

# **CS470 Introduction to Database Systems**

## **Database Recovery**

**V Kumar  
Department of Computer Networking  
University of Missouri-Kansas City**

## Database Recovery

A *database recovery is the process of eliminating the effects of a failure from the database*. A failure is a state where data inconsistency is visible to transactions if they are scheduled for execution. In databases usually three types of failure are encountered:

**Transaction failure:** a transaction cannot continue with its execution, therefore, it is aborted and if desired it may be re-executed at some other time. Reasons: Deadlock, time-out, protection violation, or system error.

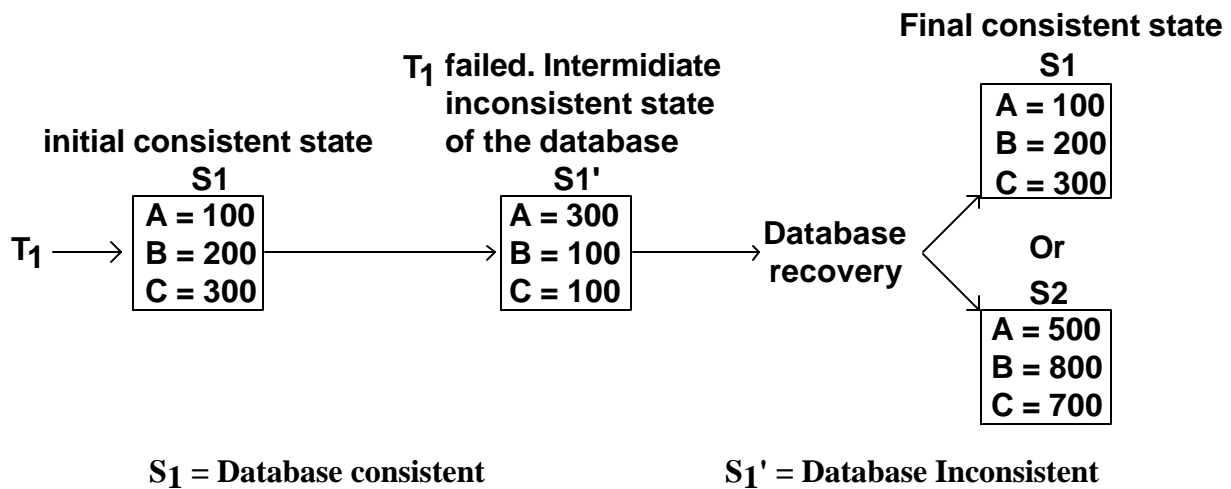
**System failure:** the database system is unable to process any transactions. Some of the common reasons of system failure are: wrong data input, register overflow, addressing error, power failure, memory failure, etc.

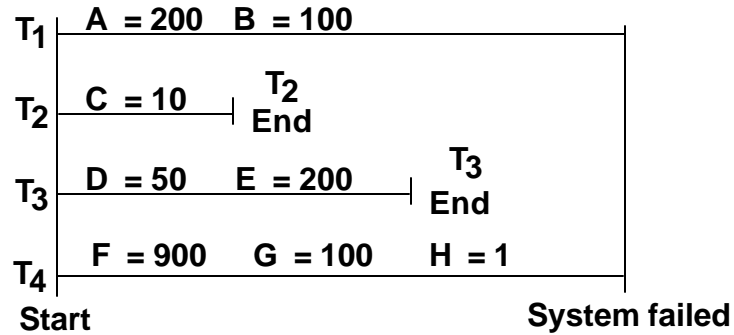
**Media failure:** failure of non-volatile storage media (mainly disk). Some of the common reasons are: head crash, dust on the recording surfaces, fire, etc.

We will discuss transaction and system failures and database recovery from these failures. We will not discuss database recovery from media 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. In database systems terminology it is called restoring the last consistent state of the data items.

The execution of a transaction T is correct if, given a consistent database, T leaves the database in the next consistent state. Initial consistent state of the database:  $S_1$ . T changes the consistent state from  $S_1$  to  $S_2$ . During this transition the database may go into an inconsistent state, which is visible only to T. If the system or T fails then database will not be able to reach  $S_2$ , consequently the visible state would be an inconsistent state. From this state we can go either forward (to state  $S_2$ ) or go backward (to state  $S_1$ ).





