CONCURRENCY CONTROL

By - ANUPAMA

Concurrency Control

Concurrency Control

- A database must provide a mechanism that will ensure that all possible schedules are both:
  - Conflict serializable
  - Recoverable and preferably cascadeless
- A policy in which only one transaction can execute at a time generates serial schedules, but provides a poor degree of concurrency
- Concurrency-control schemes tradeoff between the amount of concurrency they allow and the amount of overhead that they incur
- Testing a schedule for serializability *after* it has executed is a little too late!
  - Tests for serializability help us understand why a concurrency control protocol is correct
- **Goal** – to develop concurrency control protocols that will assure serializability

Concurrency Control

Concurrency Control

- One way to **ensure isolation** is to require that data items be accessed in a **mutually exclusive manner**; that is, while one transaction is accessing a data item, no other transaction can modify that data item
  - Should a transaction hold a lock on the whole database
- Would lead to strictly serial schedules – very poor performance

- The most common method used to implement locking requirement is to allow a transaction to access a data item only if it is currently holding a **lock** on that item

LOCK-BASED PROTOCOLS

Lock-Based Protocols

Lock-Based Protocols

- A lock is a mechanism to control concurrent access to a data item
- **Lock requests are made to the concurrency-control manager by the programmer**

- **Types of locks :**
- Data items can be locked in two modes :
  1. *exclusive (X) mode*. Data item can be both read as well as written. X-lock is requested using **lock-X** instruction
  2. *shared (S) mode*. Data item can only be read. S-lock is requested using **lock-S** instruction
- A transaction can unlock a data item *Q* by the **unlock**(*Q*) Instruction
- Transaction can proceed only after request is granted

Lock-Based Protocols

Lock-Based Protocols

- **Lock-compatibility matrix**

	S	X
S	true	false
X	false	false

- **Rule 1:**
  - **A transaction may be granted a lock on an item if the requested lock is compatible with locks already held on the item by other transactions**
- **Rule 2:**
  - **Any number of transactions can hold shared locks on an item,**
  - **But if any transaction holds an exclusive on the item no other transaction may hold any lock on the item**
- **Rule 3:**
  - **If a lock cannot be granted, the requesting transaction is made to wait till all incompatible locks held by other transactions have been released. The lock is then granted**

Lock-Based Protocols(Cont...)

- Rule 4:
  - Transaction *Ti* may unlock a data item that it had locked at some earlier point

  Moreover, it is not necessarily desirable for a transaction to unlock a data item immediately after its final access of that data item, since serializability may not be ensured

- **Note that a transaction must hold a lock on a data item as long as it accesses that item.**

Lock-Based Protocols: Example

Lock-Based Protocols: Example

- Let *A* and *B* be two accounts that are accessed by transactions *T*1 and *T*2.
  - Transaction *T*1 transfers $50 from account *B* to account *A.*
  - Transaction *T*2 displays the total amount of money in accounts *A* and *B*, that is, the sum *A* + *B*
  - Suppose that the values of accounts *A* and *B* are $100 and $200, respectively

*T*1:		T2:
lock-X(*B*);		lock-S(A);
read(*B*);		read(A);
	B := *B* − 50;	unlock(A);
write(*B*);		lock-S(B);
unlock(*B*);		read(B);
lock-X(*A*);		unlock(B);
read(*A*);		display(A + B)
	A := *A* + 50;	
write(*A*);		
unlock(*A*);		

- If these transactions are executed serially, either as *T*1, *T*2 or the order *T*2, *T*1, then transaction *T*2 will display the value $300

Lock-Based Protocols: Example

Lock-Based Protocols: Example

- If, however, these transactions are executed concurrently, then schedule 1 is possible
- In this case, transaction *T*2 displays $250, which is incorrect. The reason for this mistake is that
  - the transaction *T*1 unlocked data item *B* too early, as a result of which *T*2 saw an inconsistent state
- Suppose we delay unlocking till the end

T_1	T_2	concurreny-control manager
lock-X(B)		
		grant-X(B, T_1)
read(B)		
$B := B - 50$		
write(B)		
unlock(B)		
	lock-S(A)	
		grant-S(A, T_2)
	read(A)	
	unlock(A)	
	lock-S(B)	
		grant-S(B, T_2)
	read(B)	
	unlock(B)	
	display($A + B$)	
lock-X(A)		
		grant-X(A, T_1)
read(A)		
$A := A - 50$		
write(A)		
unlock(A)		

