UNIT – 5 Crash Recovery and Backup

In this chapter, we discuss some of the techniques that can be used for database recovery in case of system failure.

Why Recovery Is Needed

Whenever a transaction is submitted to a DBMS for execution, the system is responsible for making sure that **either all the operations in the transaction are completed successfully** and their effect is recorded permanently in the database, **or that the transaction does not have any effect** on the database or any other
transactions.

In the **first case**, the transaction is said to be **committed**, whereas
in the **second case**, the transaction is **aborted**. The DBMS must not permit some operations of a transaction T to be applied to the database while other operations of T are not, because the whole transaction is a logical unit of database processing.

If a transaction fails after executing some of its operations but before executing all of them, the operations already executed must be undone and have no lasting effect.

Types of Failures –

Failures are generally classified as transaction, system, and
media failures. There are **several possible reasons for a transaction to fail** in the middle of execution:
1. **A computer failure (system crash).** A hardware, software, or network error occurs in the computer system during transaction execution. Hardware crashes are usually media failures—for example, main memory failure.

2**. A transaction or system error.** Some operation in the transaction may cause it to fail, **such as integer overflow or division by zero**. Transaction failure may also occur because of erroneous parameter values or because of a logical programming error. Additionally, the user may interrupt the transaction during its execution.

3. **Local errors or exception conditions detected by the transaction**. During transaction execution, certain conditions may occur that necessitate cancellation of the transaction.
For example, data for the transaction may not be found.
An **exception condition, such as insufficient account balance** in a banking database, may cause a transaction, such as a fund withdrawal, to be canceled.
This exception could be programmed in the transaction itself, and in such a case would not be considered as a transaction failure.

4. **Concurrency control enforcement**. The concurrency control method may abort a transaction **because it violates serializability**, **or it may abort one or more transactions to resolve a state of deadlock** among several transactions. Transactions aborted because of serializability violations or deadlocks are typically restarted automatically at a later time.

5. **Disk failure**. Some **disk blocks may lose their data because of a read or write malfunction or because of a disk read/write head crash**. This may happen during a read or a write operation of the transaction.

6. **Physical problems and catastrophes**. This refers to an endless list of problems that includes **power or air-conditioning failure, fire, theft, sabotage, overwriting disks or tapes by mistake**, and mounting of a wrong tape by the operator.

Database Recovery Concept

Recovery Outline and Categorization of Recovery Algorithms

Recovery from transaction failures usually **means that the database is restored to the most recent consistent state** before the time of failure.

To do this, the system **must keep information about the changes** that were applied to data items by the various transactions. This information is typically kept in the **system log**.

A typical strategy for recovery may be summarized informally as follows:
1. If there is extensive damage to a wide portion of the **database due to catastrophic failure, such as a disk crash**, the recovery method restores a past

copy of the database that was backed up to archival storage (typically tape or other large capacity offline storage media) and reconstructs a more current state **by reapplying or redoing the operations** of committed transactions from the backed-up log, up to the time of failure.

2. When the database on disk is not physically damaged, and a noncatastrophic failure has occurred, the recovery strategy is **to identify any changes that may cause an inconsistency** in the database.

For example, a transaction that has updated some database items on disk but has not been committed needs to have its **changes reversed by undoing** its write operations.
It may also be necessary to **redo some operations in order to restore a consistent state of the database**; for example, if a transaction has committed but some of its write operations have not yet been written to disk.

For non catastrophic failure, the **recovery protocol does not need a complete archival copy of the database**. Rather, the entries kept in the online system log on disk are analyzed to determine the appropriate actions for recovery.

Conceptually, we can distinguish **two main policies for recovery from non catastrophic transaction failures**: **deferred update and immediate update**.

Deferred update

The deferred update techniques **do not physically update the database on disk until after a transaction commits**; then the updates are recorded in the database.

Before reaching commit, **all transaction updates are recorded in the local transaction workspace or** in the **main memory buffers** that the DBMS maintains.

