Chapter 9

Crash Recovery

- ✓ A computer system, like any other device, is subject to failure from a variety of causes: disk crash, power outage, software error, fire etc. In any failure, information may be lost.
- ✓ Therefore, the database system must take actions in advance to ensure that the atomicity and durability properties of transactions are preserved.
- ✓ An integral part of a database system is a **recovery scheme** that can restore the database to the consistent state that existed before the failure.

Failure classification

There are various types of failure that may occur in a system, each of which needs to be dealt with in a different manner. Some major types of failure are as follows:

- ❖ **Transaction failure**
  There are two types of errors that may cause a transaction to fail:
  - ➢ **Logical error**. The transaction can no longer continue with its normal execution because of some internal condition, such as bad input, data not found, overflow, or resource limit exceeded.
  - ➢ **System error**. The system has entered an undesirable state (for example, deadlock), as a result of which a transaction cannot continue with its normal execution. The transaction, however, can be re executed at a later time.
- ❖ **System crash**
  - ➢ A power failure or other hardware or software failure causes the system to crash.
  - ➢ **Fail-stop assumption**: non-volatile storage contents are assumed to not be corrupted by system crash.
  - ➢ Database systems have numerous integrity checks to prevent corruption of disk data.
- ❖ **Disk failure**
  - ➢ A disk block loses its content as a result of either a head crash or failure during a data-transfer operation.
  - ➢ Copies of the data on other disks, or tertiary media, such as DVD or tapes, are used to recover from the failure.

Storage structure

Various data items in the database may be stored and accessed in a number of different storage media. We identified three categories of storage.

- Volatile storage
- Nonvolatile storage
- Stable storage

Volatile storage

- ✓ Data residing in volatile storage does not survive system crashes
- ✓ Examples: main memory, cache memory

Nonvolatile storage

- ✓ Data residing in non-volatile storage survives system crashes
- ✓ Examples: disk, tape, flash memory, non-volatile RAM
- ✓ But may still fail ,losing data

Stable storage

- ✓ A mythical form of storage that survives all failures
- ✓ Approximated by maintaining multiple copies on distinct nonvolatile media

Stable storage implementation

- ✓ To implement stable storage, we maintain multiple copies of each block on separate disks.
- ✓ Copies can be at remote site to protect against disaster such as fire or flooding.
- ✓ RAID systems guarantee that the failure of a single disk will not result in loss of data.
- ✓ Update of information in a controlled manner to ensure failure during data transfer doesn't damage the needed information.

Failure during data transfer can still result in inconsistent copies: transfer between main memory and storage can result in:

Successful completion: The transferred information arrived safely at its destination.

Partial failure: A failure occur in midst of a transfer and the destination block has incorrect information.

Total failure: The failure occur sufficiently early during the transfer that the destination block remains intact.

If a data-transfer failure occurs, the system detects it and invokes a recovery procedure to restore the block to a consistent state. For this, the system must maintain two physical blocks for each logical database block. An output operation is executed as follows:

1. Write the information onto the first physical block.

2. When the first write completes successfully, write the information onto the second physical block.

3. The output is completed only after the second write completes successfully.

If the system fails while blocks are being written, it is possible that the two copies of a block are consistent to each other.

During recovery, for each block, the system would need to examine two copies of the blocks.

- ☞ If both are the same and no detectable error exists, then no further actions are necessary.
- ☞ If the system detects an error in one block, then it replaces its content with the content of another block.
- ☞ If both blocks contains the detectable error, but they differ in content, then the system replaces the content of the first block with the value of the second.

This recovery procedure ensures that a write to stable storage either succeeds completely (that is, update all copies) or results in no change.

Log based recovery

- ✓ The **log** is kept on stable storage
- ✓ A **log** is a sequence of **log records** and keep information about update activities on the database.
- ✓ When transaction T_i starts, it registers itself by writing a $<T_i$ **start**> log record
- ✓ *Before* T_i executes **write**(X), a log record $<T_i, X, V_1, V_2>$ is written, where V_1 is the value of X before the write (the **old value**)**,** and V_2 is the value to be written to X (the **new value**).
- ✓ When T_i finishes it last statement, the log record $<T_i$ **commi**t> is written.
- ✓ We assume for now that log records are directly to stable storage (that is ,they are not buffered)

