**Chapter 4: Database Recovery Techniques**

## Database Recovery Concepts

## Types of failures

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

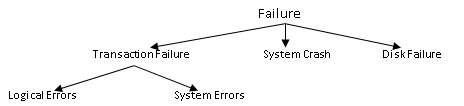
There are many different types of failure that can affect database processing, each of which has to be dealt with in a different manner. Some failures affect main memory only, while others involve nonvolatile (secondary) storage.

* **Transactions may fail because** of:
  + - **Logical errors**: transaction cannot complete due to some internal  
      error condition
    - **System errors**: the database system must terminate an active  
      transaction due to an error condition (e.g., incorrect input, **deadlock**)

Among the causes of failure are:

* **System crashes failures:** due to hardware or software errors, resulting in loss of main memory;
* **Media failures**, such as head crashes or unreadable media, resulting in the loss of parts of secondary storage;
* **Application software errors**, such as logical errors in the program that is accessing the database, that cause one or more transactions to fail;
* **Natural physical disasters**, such as fires, floods, earthquakes, or power failures;
* **Carelessness** or unintentional destruction of data or facilities by operators or users;
* **Sabotage**, or intentional corruption or destruction of data, hardware, or software facilities.

In databases, usually a failure can generally be categorized as one of the following major groups:

* **Transaction failure**:
* **System failure:**
* **Media failure**:
* **Transaction failure**. a transaction cannot continue with its execution; therefore, it is aborted and if desired it may be restarted at some other time. Reasons: Deadlock, timeout, protection violation, or system error. 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**. There is a hardware malfunction, or a bug in the database software or the operating system, that causes the loss of the content of volatile storage, and brings transaction processing to a halt. The content of nonvolatile storage remains intact, and is not corrupted. The assumption that hardware errors and bugs in the software bring the system to a halt, but do not corrupt the nonvolatile storage contents, is known as the **fail-stop assumption**. Well-designed systems have numerous internal checks, at the hardware and the software level, that brings the system to a halt when there is an error. Hence, the fail-stop assumption is a reasonable one. The database system is unable to process any transactions. Some of the common reasons of system failure are: register overflow, addressing error, power failure, memory failure, etc.
* **Disk failure**. A disk block loses its content as a result of either a head crash or failure during a data-transfer operation. Copies of the data on other disks, or archival backups on tertiary media, such as DVD or tapes, are used to recover from the failure. Failure of non-volatile storage media (mainly disk). Some of the common reasons are: head crash, dust on the recording surfaces, fire, etc.

Whatever the cause of the failure, there are two principal effects that we need to consider: the loss of main memory, including the database buffers, and the loss of the disk copy of the database. All of these failures can and do occur in database applications. Since a database is shared by many people and since it is a key element of the organization’s operations, it is important to recover it as soon as possible. To make the database secured, one should formulate a “plan of attack” in advance. The plan will be used in case of database insecurity that may range from minor inconsistency to total loss of the data due to hazardous events.

The basic steps in performing a recovery are

1. Isolating the database from other users. Occasionally, you may need to drop and re-create the database to continue the recovery.
2. Restoring the database from the most recent useable dump.
3. Applying transaction log dumps, in the correct sequence, to the database to make the data as current as possible.

It is a good idea to test your backup and recovery plans periodically by loading the backups and transaction logs into a test database and verifying that your procedure really works.

One can recover databases after three basic types of problems:

* **User error,**
* **Software failure, and**
* **Hardware failure.**

Each type of failure requires a recovery mechanism. In a transaction recovery, the effect of failed transaction is removed from the database, if any. In a system failure, the effects of failed transactions have to be removed from the database and the effects of *completed* transactions have to be *installed* in the database. The database recovery manger is responsible to guarantee the atomicity and durability properties of the ACID property.

**Example:**

|  |  |
| --- | --- |
| T1: Read (A,t); t ← t×2  Write (A,t);  Read (B,t); t ← t×2  Write (B,t);  failure!  Output (A);  Output (B); |  |
|  |  |

As you can see above on the picture the value of A and B is 16 on the memory. Whereas, the value of A is 16 and before the value of B which should be 16 is 8. The failure occurs before the value of B=16 is written to disk. There should be a recovery mechanism to return either both values to 16 or both values to 8.

Transaction Log

Execution history of concurrent transactions.

* DBMS starts at time t0, but fails at time tf. Assume data for transactions T2 and T3 have been written to secondary storage.
* T1 and T6 have to be undone. In absence of any other information, recovery manager has to redo T2, T3, T4, and T5.
* tc is the checkpoint time by the DBMS

A logging and recovery manager, responsible for the durability of transactions.

**Log Records**

The most widely used structure for recording database modifications is the **log**.

The log is a sequence of **log records**, recording all the update activities in the database.