Before commit, the **updates are recorded persistently in the log file on disk**, and then **after commit**, the **updates are written to the database from the main memory buffers.**

If a transaction fails before reaching its commit point, it will not have changed the database on disk in any way, so UNDO is not needed. It may be **necessary to REDO the effect of the operations** of a committed transaction from the log, because their effect may not yet have been recorded in the database on disk. Hence, **deferred update is also known as the NO-UNDO/REDO algorithm.**

Immediate update

In the **immediate update techniques**, the database may be **updated by some operations of a transaction before the transaction reaches its commit point**.

However, these **operations must also be recorded in the log on disk by force-writing before they are applied to the database on disk**, making recovery still possible.

If a transaction fails after recording some changes in the database on disk but before reaching its commit point, the **effect of its operations on the database must be undone**; that is, the **transaction must be rolled back**.

In the general case of immediate update, **both undo and redo may be required during recovery**. This **technique, known as the UNDO/REDO algorithm**, requires both operations during recovery and is used most often in practice.

A variation of the algorithm where all updates are required to be recorded in the database on disk before a transaction commits requires undo only, so it is known as the UNDO/NO-REDO algorithm.

Database Recovery Techniques

1. Checkpoints in the System Log and Fuzzy Checkpointing

Another type of entry in the log is called a **checkpoint** (**checkpoint, list of active transactions).**

A record is written into the log periodically at that point when the system **writes out to the database** on disk all DBMS buffers that have been modified.

As part of checkpointing, **the list of transaction ids for active transactions at the time of the checkpoint is included in the checkpoint record**, so that these transactions can be easily identified during recovery.

The **recovery manager of a DBMS must decide at what intervals to take a checkpoint**. The **interval may be measured in time**—say, every **m minutes**—

or in the **number t of committed transactions** since the last checkpoint, where the values of m or t are system parameters.

Taking a **checkpoint consists of the following actions**:

1. Suspend execution of transactions temporarily.

2. Force-write all main memory buffers that have been modified to disk.

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

4. Resume executing transactions.

(**Force write** (or the **Force policy**) is a data recovery and buffer management strategy that requires all modified data pages by a transaction to be written from the main memory buffer to persistent disk storage before the transaction is allowed to formally commit.)

The Problem: Checkpointing Pauses the System

In a database system, a **checkpoint** is **a point in the log** where the system knows it has written enough data to disk so that recovery later will be faster.

But creating a **checkpoint usually has multiple steps**—especially *collecting information about active transactions and dirty pages*.
If the **system waited for step 2 to finish before moving on, it would pause transaction processing**, which is **bad** for performance.

The Solution: Fuzzy Checkpointing

Fuzzy checkpointing allows the system **to *keep processing new transactions*** even while it is building the checkpoint.

Here's how it works:

1. Start the checkpoint early

The system writes a special record to the log:

[begin_checkpoint]

After this is written, the **system is allowed to continue processing transactions normally.**

Step 2 (collecting the list of active transactions, dirty pages, etc.) happens in the background.

2. Keep the previous checkpoint active during this time

Because the new checkpoint isn't finished yet, the **system** still needs to **rely on the *old* checkpoint for recovery**.

So the system keeps a **file on disk** that **stores a pointer to the most recent *valid* checkpoint** in the **log**.

3. Finish the checkpoint

Once all the necessary information has been collected, the system writes **a new log record:**

[end_checkpoint, ...]

This record contains the transaction table and dirty-page (It is a block of memory that contains data which has been modified but not yet written back to disk) table (or equivalent information).

4. Update the checkpoint pointer

Now that the checkpoint is complete and valid, the system **updates the pointer in the checkpoint file** to the location of this **new checkpoint**.

This makes it the new starting point for recovery.

2. NO-UNDO/REDO Recovery Based on Deferred Update

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

During transaction execution, the updates are r**ecorded only in the log and in the cache buffers**. After the transaction **reaches its commit point** and the log is force written to disk, the updates are recorded in the database.

If a transaction fails before reaching its commit point, there is no need to undo any operations because the transaction has not affected the database on disk in any way.