Schedule 1

Lock-Based Protocols: Example

- **Delaying unlocking till the end, T1 becomes T3 and T2 becomes T4**

*T*3:

```
lock-X(B);
read(B);
    B := B − 50;
write(B);
lock-X(A);
read(A);
    A := A + 50;
write(A);
unlock(B);
unlock(A)
```

*T*4:

```
lock-S(A);
read(A);
lock-S(B);
read(B);
display(A + B);
unlock(A);
unlock(B)
```

- Hence, sequence of reads and writes as in Schedule 1 is no longer possible
- T4 will correctly display $300

T_1	T_2	concurreny-control manager
lock-X(B)		
		grant-X(B, T_1)
read(B)		
$B := B - 50$		
write(B)		
unlock(B)		
	lock-S(A)	
		grant-S(A, T_2)
	read(A)	
	unlock(A)	
	lock-S(B)	
		grant-S(B, T_2)
	read(B)	
	unlock(B)	
	display($A + B$)	
lock-X(A)		
		grant-X(A, T_1)
read(A)		
$A := A - 50$		
write(A)		
unlock(A)		

Schedule 1

Lock-Based Protocols: Example

Lock-Based Protocols: Example

- Given, T3 and T4, consider Schedule 2 (partial)
- Since *T*3 is holding an exclusive mode lock on *B* and *T*4 is requesting a shared-mode lock on *B*, *T*4 is waiting for *T*3 to unlock *B*
- Similarly, since *T*4 is holding a shared-mode lock on *A* and *T*3 is requesting an exclusive-mode lock on *A*, *T*3 is waiting for *T*4 to unlock *A*
- Thus, we have arrived at a state where neither of these transactions can ever proceed with its normal execution
- This situation is called **deadlock**

*T*3:

```
lock-X(B);
read(B);
  B := B − 50;
write(B);
lock-X(A);

read(A);
  A := A + 50;
write(A);
unlock(B);
unlock(A)
```

*T*4:

```
lock-S(A);
read(A);
lock-S(B);
read(B);
display(A +
B);
unlock(A);
unlock(B)
```

T_3	T_4
lock-X(B)	
read(B)	
$B := B - 50$	
write(B)	
	lock-S(A)
	read(A)
	lock-S(B)
lock-X(A)	

Schedule 2

Lock-Based Protocols: Example

- Solution to deadlock : rollback
  - When deadlock occurs, the system must roll back one of the two transactions.
  - Once a transaction has been rolled back, the data items that were locked by that transaction are unlocked
  - These data items are then available to the other transaction, which can continue with its execution

Lock-Based Protocols

Lock-Based Protocols

- Problem associated with LOCK BASED PROTOCOLS

1) If we do not use locking, or if we unlock data items too soon after reading or writing them, we may get inconsistent states

2) On the other hand, if we do not unlock a data item before requesting a lock on another data item, deadlocks may occur

3) Deadlocks are a necessary evil associated with locking, if we want to avoid inconsistent states

4) Deadlocks are definitely preferable to inconsistent states, since they can be handled by rolling back transactions, whereas inconsistent states may lead to real-world problems that cannot be handled by the database system

Lock-Based Protocols

- A **locking protocol** is a set of rules followed by all transactions while requesting and releasing locks

- Locking protocols restrict the set of possible schedules

The Two-Phase Locking(2PL) Protocol

The Two-Phase Locking Protocol

- This protocol ensures conflict-serializable schedules
- **Phase 1: Growing Phase**
  - Transaction may obtain locks
  - Transaction may not release locks
- **Phase 2: Shrinking Phase**
  - Transaction may release locks
  - Transaction may not obtain locks
- The protocol assures serializability. It can be proved that the transactions can be serialized in the order of their **lock points**

- **Note: If any schedule is not conflict serializable, then we can't apply 2-PL also.**

Lock Conversions

- Two-phase locking with lock conversions:
  - First Phase:
    - can acquire a lock-S on item
    - can acquire a lock-X on item
    - can convert a lock-S to a lock-X (upgrade)
  - Second Phase:
    - can release a lock-S
    - can release a lock-X
    - can convert a lock-X to a lock-S (downgrade)
- This protocol assures serializability. But still relies on the programmer to insert the various locking instructions