If we go back to S<sub>1</sub> state we lose the updates of T<sub>2</sub> and T<sub>3</sub>. If we want to reach S<sub>2</sub> (consistent state) then we have to roll back T<sub>1</sub> and T<sub>4</sub> and re-run them again. A good idea is to do both.

## Recovery Algorithms

We begin our discussion of tools/subsystems/data structures used in recovery algorithms. The log has complete information for UNDO (roll-back) and REDO (forward processing) operation. For protection the log is saved on log disks, which are separate from database disk.

**Transaction Log:** Execution history of concurrent transactions. Example

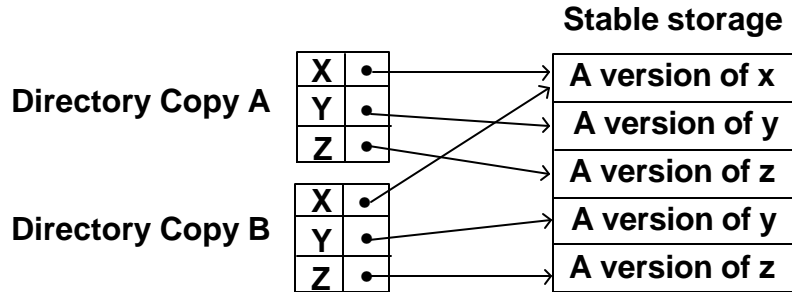
T ID	Back P	Next P	Time	Op	Item	BFIM	AFIM
T1	0	1	11:42	Start			
T1	1	4	11:43	Write	A	A=100	A=200
T2	0	8	11:46	Start			
T1	2	5	11:47	W	B	B=50	B=100
T1	4	7	11:48	R	C	C=200	C=200
T3	0	9	11:49	R	D	D=400	D=400
T1	5	nil	11:50	End			
T2	3	10	11:59	W	E	E=10	E=200
T3	6	20	12:00	Insert	F		F=40
T2	8	nil	12:05	End			

**Database update:** A transaction's updates to the database can be applied in two ways:

**Update-in-place:** As soon as a transaction updates a data item, it updates the final copy of the database on the database disk.

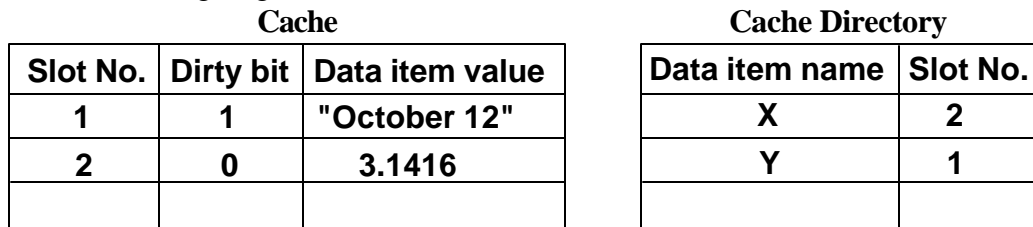
**Deferred update:** A transaction first modifies all its data items and then writes all its updates to the final copy of the database.

**Stable Storage:** Disk. Update method: (a) Update-in-place and (b) shadow. In shadow each database state is may need to be maintained. The following diagram illustrates this update process. Directory copies A and B are two database states.



The granularity of data items that are parameters to write may be different and this size disparity must be considered in designing a recovery algorithms. We will see the effect of this mismatch very soon.

**Cache Manager:** To minimize I/O during transaction execution, a copy of the desired data items may be kept in the main memory. To read a data item, simply obtain its value from the main memory copy. To write, record the new value both in stable store and in the main memory. The portion of the main memory that holds a copy of the data items is called cache and the process is called caching or buffering. Since cache is shared by many transactions, it is managed by a subsystem called Cache Manager. The following diagram shows the structure of a cache.



**Dirty bit = 1** meaning the value of this data item is different than its disk copy value. This means this data item has been written to by a transaction and it must be flushed to the disk at an appropriate time.