Therefore, **only REDO** type log **entries are needed** in the log.

The **UNDO**-type log **entries are not needed** since no undoing of operations will be required during recovery.

Although **this may simplify the recovery process**, it **cannot be used in practice unless transactions are short and each transaction changes few items**.

For other types of transactions, there is the **potential for running out of buffer space** because transaction changes must be held and cannot be replaced.

We can state a typical deferred update protocol as follows:

Deferred update (also called NO-UNDO/REDO protocol) means:

A transaction does not immediately update the database on disk.
Instead, it keeps all its changes in memory until it successfully commits.

i. No database changes on disk until commit (No-Steal Policy)

- A transaction can modify data **only in the buffer (memory)**.
- These modified buffer pages must be **pinned**, meaning:
  - They cannot be written to disk by the buffer manager.
  - They **must remain in memory until the transaction commits**.
- The disk version of the data remains unchanged until commit.

Why?
Because **if a crash occurs *before commit***, there should be **no partial updates** on disk.
So the system can simply discard the in-memory changes—no need to undo anything.

This is called the **NO-UNDO guarantee**.

ii. Transaction can commit only after all REDO log entries are force-written (WAL rule)

Before the system considers the transaction as "committed":

- All **REDO log entries** (records describing all updates) must be **written to the log**.

- The log buffer must be **force-written to disk**.

This is required by **Write-Ahead Logging (WAL)**:

“Log must reach disk before the actual updated data reaches disk.”

Only after log entries are safely on disk can we say the transaction is committed.

Why no UNDO is needed?

- The database on disk is **not changed** until commit.
- If a transaction fails before commit:
  - Its changes exist only in memory → system simply discards them.
- Since nothing reached disk, there is **nothing to undo**.

So, deferred update is also known as a **NO-UNDO** recovery method.

Why REDO is needed?

Suppose the transaction committed, but:

- The system **crashes before writing** the updated data pages from memory to disk.

Now the log contains the updates but the **disk does not**.

During recovery:

- The system scans the log.
- For **all committed transactions**, it **reapplies (REDO)** their updates to the database.
- This ensures that the disk reflects the state of committed transactions.

This is why the protocol is also a **REDO-only protocol**.

Figure 22.2 illustrates a timeline for a possible schedule of executing transactions.

When the checkpoint was taken **at time t1**, **transaction T1 had committed, whereas transactions T3 and T4 had not.**

Before the system **crash** at **time t2**, T3 and T2 were committed but not T4 and T5.

According to the RDU_M method, there is **no need to redo** the write_item **operations of transaction T1**—or any transactions committed before the last checkpoint time t1.

The write_item operations of **T2 and T3 must be redone**, however, because both transactions **reached their commit points after the last checkpoint**. Recall that the log is force-written before committing a transaction.

Transactions **T4** and **T5** are **ignored**: They are effectively **canceled or rolled back** because none of their write_item operations were recorded in the database on disk under the deferred update protocol (no-steal policy)