Deadlocks

- Two-phase locking *does not* ensure freedom from deadlocks

*T*3:

lock-X(*B*);
read(*B*);
B := *B* − 50;
write(*B*);
lock-X(*A*);
read(*A*);
A := *A* + 50;
write(*A*);
unlock(*B*);
unlock(*A*)

*T*4:

lock-S(*A*);
read(*A*);
lock-S(*B*);
read(*B*);
display(*A* + *B*);
unlock(*A*);
unlock(*B*)

T_3	T_4
lock-x (*B*)	
read (*B*)	
B := *B* - 50	
write (*B*)	
	lock-s (*A*)
	read (*A*)
	lock-s (*B*)
lock-x (*A*)	

- Observe that transactions *T*3 and *T*4 are two phase, but, in deadlock

Starvation

- In addition to deadlocks, there is a possibility of **starvation**
- **Starvation** occurs if the concurrency control manager is badly designed. For example:
  - A transaction may be waiting for an X-lock on an item, while a sequence of other transactions request and are granted an S-lock on the same item
  - The same transaction is repeatedly rolled back due to deadlocks
- Concurrency control manager can be designed to prevent starvation

Cascading roll-back

Cascading roll-back

- The potential for deadlock exists in most locking protocols. Deadlocks are a necessary evil
- When a deadlock occurs there is a possibility of cascading roll-backs
- Cascading roll-back is possible under two-phase locking
- In the schedule here, each transaction observes the two-phase locking protocol, but the failure of $T5$ after the read(A) step of $T7$ leads to cascading rollback of $T6$ and $T7$.

T_5	T_6	T_7
lock-X(A)		
read(A)		
lock-S(B)		
read(B)		
write(A)		
unlock(A)		
	lock-X(A)	
	read(A)	
	write(A)	
	unlock(A)	
		lock-S(A)
		read(A)

More Two Phase Locking Protocols

More Two Phase Locking Protocols

- To avoid Cascading roll-back, follow a modified protocol called **strict two-phase locking**
  - a transaction must hold all its exclusive locks till it commits/aborts

- **Rigorous two-phase locking** is even stricter.
  - *All* locks are held till commit/abort. In this protocol,

  transactions can be serialized in the order in which they commit

strict two-phase locking

Rule 1:

All locks are released after commit transactions i.e. either after read/write.

- When multiple transaction are there, then T1 will complete first then T2,T3,T4....
- When multiple transaction are done on different variable i.e. no parallel lock is there, then only strict 2PL will be followed.
- Strict 2PL also allow recoverable schedule.

Rigourous 2PL

- It is 2PL
- If lock is acquired for write, then it can be unlock only after commit statement

- X-lock(A)
- R(A)
- W(X)
- Commit
- Unlock
    - S-lock(A)
    - R(A)……..

DEADLOCK HANDLING

Deadlock Handling

Deadlock Handling

- System is deadlocked if there is a set of transactions such that every transaction in the set is waiting for another transaction in the set

- ***Deadlock prevention*** protocols ensure that the system will *never* enter into a deadlock state. Some prevention strategies :
  - Require that each transaction locks all its data items before it begins execution (predeclaration)
  - Impose partial ordering of all data items and require that a transaction can lock data items only in the order specified by the partial order

Deadlock Prevention

Deadlock Prevention

- Following schemes use transaction timestamps for the sake of deadlock prevention alone
- **wait-die** scheme — non-preemptive
  - Older transaction may wait for younger one to release data item. (older means smaller timestamp)
- Younger transactions never wait for older ones; they are rolled back instead
  - A transaction may die several times before acquiring needed data item
- **wound-wait** scheme — preemptive
  - Older transaction *wounds* (forces rollback) of younger transaction instead of waiting for it
- Younger transactions may wait for older ones
  - May be fewer rollbacks than *wait-die* scheme

Similarity and difference between wait-die and wound wait

In **both** the **wait-die** and the **wound-wait** scheme:

- The ***older*** **transaction** will **"win"** over the ***newer*** **transaction**