CM uses Flush and Fetch to manage the cache. To read a data item named x, the CM fetches the value of x if it is not in the cache, and returns this value from the cache. To write x, the CM allocates a slot for x if it is not already in the cache, records the *new x* in a cache slot, and sets the dirty bit for the cache slot. Whether it flushes *new x* to stable storage at this point or later on is a decision left to RM. This flushing of data item from cache to stable storage can be done at different times and each method defines a different recovery algorithm.

CM uses two additional operations to synchronize flush and fetch operations. At times it is necessary that the cache slot be not flushed to disk, for example, while updating the contents of the slot. For this reason, the CM uses two additional operations: PIN and UNPIN.

**PIN:** tells the CM not to flush the slot until it is unpinned.

**UNPIN:** makes a previously pinned slot available for flushing.

**Physical logging:** Logging actual values of data items and nothing about the type of operation.

## Recovery Algorithms

We are now in a position to discuss mechanism for recovering database from a failure. We identify the following operations:

**RM-read (Ti, old x):** read the value of *old x* for Ti.

**RM-write (Ti, old x, new x):** overwrite the value of *old x* with *new x* for transaction Ti.

**RM-commit (Ti):** Commit Ti.

**RM-abort(Ti):** Abort Ti and

**Restart:** bring the database in its last consistent state after a system failure.

RM-abort and Restart are managed by two operations: UNDO and REDO, which can be defined by the following rules. These two rules ensure that the last committed value of a data item is always available in stable storage.

**Undo Rule:** The *old x* (last committed value of *x* in the stable database) must be saved in a stable place (log) before it is overwritten by *new x* in the stable database by an uncommitted transaction. (This is often called Write Ahead Logging - WAL).

**Redo Rule:** The *new x* must be saved in a stable place (usually log or it may be stable database) before a transaction can commit.

**Idempotent rule:** A system can fail during a restart also. The recovery system will regard as a system failure and initiate recovery operation. To handle this situation, the system must make sure that the result of one execution of recovery algorithm is the same as  $n$  ( $n > 1$ ) execution of the recovery algorithm. This property is called *idempotent*.

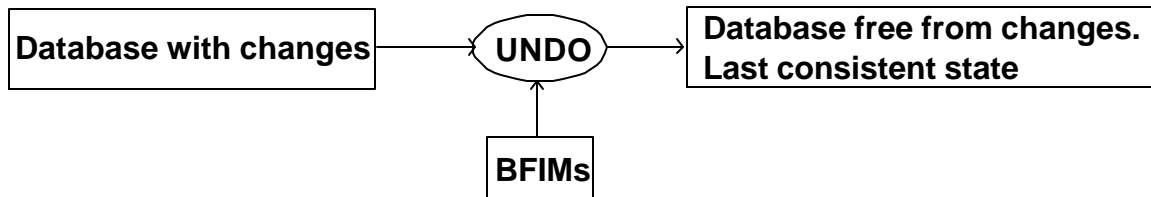
Undo and redo can be combined in four different ways and each combination defines a new recovery algorithm. We will discuss recovery algorithms based on update-in-place and deferred update. The main aim of recovery algorithms is to make sure that up to date copy of log is available to the recovery algorithm. We begin with undo/redo combination.

## UNDO/REDO Algorithm

In this algorithm the recovery process uses undo and redo operations to recover the last consistent state of the database. Undo is required since the database is updated in-place. Redo is required because some updates by a completed (not committed) transaction may not have gone to the database. Suppose transaction T updates A and B. The steps of this algorithm can be summarized as follows:

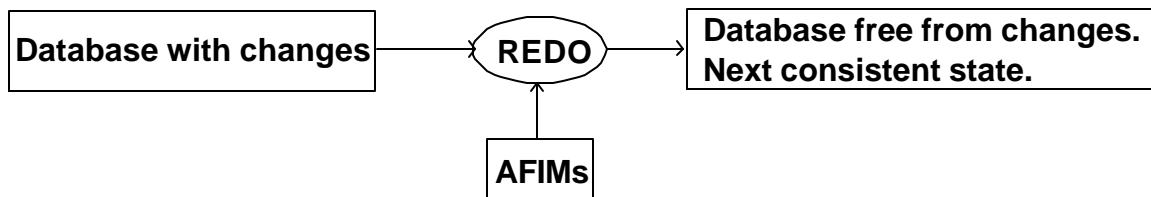
1. Get data item A in cache if it is not there.
2. Write BFIM of A (oldA) in the log (log in volatile memory).
3. Modify A in cache (newA).
4. Write AFIM of A in the log (log in volatile memory).
5. Save log file on the log disk.
6. After the log file successfully written to the log disk, write A's AFIM in the final database.