Two approaches using logs

1. Immediate database modification
2. Deferred database modification

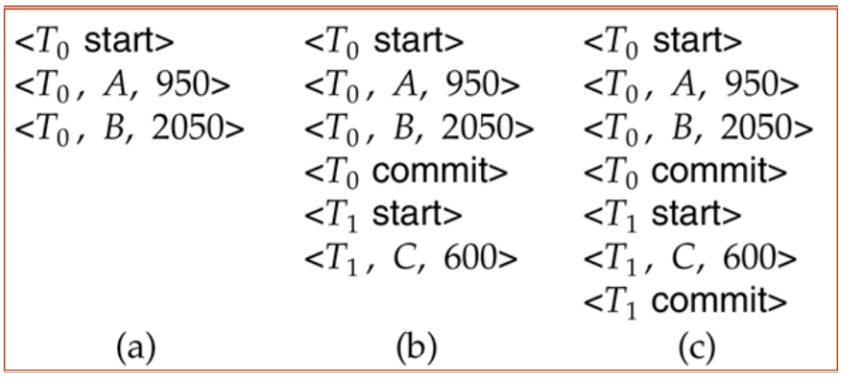
1. Deferred database modification

- ✓ The deferred database modification scheme records all modifications to the log, but defers all the writes to after partial commit.
- ✓ If the system crashes before the transaction completes its execution, or if the transaction abort, then the information on the log is simply ignored.
- ✓ Transactions starts by writing $<T_i$ start> record to the log.
- ✓ A write(X) operations results in a log record $< T_i,X,V>$ being written, where V is the new value for X.
- ✓ The write is not performed on X at this time, but is deferred.
- ✓ When T_i Partially commits, $< T_i,$ commit> is written to the log.
- ✓ Finally ,the log records are read and used to actually execute the previously deferred writes.

During recovery after crash, a transaction needs to be redone if and only if both $<T_i$ start> and

$<T_i$ commit> are there is the log.

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



If log on stable storage at time of crash is as in case:

a) No redo actions need to be taken

b) redo (T_0) must be performed since < T_0 **commit**> is present

c) redo (T_0) must be performed followed by redo (T_1) since < T_0 **commit**> < T_1 **commit**> are present.

If the transaction fails before reaching its commit point, it will have made no changes to database in any way so no UNDO operation is necessary. It may be necessary to REDO effect of the operations of commit transactions from the log because their effect may not have been recorded in the database. Therefore it is also known as **no undo** algorithm**.**

2. Immediate database modification

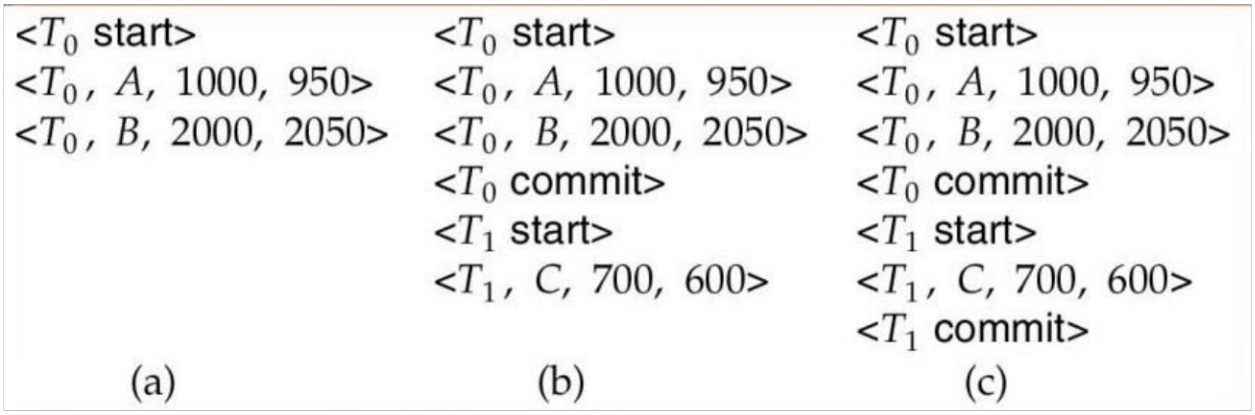
- ✓ Allows database modification while the transaction is still active.
- ✓ Which means all the modifications that is performed before the transaction reaches to commit state are updated to database.
- ✓ Database modifications written by active transactions are called uncommitted modifications.
- ✓ Update log must be written before database items is written.
- ✓ Transaction starts by writing <T_i start> record to log.
- ✓ A write(X) operations result in the log record <T_i X,V1,V2> where V1 is the old value and V2 is the new value . Since undoing may be needed, update logs must have both old value and new value.
- ✓ The write operation on X is recorded in log on disk and is output directly to stable storage without concerning transaction commits or not.
- ✓ In case of failure recovery procedure has two operations instead of one:
    1. undo (T_i) restores the value of all data items updated by T_i to their old values, going backwards from the last record for T_i.
    2. redo(T_i) sets the value of all data items updated by T_i to the new values, going forward from the first log record for T_i.