- **Wait-die:**
  - The ***newer*** **transactions** are ***killed*** when:
    - ***It*** **(= the *newer* transaction) *makes* a reqeust** for a **lock** being **held** by an ***older*** **transactions**
- **Wound-wait:**
  - The ***newer*** **transactions** are ***killed*** when:
    - An ***older*** **transaction *makes* a request** for a **lock** being **held** by the ***newer*** **transactions**

Deadlock Prevention

Deadlock Prevention

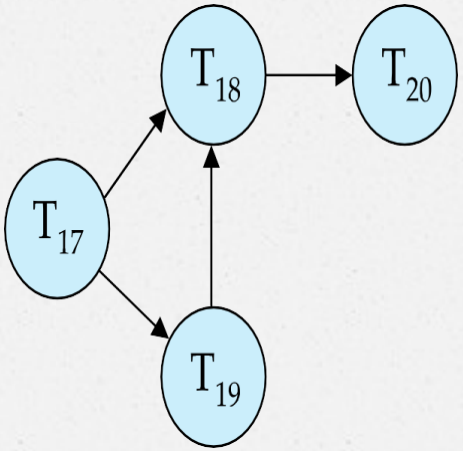
- Both in *wait-die* and in *wound-wait* schemes, a rolled back transactions is restarted with its original timestamp. Older transactions thus have precedence over newer ones, and starvation is hence avoided
- **Timeout-Based Schemes:**
  - a transaction waits for a lock only for a specified amount of time. If the lock has not been granted within that time, the transaction is rolled back and restarted,
  - Thus, deadlocks are not possible
  - simple to implement; but starvation is possible. Also difficult to determine good value of the timeout interval

Deadlock Detection

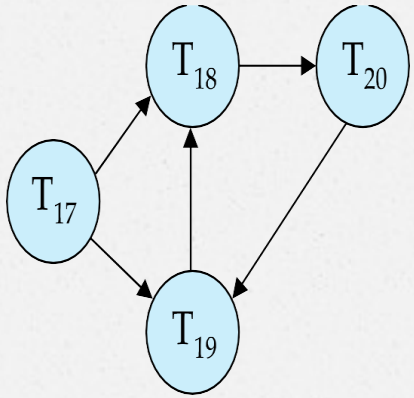
Deadlock Detection

- Deadlocks can be described as a *wait-for graph*, which consists of a pair $G = (V,E)$,
  - V is a set of vertices (all the transactions in the system)
  - E is a set of edges; each element is an ordered pair $T_i \rightarrow T_j$.
- If $T_i \rightarrow T_j$ is in E, then there is a directed edge from T_i to T_j, implying that T_i is waiting for T_j to release a data item
- When T_i requests a data item currently being held by T_j, then the edge $T_i \rightarrow T_j$ is inserted in the wait-for graph. This edge is removed only when T_j is no longer holding a data item needed by T_i
- The system is in a deadlock state if and only if the wait-for graph has a cycle. Must invoke a deadlock-detection algorithm periodically to look for cycles

Deadlock Detection: Example



Wait-for graph without a cycle



Wait-for graph with a cycle

Deadlock Recovery

Deadlock Recovery

- When deadlock is detected :
  - Some transaction will have to rolled back (made a victim) to break deadlock. Select that transaction as victim that will incur minimum cost
  - Rollback -- determine how far to roll back transaction
- Total rollback: Abort the transaction and then restart it
- More effective to roll back transaction only as far as necessary to break deadlock
  - Starvation happens if same transaction is always chosen as victim. Include the number of rollbacks in the cost factor to avoid starvation

Validation-Based Protocol

Validation-Based Protocol

- Execution of transaction T_i is done in three phases.
  1. **Read and execution phase**: Transaction T_i writes only to temporary local variables
  2. **Validation phase**: Transaction T_i performs a "validation test" to determine if local variables can be written without violating serializability.
  3. **Write phase**: If T_i is validated, the updates are applied to the database; otherwise, T_i is rolled back.

- The three phases of concurrently executing transactions can be interleaved, but each transaction must go through the three phases in that order.
- Also called as **optimistic concurrency control** since transaction executes fully in the hope that all will go well during validation

Validation-Based Protocol (Cont.)

- Each transaction T_i has 3 timestamps
  - **Start**(T_i) : the time when T_i started its execution
  - **Validation**(T_i): the time when T_i entered its validation phase
  - **Finish**(T_i) : the time when T_i finished its write phase.