There are several types of log records.

An **update log record** describes a single database write. It has these fields:

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

Transaction creates a log record prior to modifying the database. The log records allow the system to undo changes made by a transaction in the event that the transaction must be aborted; they allow the system also to redo changes made by a transaction if the transaction has committed but the system crashed before those changes could be stored in the database on disk.

We represent an update log record as*<Ti , Xj , V*1*, V*2*>*, indicating that transaction *Ti* has performed a write on data item *Xj* . *Xj* had value *V*1 before the write, and has value *V*2 after the write. Other special log records exist to record significant events during transaction processing, such as the start of a transaction and the commit or abort of a transaction. Among the types of log records are:

* + *<Ti* start*>*. Transaction *Ti* has started.
  + *<Ti* commit*>*. Transaction *Ti* has committed.
  + *<Ti* abort*>*. Transaction *Ti* has aborted.

## Recovery Techniques

A computer system, like any other device, is subject to failure from a variety of causes: disk crash, power outage, software error, a fire in the machine room, even sabotage. A failure is a state where data inconsistency is visible to transactions if they are scheduled for execution. 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.

Database recovery is the process of restoring database to a correct state in the event of a failure. A database recovery is the process of eliminating the effects of a failure from the database. ***Recovery***, in database systems terminology, is called **restoring the last consistent state of the data items**.

**Recovery Facilities**

* DBMS should provide following facilities to assist with recovery:
  + **Backup mechanism:** that makes periodic backup copies of database.
  + **Logging facility:** that keeps track of current state of transactions and database changes.
  + **Checkpoint facility:** that enables updates to database in progress to be made permanent.
  + **Recovery manger:** This allows DBMS to restore the database to a consistent state following a failure.

Damage to the database could be either physical and relate which will result in the loss of the data stored or just inconsistency of the database state after the failure. For each we can have a recover mechanism:

1. If database has been damaged:
   * Need to restore last backup copy of database and reapply updates of committed transactions using log file.
   * Extensive damage/catastrophic failure: physical media failure; is restored by using the backup copy and by re executing the committed transactions from the log up to the time of failure.
2. If database is only inconsistent:
   * No physical damage/only inconsistent: the restoring is done by reversing the changes made on the database by consulting the transaction log.
   * Need to undo changes that caused inconsistency. May also need to redo some transactions to ensure updates reach secondary storage.
   * Do not need backup, but can restore database using before- and after-images in the log file.

**Recovery Techniques for Inconsistent Database State**

Recovery is required if only the database is updated. The kind of recovery also depends on the kind of update made on the database.

Restoring the database means transforming the state of the database to the immediate good state before the failure. To do this, the change made on the database should be preserved. Such kind of information is stored in a ***system log*** or ***transaction log*** file.

The database buffers occupy an area in main memory from which data is transferred to and from secondary storage. Only once the buffers have been **flushed** to secondary storage can any update operations be regarded as permanent. This flushing of the buffers to the database can be triggered by a specific command (for example, transaction commit) or automatically when the buffers become full. The explicit writing of the buffers to secondary storage is known as **force-writing**.

If a failure occurs between writing to the buffers and flushing the buffers to secondary storage, the recovery manager must determine the status of the transaction that performed the write at the time of failure. If the transaction had issued its commit, then to ensure durability the recovery manager would have to **redo** that transaction’s updates to the database (also known as **roll forward**).

On the other hand, if the transaction had not committed at the time of failure, then the recovery manager would have to **undo** (**rollback**) any effects of that transaction on the database to guarantee transaction atomicity.

If only one transaction has to be undone, this is referred to as **partial undo**. A partial undo can be triggered by the scheduler when a transaction is rolled back and restarted as a result of the concurrency control protocol, as described in the previous section. A transaction can also be aborted unilaterally, for example, by the user or by an exception condition in the application program. When all active transactions have to be undone, this is referred to as **global undo**.

**Three main recovery techniques:**

1. Deferred Update
2. Immediate Update
3. Shadow Paging:

The Deferred Update and Immediate Update are log based approaches of database recovery, due to this they serve on non-catastrophic failure.

* + Recovery from catastrophic
    - If there is extensive damage to the wide portion of the database
    - This method restore a past copy of the database from the backup storage and reconstructs operation of a committed transaction from the back up log up to the time of failure
  + Recovery from non-catastrophic failure
    - When the database is not physically damaged but has be come inconsistent
    - The strategy uses *undoing and redoing* some operations in order to restore to a consistent state :

The shadow paging is an alternative approach to the deferred update. The modified version of a data item does not overwrite its disk copy but is written at a separate disk location.

### **Deferred Update**

Updates are not written to the database until after a transaction has reached its commit point. If transaction fails before commit, it will not have modified database and so no undoing of changes required. May be necessary to redo updates of committed transactions as their effect may not have reached database.