When recovering after failure:

- ✓ Transaction T_i needs to be undone if the log contains the record **<T_i start>** ,but does not contain the record **<T_i commit>**
- ✓ Transaction T_i needs to be redone if the log contains both the record **<T_i start>** and the record **<T_i commit>**
- ✓ Undo operations are performed first, then redo operations.

Example:

Below we show the log as it appears at three instances of time



Recovery actions in each case above are:

a) Undo (T_0):B is restored to 2000 and A to 1000

b) Undo (T_1) and redo (T_0) : C is restored to 700,and A and B are set to 950 and 2050 respectively.

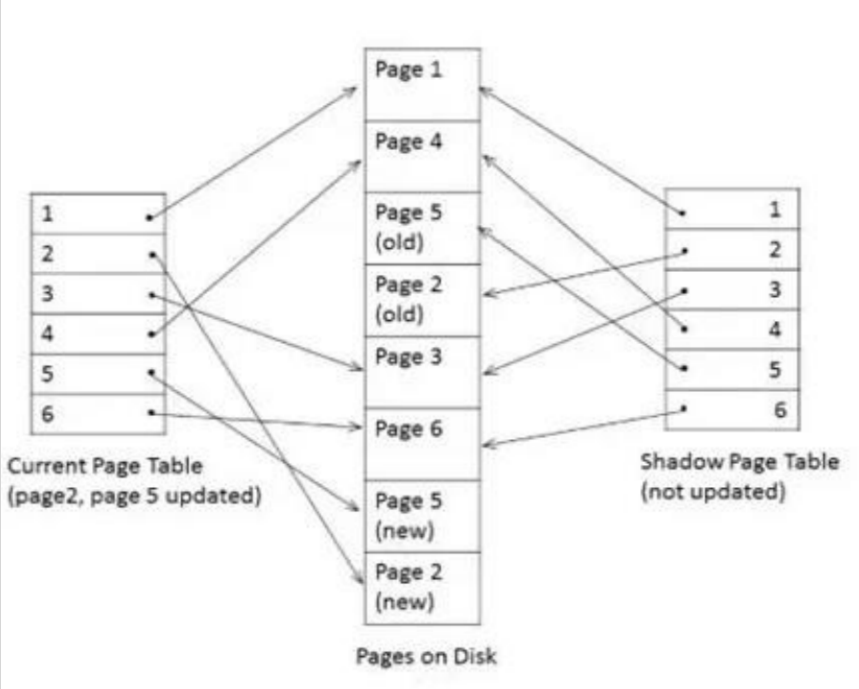
c) Redo (T_0) and Redo (T_1): A and B is set to 950 and 2050 respectively. Then C is set to 600.

Shadow Paging

- ✓ **Shadow paging** is an alternative to log-based recovery and this scheme is useful if transactions execute serially.
- ✓ Basic Idea is to maintain *two* page tables during the lifetime of a transaction –the **current page table**, and the **shadow page table**
- ✓ Store the shadow page table in nonvolatile storage, such that state of the database prior to transaction execution may be recovered.
- ✓ When the transaction starts both the page tables are identical.
- ✓ Shadow page table is never modified during execution.
- ✓ Only current page table is used for data item accesses during execution of the transaction.

Whenever any page is about to be written for the first time

- ✓ A copy of this page is made onto an unused page.
- ✓ The current page table is then made to point to the copy
- ✓ The update is performed on the copy



The transaction can commits after performing the following operations

1. Flush all modified pages in main memory to disk
2. Output current page table to disk
3. Make the current page table the new shadow page table, as follows:
    a) keep a pointer to the shadow page table at a fixed (known) location on disk.
    b) to make the current page table the new shadow page table, simply update the pointer to point to current page table on disk

✓ Once pointer to shadow page table has been written, transaction is committed.

To recover from a failure during transaction execution, it is sufficient to free the modified database pages and to discard the current page table. The state of the database before transaction execution is available through the shadow page table, and that state is recovered by reinstating the shadow page table.