- Serializability order is determined by timestamp given at validation time, to increase concurrency. Thus $TS(T_i)$ is given the value of **Validation**(T_i).

- This protocol is useful and gives greater degree of concurrency if probability of conflicts is low. That is because the serializability order is not pre-decided and relatively less transactions will have to be rolled back.

Validation Test for Transaction T_j

- If for all T_i with TS (T_i) < TS (T_j) either one of the following condition holds:
  - **FINISH**(T_i) < **START**(T_j)
  - **START**(T_j) < **FINISH**(T_i) < **VALIDATION**(T_j) **and** the set of data items written by T_i does not intersect with the set of data items read by T_j.

  then validation succeeds and T_j can be committed. Otherwise, validation fails and T_j is aborted.

Schedule Produced by Validation

- Example of schedule produced using validation

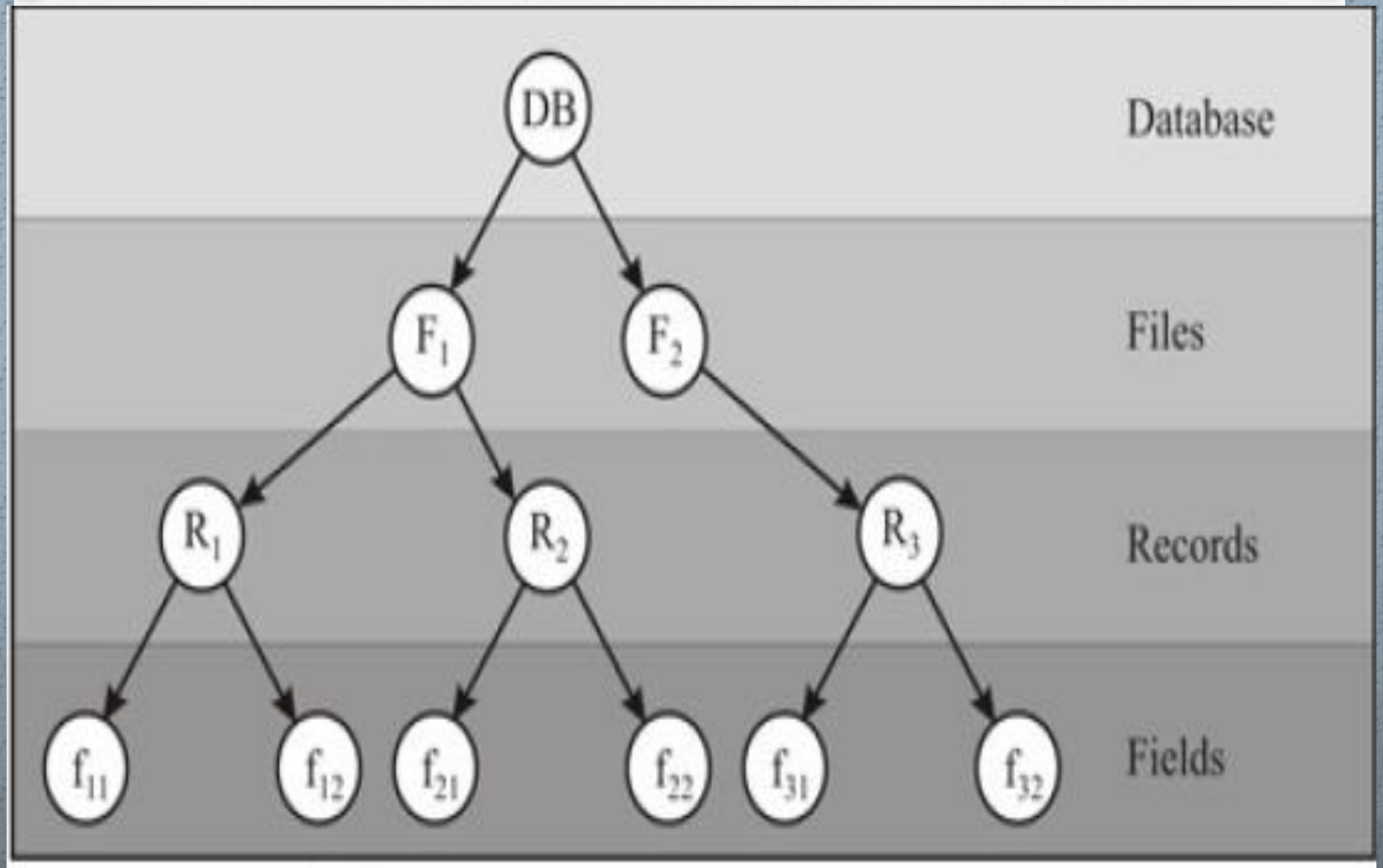
T_{14}	T_{15}
read(*B*)	
	read(*B*)
	B:- B-50
	read(*A*)
	A:- A+50
read(*A*)	
(*validate*)	
display (*A+B*)	
	(*validate*)
	write (*B*)
	write (*A*)