A transaction first modifies all its data items and then writes all its updates to the final copy of the database. No change is going to be recorded on the database before commit. The changes will be made only on the local transaction workplace. Update on the actual database is made after commit and after the change is recorded on the log. Since there is no need to perform undo operation it is also called NO-UNDO/REDO Algorithm

Example transactions T*0*and *T1*(*T0*executes before *T1*):

*T0*: **read** (*A*) *T1*: **read** (*C*)

*A: - A - 50* *C:- C- 100*

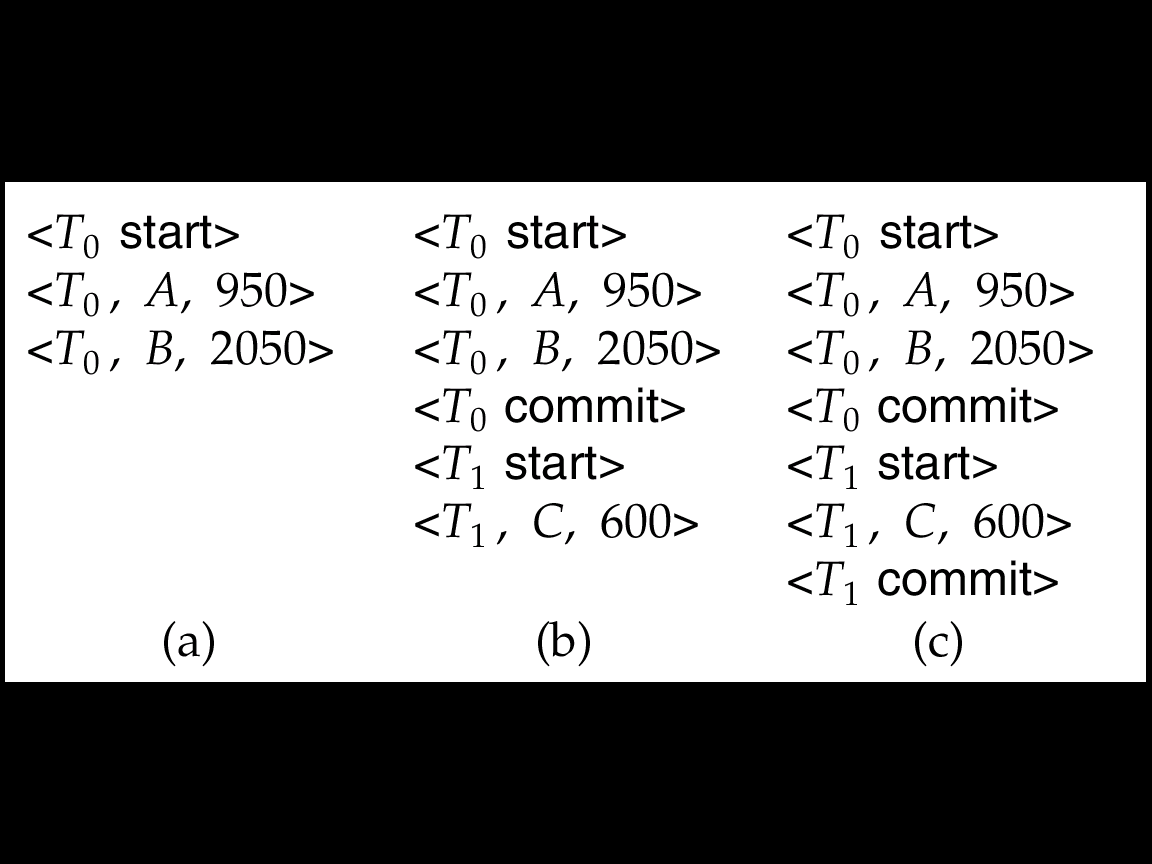
**Write** (*A*) **write** (*C*)

**read** (*B*)

*B:- B + 50*

**write** (*B*)

On these three scenarios initial value of A=1000 , B=2000, and C=700



Based on the above log, 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 **commi**t> is present

(c) **redo**(*T*0) must be performed followed by redo(*T*1) since

<*T*0**commit**> and <*Ti* commit> are present

### **Immediate Update/ Update-In-Place**

Using the *immediate update* recovery protocol, updates are applied to the databases they occur without waiting to reach the commit point. As well as having to redo the updates of committed transactions following a failure, it may now be necessary to undo the effects of transactions that had not committed at the time of failure.