The database thus is returned to its state prior to the transaction that was executing when the crash occurred, and any modified pages are discarded.

Committing a transaction corresponds to discarding the previous shadow page table.

Since recovery involves neither undoing nor redoing data items, this technique can be categorized as a NO-UNDO/NO-REDO technique for recovery

Advantages of shadow-paging over log-based schemes

- no overhead of writing log records
- recovery is simple

Disadvantages:

- ✓ Copying the entire page table is very expensive
- ✓ **Commit overhead:** The commit of a single transaction requires multiple blocks to be output: the current page table, the actual data i.e. updated pages. Log based scheme need to output only the log records.
- ✓ **Data fragmentation:** shadow paging cause database pages to change location when they are updated.
- ✓ **Garbage collection:** Each time a transaction commits, the database pages containing the old version of data changed by the transaction must become inaccessible. Such pages do not contain usable information hence considered as garbage. Periodically it is necessary to find all the garbage pages and add them to the list of free pages. The process is called garbage collection and requires additional overhead and complexity on the system.
- ✓ Hard to extend algorithm to allow transaction to run concurrently.

Checkpoint

The Checkpoint is used to declare a point before which the DBMS was in a consistent state, and all transactions were committed. During transaction execution, such checkpoints are traced. After execution, transaction log files will be created. Upon reaching the savepoint/checkpoint, the log file is destroyed by saving its update to the database. Then a new log is created with upcoming execution operations of the transaction and it will be updated until the next checkpoint and the process continues.

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

1. The search process is time-consuming.

2. Most of the transactions that, according to our algorithm, need to be redone have already written their updates into the database.

A scheme called check point is used to limit the volume of log information that has to be handled and processed in the events of system failure. A check point performs periodic copy of log information onto the stable storage. A check point requires the following sequence of actions to take place:

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

Transactions are not allowed to perform any update actions, such as writing to a buffer block or writing a log record, while a checkpoint is in progress.

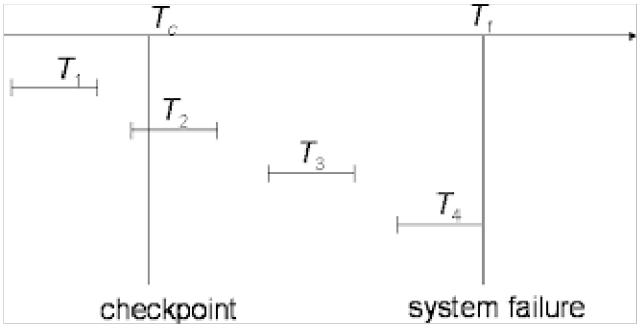
During recovery we need to consider only the most recent transaction T_i that started before the checkpoint, and transactions that started after T_i.

- ✓ Scan backwards from end of log to find the most recent <**checkpoint** *L*> record
- ✓ continue scanning backwards till a record **<T_i start>** is found.
- ✓ Need to consider the part of log following above **<T_i start>** record. Earlier part of log can be ignored during recovery .

After the transaction T_i identified, the redo and undo operations to be applied to the T_i and all T_j that started execution after transaction T_i .

For all transactions (starting from T_i or later)

- ➢ with no **<T_i commit>,** execute **undo** (T_i) *(Done only in case of immediate database modification)*
- ➢ with **<T_i commit>,** execute **redo** (T_i)

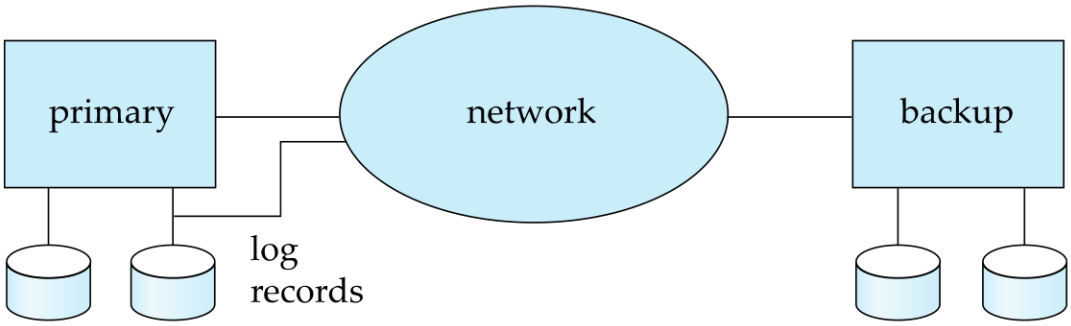


- ✓ T_1 can be ignored (updates already output to disk due to checkpoint)
- ✓ T_2 and T_3 redone
- ✓ T_4 undone