Multiple Granularity

- **Granularity:** It is the size of data item allowed to lock.
- **Multiple Granularity:**
  - It can be defined as hierarchically breaking up the database into blocks which can be locked.
  - The Multiple Granularity protocol enhances concurrency and reduces lock overhead.
  - It maintains the track of what to lock and how to lock.
  - It makes easy to decide either to lock a data item or to unlock a data item. This type of hierarchy can be graphically represented as a tree.

Multiple Granularity

- Granularity of locking
- (the levels in tree where locking is done) are:
  - **fine granularity (lower in the tree):** It allows high concurrency but has high locking overhead,
  - **coarse granularity (higher in the tree)**: It has low locking overhead but also has low concurrency.
- The highest level in the example hierarchy is the entire database. The levels below are file, record and field.


DB
Database
F1
F2
Files
R1
R2
R3
Records
f11
f12
f21
f22
f31
f32
Fields

Additional lock modes with multiple granularity:

- There are three additional lock modes with multiple granularity:
- Intention Mode Lock
- **Intention-shared (IS):** It contains explicit locking at a lower level of the tree but only with shared locks.
- **Intention-Exclusive (IX):** It contains explicit locking at a lower level with exclusive or shared locks.
- **Shared & Intention-Exclusive (SIX):** In this lock, the node is locked in shared mode, and some node is locked in exclusive mode by the same transaction.

- **Volatile storage**:
  - does not survive system crashes
  - examples: main memory, cache memory
- **Nonvolatile storage**:
  - survives system crashes
  - examples: disk, tape, flash memory, non-volatile (battery backed up) 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

Failure Classification

- To find that where the problem has occurred, we generalize a failure into the following categories:
- Transaction failure
- System crash
- Disk failure

1. Transaction failure

- The transaction failure occurs when it fails to execute or when it reaches a point from where it can't go any further. If a few transaction or process is hurt, then this is called as transaction failure.

- Reasons for a transaction failure could be -
  - **Logical errors:** If a transaction cannot complete due to some code error or an internal error condition, then the logical error occurs.

  - **Syntax error:** It occurs where the DBMS itself terminates an active transaction because the database system is not able to execute it.

  For example, The system aborts an active transaction, in case of deadlock or resource unavailability.

2. System Crash

- System failure can occur due to power failure or other hardware or software failure.

Example: Operating system error.

- **Fail-stop assumption:** In the system crash, non-volatile storage is assumed not to be corrupted.