As soon as a transaction updates a data item, it updates the final copy of the database on the database disk. During making the update, the change will be recorded on the transaction log to permit rollback operation in case of failure. UNDO and REDO are required to make the transaction consistent. Thus, it is called **UNDO/REDO Algorithm.** This algorithm will undo all updates made in place before commit. The redo is required because some operations which are completed but not committed should go to the database. If we don’t have the second scenario, then the other variation of this algorithm is called **UNDO/NO-REDO Algorithm.**

### **Shadow Paging**

This scheme maintains two page tables during life of a transaction: current page and shadow page table. When transaction starts, two pages are the same. Shadow page table is never changed thereafter and is used to restore database in event of failure. During transaction, current page table records all updates to database. When transaction completes, current page table becomes shadow page table.

Shadow paging has several advantages over the log-based schemes: the overhead of maintaining the log file is eliminated and recoveries is significantly faster as there is no need for undo or redo operations. However, it has disadvantages as well, such as data fragmentation and the need for periodic garbage collection to reclaim inaccessible blocks.

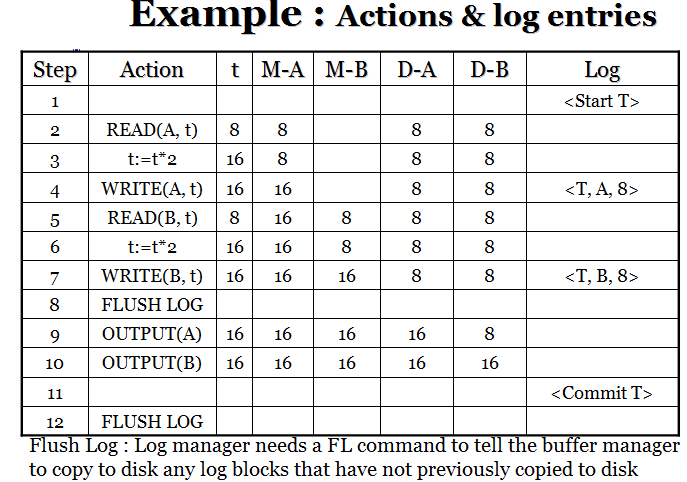
The three styles of logging for database recovery are:

* + **Undo**: Restore all BFIMs on to disk (Remove all AFIMs)
  + **Redo**: Restore all AFIMs on to disk
  + **Undo/Redo** restore both BFIM and AFIM

Database recovery is achieved either by performing only Undos or only Redos or by a combination of the two. These operations are also recorded in the log as they happen.

**Summary:**

Material associated with one transaction must be written to disk in the following order:

* U1 : Log records indicating changed DB elements
* U2 : Changed DB elements themselves
* U3 : COMMIT log record

**Recovery using Undo Logging**

Simple form of recovery assumes that, any time failure may occur. This failure may have brought certain changes written to disk, where as some changes that should occur may not be written to disk. T recover using undo the following are series of steps performed:

1. Look at the entire log and Divide the transaction into committed and uncommitted (active) transactions
2. Scan the log end by recovery manager, if it sees <T, X, V>

If <Commit T> is found: Transaction could not have left DB in an inconsistent state

If <Commit T> is not found: Some changes may be in disk and other changes in MM, not copied to disk. This means T is incomplete or aborted transaction, the recovery manger must Use <T, X, v> to undo the changes, the value of X to be V.

1. After this write <ABORT T> for each incomplete transaction
2. Then flush log the transaction.

On the above example, what will happen if a crash occurs;

* 1. Immediately After step 12: we know the <COMMIT> record got to the disk before the crash. No need of undo, all records by the record manger are ignored
  2. Immediately After step 11: it is seen that the <COMMIT> got flushed to the disk, so no need of undo. If commit is not flushed the action to be taken will be similar to the one mentioned on number 3 next.
  3. At step 11: However if the <COMMIT> record never reached to disk the recovery manger considers T is incomplete it therefore stores 8 as the value of B, and continue on scanning, it will make the value of A to 8 again and <ABORT T> is written to the log, and the log is flushed.
  4. Immediately After step 10: now the <COMMIT> record surely not written, so T is incomplete and it is undone like step three above..
  5. Immediately After step 7: now it is not certain that whether any of the records reached to disk, therefore the recovery manger will undo the changes.
  6. Immediately After step 4: no record is known about database element B. there is no <COMMIT> record observed. Due to this the recovery manger will undo the change to A and <ABORT T>. Note that no new value, 16, is mentioned on undo log.

## Check pointing

Recovery require entire log to be examined due to this Recovery is very, very SLOW! The simplest way to untangle potential problems is to checkpoint the log periodically. If a transaction has its commit log recorded on the disk, log record of that transaction are no longer needed. The solution is a simple check pointing.

**Check pointing**

Checkpoint: is a point of synchronization between database and a transaction log file. All buffers are force-written to secondary storage. Checkpoint record is created containing identifiers of all active transactions. When failure occurs, redo all transactions that committed since the checkpoint and undo all transactions active at time of crash.