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

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

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

**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 $A = 50$ and $B = 25$. 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

10 as we go. □

### 17.2.6 Exercises for Section 17.2

**Exercise 17.2.1:** 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 17.2.2:** The pattern introduced in Exercise 17.2.1 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 17.2.3:** The following is a sequence of undo-log records written by two transactions $T$ and $U$: <START $U$>; <$U, A, 10$>; <START $T$>; <$T, B, 20$>; <$U, C, 30$>; <$T, D, 40$>; <COMMIT $T$>; <$U, E, 50$>; <COMMIT $U$>. 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 $T$> (b) <COMMIT $T$> (c) <$U, E, 50$> (d) <COMMIT $U$>.

**Exercise 17.2.4:** For each of the situations described in Exercise 17.2.3, what values written by $T$ and $U$ *must* appear on disk? Which values *might* appear on disk?

! **Exercise 17.2.5:** Suppose that the transaction $U$ in Exercise 17.2.3 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 17.2.6:** Show the undo-log records for each of the transactions (call each $T$) of Exercise 17.1.1, assuming that initially $A = 50$ and $B = 25$.

**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.

<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 $A = 50$ and $B = 25$.

[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:** 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.

**Exercise 17.3.3:** Repeat Exercise 17.2.1 for redo logging.

**Exercise 17.3.4:** Repeat Exercise 17.2.3 for redo logging.

**Exercise 17.3.5:** Repeat Exercise 17.2.4 for redo logging.

## 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.

### 17.4.4 Exercises for Section 17.4

**Exercise 17.4.1:** 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 17.4.2:** The following is a sequence of undo/redo-log records written by two transactions $T$ and $U$: <START $U$>; <$U, A, 10, 11$>; <START $T$>; <$T, B, 20, 21$>; <$U, C, 30, 31$>; <$T, D, 40, 41$>; <COMMIT $T$>; <$U, E, 50, 51$>; <COMMIT $U$>. 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 $T$> (b) <COMMIT $T$> (c) <$U, E, 50, 51$> (d) <COMMIT $U$>.

**Exercise 17.4.3:** For each of the situations described in Exercise 17.4.2, what values written by $T$ and $U$ *must* appear on disk? Which values *might* appear on disk?

**Exercise 17.4.4:** 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.

**Exercise 17.4.5:** Show the undo/redo-log records for each of the transactions (call each $T$) of Exercise 17.1.1, assuming that initially $A = 50$ and $B = 25$.

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.