. We use the log records for T_1 to determine that A must be restored to value 1 and B to 2. It happens that they had these values in the archive, but the actual archive value could have been different because the modified A and/or B had been included in the archive. □

17.5.4 Exercises for Section 17.5

Exercise 17.5.1: If a redo log, rather than an undo/redo log, were used in Examples 17.14 and 17.15:

a) What would the log look like?

! b) If we had to recover using the archive and this log, what would be the consequence of T_1 not having committed?

c) What would be the state of the database after recovery?

17.6 Summary of Chapter 17

✦ *Transaction Management*: The two principal tasks of the transaction manager are assuring recoverability of database actions through logging, and assuring correct, concurrent behavior of transactions through the scheduler (discussed in the next chapter).

✦ *Database Elements*: The database is divided into elements, which are typically disk blocks, but could be tuples or relations, for instance. Database elements are the units for both logging and scheduling.

✦ *Logging*: A record of every important action of a transaction — beginning, changing a database element, committing, or aborting — is stored on a log. The log must be backed up on disk at a time that is related to when the corresponding database changes migrate to disk, but that time depends on the particular logging method used.

✦ *Recovery*: When a system crash occurs, the log is used to repair the database, restoring it to a consistent state.

✦ *Logging Methods*: The three principal methods for logging are undo, redo, and undo/redo, named for the way(s) that they are allowed to fix the database during recovery.

✦ *Undo Logging*: This method logs the old value, each time a database element is changed. With undo logging, a new value of a database element can be written to disk only after the log record for the change has reached disk, but before the commit record for the transaction performing the change reaches disk. Recovery is done by restoring the old value for every uncommitted transaction.

✦ *Redo Logging*: Here, only the new value of database elements is logged. With this form of logging, values of a database element can be written to disk only after both the log record of its change and the commit record for its transaction have reached disk. Recovery involves rewriting the new value for every committed transaction.

✦ *Undo/Redo Logging* In this method, both old and new values are logged. Undo/redo logging is more flexible than the other methods, since it requires only that the log record of a change appear on the disk before the change itself does. There is no requirement about when the commit record appears. Recovery is effected by redoing committed transactions and undoing the uncommitted transactions.

✦ *Checkpointing*: Since all recovery methods require, in principle, looking at the entire log, the DBMS must occasionally checkpoint the log, to assure that no log records prior to the checkpoint will be needed during a recovery. Thus, old log records can eventually be thrown away and their disk space reused.

✦ *Nonquiescent Checkpointing*: To avoid shutting down the system while a checkpoint is made, techniques associated with each logging method allow the checkpoint to be made while the system is in operation and database changes are occurring. The only cost is that some log records prior to the nonquiescent checkpoint may need to be examined during recovery.

✦ *Archiving*: While logging protects against system failures involving only the loss of main memory, archiving is necessary to protect against failures where the contents of disk are lost. Archives are copies of the database stored in a safe place.

✦ *Incremental Backups*: Instead of copying the entire database to an archive periodically, a single complete backup can be followed by several incremental backups, where only the changed data is copied to the archive.

- *Nonquiescent Archiving*: We can create a backup of the data while the database is in operation. The necessary techniques involve making log records of the beginning and end of the archiving, as well as performing a checkpoint for the log during the archiving.

- *Recovery From Media Failures*: When a disk is lost, it may be restored by starting with a full backup of the database, modifying it according to any later incremental backups, and finally recovering to a consistent database state by using an archived copy of the log.

17.7 References for Chapter 17

The major textbook on all aspects of transaction processing, including logging and recovery, is by Gray and Reuter [5]. This book was partially fed by some informal notes on transaction processing by Jim Gray [3] that were widely circulated; the latter, along with [4] and [8] are the primary sources for much of the logging and recovery technology.

[2] is an earlier, more concise description of transaction-processing technology. [7] is a recent treatment of recovery.

Two early surveys, [1] and [6] both represent much of the fundamental work in recovery and organized the subject in the undo-redo-undo/redo tricotomy that we followed here.

1. P. A. Bernstein, N. Goodman, and V. Hadzilacos, "Recovery algorithms for database systems," *Proc. 1983 IFIP Congress*, North Holland, Amsterdam, pp. 799–807.

2. P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*, Addison-Wesley, Reading MA, 1987.

3. J. N. Gray, "Notes on database operating systems," in *Operating Systems: an Advanced Course*, pp. 393–481, Springer-Verlag, 1978.

4. J. N. Gray, P. R. McJones, and M. Blasgen, "The recovery manager of the System R database manager," *Computing Surveys* **13**:2 (1981), pp. 223–242.

5. J. N. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques*, Morgan-Kaufmann, San Francisco, 1993.

6. T. Haerder and A. Reuter, "Principles of transaction-oriented database recovery — a taxonomy," *Computing Surveys* **15**:4 (1983), pp. 287–317.

7. V. Kumar and M. Hsu, *Recovery Mechanisms in Database Systems*, Prentice-Hall, Englewood Cliffs NJ, 1998.

8. C. Mohan, D. J. Haderle, B. G. Lindsay, H. Pirahesh, and P. Schwarz, "ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging," *ACM Trans. on Database Systems* **17**:1 (1992), pp. 94–162.