Remote backup system

- ✓ Traditional transaction-processing systems are centralized or client–server systems. Such systems are vulnerable to environmental disasters such as fire, flooding, or earthquakes.
- ✓ So that there is a need for transaction-processing systems that can function in spite of system failures or environmental disasters and such systems must provide high availability.
- ✓ We can achieve high availability by performing transaction processing at one site, called the primary site, and having a remote backup site where all the data from the primary site are replicated. The remote backup site is sometimes also called the secondary site.
- ✓ The remote site must be kept synchronized with the primary site, as updates are performed at the primary.
- ✓ We achieve synchronization by sending all log records from primary site to the remote backup site. The remote backup site must be physically separated from the primary—for example, we can locate it in a different state—so that a disaster at the primary does not damage he remote backup site.

Figure below shows the architecture of a remote backup system.



- ✓ When the primary site fails, the remote backup site takes over processing. It performs recovery, using its copy of the data from the primary, and the log records received from the primary. In effect, the remote backup site is performing recovery actions that would have been performed at the primary site when the latter recovered.
- ✓ Once recovery has been performed, the remote backup site starts processing transactions.

Several issues must be addressed in designing a remote backup system:

Detection of failure: It is important for the remote backup system to detect when the primary has failed. Failure of communication lines can fool the remote backup into believing that the primary has failed. To avoid this problem, we maintain several communication links with independent modes of failure between the primary and the remote backup.

Transfer of control: When the primary fails, the backup site takes over processing and becomes the new primary. When the original primary site recovers, it can either play the role of remote backup, or take over the role of primary site again. In either case, the old primary must receive a log of updates carried out by the backup site while the old primary was down.

Time to recover: If the log at the remote backup grows large, recovery will take a long time. The remote backup site can periodically process the redo log records that it has received and can perform a checkpoint, so that earlier parts of the log can be deleted.

A hot-spare configuration can make takeover by the backup site almost instantaneous. In this configuration, the remote backup site continually processes redo log records as they arrive, applying the updates locally. As soon as the failure of the primary is detected, the backup site completes recovery by rolling back incomplete transactions; it is then ready to process new transactions.

Time to commit: To ensure that the updates of a committed transaction are durable, a transaction must not be declared committed until its log records have reached the backup site. This delay can result in a longer wait to commit a transaction, and some systems therefore permit lower degrees of durability.

The degrees of durability can be classified as follows:

One-safe: A transaction commits as soon as its commit log record is written to stable storage at the primary site.

Two-very-safe: A transaction commits as soon as its commit log record is written to stable storage at the primary and the backup site.

Two-safe: This scheme is the same as two-very-safe if both primary and backup sites are active. If only the primary is active, the transaction is allowed to commit as soon as its commit log record is written to stable storage at the primary site.

Data Backup/Recovery

Backup refers to storing a copy of original data which can be used in case of data loss. Backup is considered one of the approaches to data protection. Important data of the organization needs to be kept in backup efficiently for protecting valuable data. Backup can be achieved by storing a copy of the original data separately or in a database on storage devices. There are various types of backups are available like full backup, incremental backup, Local backup, mirror backup, etc.

Recovery refers to restoring lost data by following some processes. Even if the data was backed up still lost so it can be recovered by using/implementing some recovery techniques. When a database fails due to any reason then there is a chance of data loss, so in that case recovery process helps in improve the reliability of the database.

Difference between backup and recovery

Backup	Recovery
Backup refers to storing a copy of original data separately.	Recovery refers to restoring the lost data in case of failure.
So we can say Backup is a copy of data which is used to restore original data after a data loss/damage occurs.	So we can say Recovery is a process of retrieving lost, corrupted or damaged data to its original state.
It helps in improving data protection.	It helps in improving the reliability of the database.
Backup makes the recovery process more easier.	Recovery has no role in data backup.
Backup stores the copy of the file in an external location.	A restoration is performed internally on computer system.
Backup requires extra storage space.	Recovery does not require additional external storage space because restoring is done internally.
It is not generated automatically.	It is the automatic creation of restore points by computer system.
The cost of backup is affordable.	The cost of recovery is expensive.