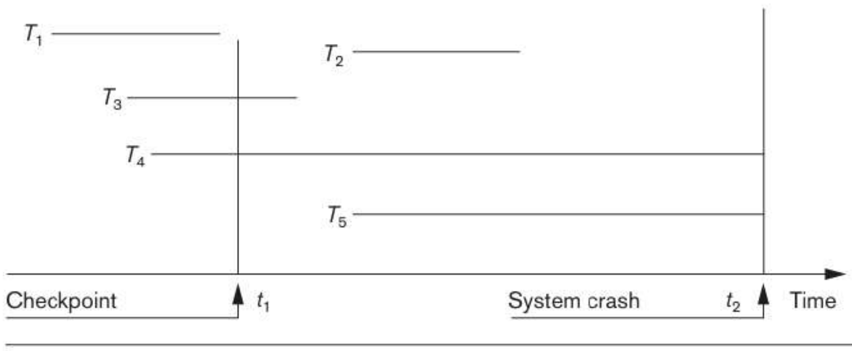


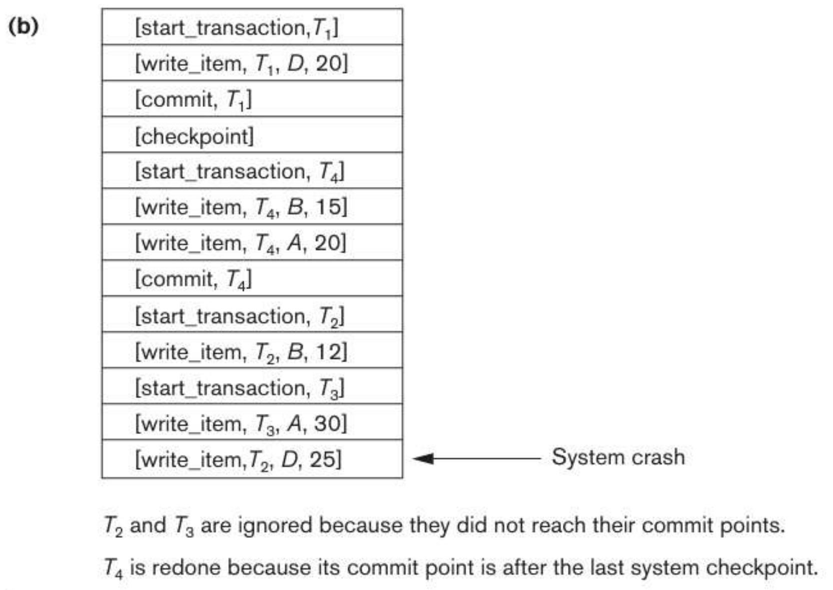
Figure 22.2
An example of a recovery timeline to illustrate the effect of checkpointing.

Recovery Techniques Based on Immediate Update

In these techniques, when a transaction issues an update command, the **database on disk can be updated immediately**, **without any need to wait** for the **transaction** to reach its **commit point**.

(a)

T_1	T_2	T_3	T_4
read_item(A)	read_item(B)	read_item(A)	read_item(B)
read_item(D)	write_item(B)	write_item(A)	write_item(B)
write_item(D)	read_item(D)	read_item(C)	read_item(A)
	write_item(D)	write_item(C)	write_item(A)



(b)

[start_transaction, T_1]
[write_item, T_1, D, 20]
[commit, T_1]
[checkpoint]
[start_transaction, T_4]
[write_item, T_4, B, 15]
[write_item, T_4, A, 20]
[commit, T_4]
[start_transaction, T_2]
[write_item, T_2, B, 12]
[start_transaction, T_3]
[write_item, T_3, A, 30]
[write_item, T_2, D, 25] ← System crash

T_2 and T_3 are ignored because they did not reach their commit points.

T_4 is redone because its commit point is after the last system checkpoint.

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

Provisions must be made for **undoing the effect of update operations that have been applied** to the database **by** a **failed transaction**. This is accomplished by **rolling back the transaction** and **undoing the effect** of the transaction's write_item operations.

Therefore, the UNDO-type log entries, which include the old value of the item, must be stored in the log. Because UNDO can be needed during recovery, these methods follow a steal strategy for deciding when updated main memory buffers can be written back to disk.

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

1. If the **recovery technique** ensures that **all updates of a transaction** are **recorded** in the **database** on disk **before** the **transaction commits**, there is never **a need to REDO any operations of committed transactions**. This is called the UNDO/NO-REDO recovery algorithm.

In this method, **all updates by a transaction must be recorded on disk before the transaction commits**, so that REDO is never needed. Hence, this method must utilize the **steal/forceto disk**.

2. If the transaction is allowed to commit before all its changes are written to the database, we have the most general case, known as the UNDO/REDO recovery algorithm.
In this case, the **steal/no-force strategy** is applied.

Shadow Paging

This recovery scheme **does not require the use of a log in a single-user environment**. In a **multiuser environment**, a **log may be needed** for the concurrency control method.