(At this point A's BFIM is safe in the log on the log disk). If system fails then the transaction must be undone since it is not yet committed.



7. Get data item B (oldB) in cache if it is not already there.
8. Write BFIM of B in the log (log in volatile memory).
9. Modify B in cache.
10. Write AFIM of B in the log (log in volatile memory).
11. Flush log file to the log disk.
12. After log file successfully written to the log disk, start writing AFIM of B to the final copy of the database. If the system fails at this point, i.e., before writing AFIM of B to the database, we have the following situation:
  - a. AFIM and BFIMs of A and B are in the log and log is safe on the disk.
  - b. AFIM of A is in the final database but AFIM of B is not.

If the system fails at this point, T must be redone since T's complete execution history is saved in the log on the disk. It is not necessary to undo T and then execute it again.



13. If there is no system failure then commit the transaction. At this point the following situation exists:
  - a. AFIMs and BFIMs of A and B are in the log and log is on the disk.
  - b. AFIMs and BFIMs of A and B are in the final database on the disk.

If T is a withdrawal transaction then its updates are durable and the message (transaction complete and the money) is sent to the user.

## **N0-UNDO/REDO Algorithm**

In this algorithm the recovery manager only requires redo to recover the database from a system failure. This means that a transaction does not update the database in-place. It records only the AFIMs of data items in the log. At the time of commit the final database is updated. The steps are:

1. Get the data item A in cache if it is not there.
2. Modify A in cache.
3. Write AFIM of A in the log (log in volatile memory).

4. Flush log on the log disk. If the system fails at this point, the recover manager just discards all AFIMs of data items (in our case of A) written by this transaction in the log since the final database has not been updated.
5. Get B in cache if it is not there.
6. Modify B in cache.
7. Write AFIM of B in the log (log in volatile memory).
8. Flush log on the log disk.
9. Write commit record on the log and save the log on the disk. If the system fails at this point then all AFIMs are in the log. The recovery manager during recovery gets AFIMs of A and B and updates the final database.
10. Update the final database. At the end of this update send message to the user.

## NO-UNDO/NO-REDO algorithm

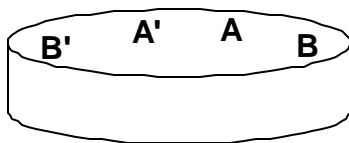
In this algorithm undo and redo are not used to recover the database from a system failure. As mentioned before, undo is required only when database is updated in-place. In the absence of in-place update undo is not necessary. Similarly, if we do not write AFIM in the log then redo is not necessary. These operations are eliminated by writing AFIMs of data items at a separate place on the database disk. The final database is not touched during the execution of a transaction. When the transaction has completed its execution then all AFIMs replaces their old values in the final database.

**Transaction:** T.

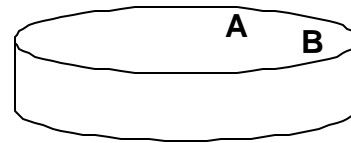
**Data items:** A and B.

**Execution:** write (A) generates A' (AFIM)  
write (B) generates B' (AFIM)

At commit, A' and B' replace A and B as shown in the diagram.



**Database before T commits**



**Database after T commits**

### Steps

1. Get data item A.
2. Modify A.
3. Create its shadow (A') on the database disk.
4. Get B
5. Modify B.
6. Create its shadow (B') on the database disk. If the system fails at this point, no harm done. The recovery manager simply deletes the shadows of A and B, i.e., A' and B'. The transaction is re-executed at some later time.
7. At commit, first copy A' and B' to A and B in final database and then send the message to the user.

## Database Recovery

We discussed three recovery algorithms in brief. The implementation of these algorithms are quite complex. Each algorithm has some good points and some bad points. Database recovery is a time consuming activity and, depending upon the size of the database, the recovery may take somewhere from 30-50 mins. Most commonly used recovery algorithm is Undo/Redo.