Shadow paging **considers the database** to be made up of a number of **fixedsize disk pages** (or disk blocks)—say, n—for recovery purposes.

A **directory with n entries** is constructed, where the **ith entry points to the ith database page** on disk.

The **directory is kept in main memory** if it is not too large, and all references—reads or writes—to database pages on disk go through it.

When a transaction begins executing, **the current directory**—whose entries point to the **most recent or current** database **pages** on disk—is **copied into a shadow directory.**

The **shadow directory is then saved on disk** while the **current directory is used by the transaction.**

How updates work

- When a page is updated, the system **does not overwrite** the original page.
- Instead, it:
  1. Creates a **new copy** of the page in a free disk block.
  2. Updates the **current directory** to point to this new page.
  3. The **shadow directory still points to the old page**.

Thus, both old and new versions exist at the same time.

Recovery during a crash

If a **crash happens before commit**:

- The system **discards the current directory** and all **new pages**.
- It **restores the shadow directory**, which still points to the unmodified pages.
- Result: the **database returns** to the exact **state before** the **transaction started**.
- This requires **NO UNDO and NO REDO**, since old pages were never overwritten.

Commit

When a transaction commits:

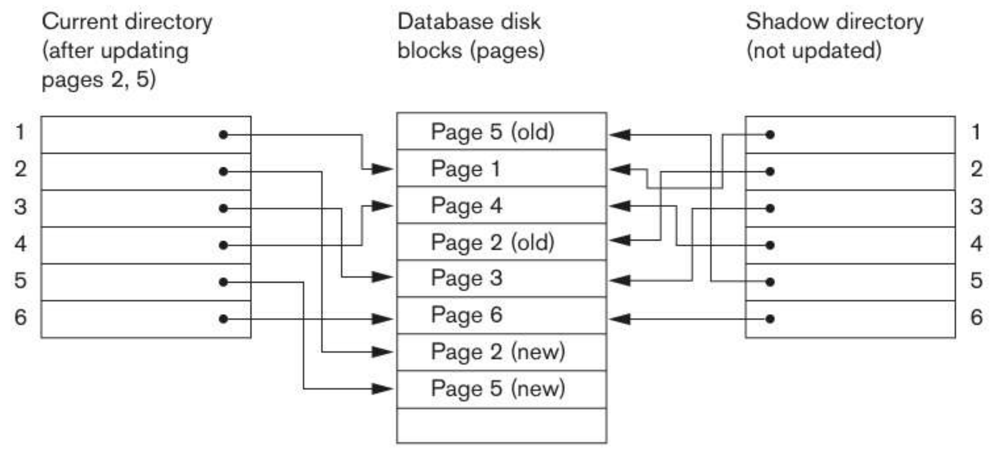
- The system **discards the old shadow directory**.
- The current directory becomes the new shadow directory.
- Old, outdated pages must be **garbage-collected** and **returned to the free-page list**.

Problems and disadvantages

1. **Pages keep moving around**
   Updated pages get written to new disk blocks, so physical locality of related pages can be lost → **slows performance** unless complex storage management is used.
2. **Shadow directory can be large**
   Copying the directory to disk at each commit is expensive.
3. **Garbage collection needed**
   Pages replaced by new versions must be identified and freed after commit.
4. **Atomic switch requirement**
   Switching from old directory to new directory must be done **atomically**, so the system is **never** left with a **half-updated** directory.
5. **Multiuser environment is harder**
   Concurrent transactions require:
   - logs
   - checkpoints
   - careful handling of page versions
     This removes much of shadow paging's simplicity.

Figure 22.4 illustrates the concepts of shadow and current directories. For pages updated by the transaction, two versions are kept. The old version is referenced by the shadow directory and the new version by the current directory.

Figure 22.4
An example of shadow paging.



Current directory (after updating pages 2, 5)
Database disk blocks (pages)
Shadow directory (not updated)
1
2
3
4
5
6
Page 5 (old)
Page 1
Page 4
Page 2 (old)
Page 3
Page 6
Page 2 (new)
Page 5 (new)
1
2
3
4
5
6

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