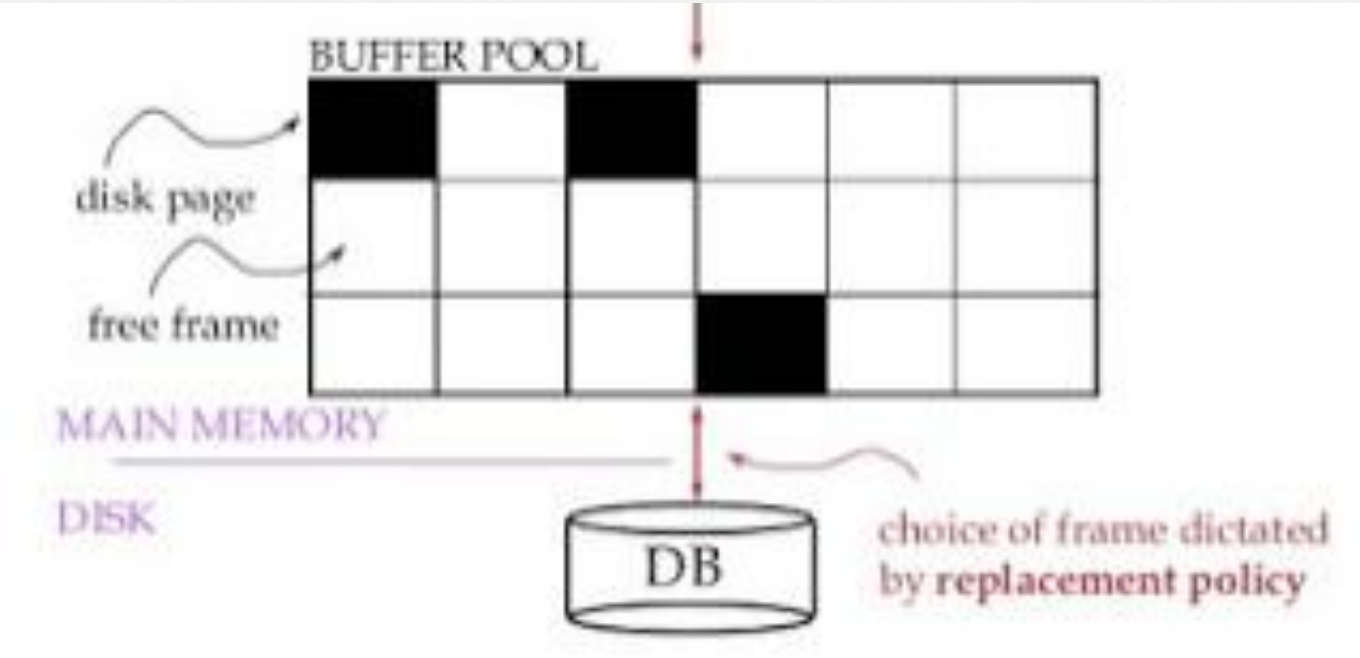
System Crash (cont...)

- **Disk Failure**
  - It occurs where hard-disk drives or storage drives used to fail frequently. It was a common problem in the early days of technology evolution.
  - Disk failure occurs due to the formation of bad sectors, disk head crash, and unreachability to the disk or any other failure, which destroy all or part of disk storage.

BUFFER MANAGEMENT

- Data must be in RAM for DBMS to operate on it!
- Buffer manager hides the fact that not all data is in RAM.

- The goal of the buffer manager is to ensure that the data requests made by programs are satisfied by copying data from secondary storage devices into buffer.



Buffer Management in DBMS

Failure with Loss of Nonvolatile Storage

- Failures in which the content of nonvolatile storage is lost are rare, we nevertheless need to be prepared to deal with this type of failure.

- The basic scheme is to **dump** the entire content of the database to stable storage periodically—say, once per day.

Failure with Loss of Nonvolatile Storage

- Dump database – At periodic interval
  - On the magnetic disk
  - Output all log records currently residing in main memory onto stable storage.
  - Output all buffer blocks onto the disk.
  - Copy the contents of the database to stable storage.

Log-Based Recovery

- **The log is a sequence of records.** Log of each transaction is maintained in some stable storage so that if any failure occurs, then it can be recovered from there.
- If any operation is performed on the database, then it will be recorded in the log.
- But the process of storing the logs should be done before the actual transaction is applied in the database.

Example : Log based recovery

- Let's assume there is a transaction to modify the City of a student. The following logs are written for this transaction.
- When the transaction is initiated, then it writes 'start' log.
- <Tn, Start>
- When the transaction modifies the City from 'Noida' to 'Bangalore', then another log is written to the file.
- <Tn, City, 'Noida', 'Bangalore' >
- When the transaction is finished, then it writes another log to indicate the end of the transaction.
- <Tn, Commit>

Approaches to modify the database:

- There are two approaches to modify the database:

- 1. Deferred database modification:
  - The deferred modification technique occurs if the transaction does not modify the database until it has committed.
  - In this method, all the logs are created and stored in the stable storage, and the database is updated when a transaction commits.

- 2. Immediate database modification:
  - The Immediate modification technique occurs if database modification occurs while the transaction is still active.
  - In this technique, the database is modified immediately after every operation. It follows an actual database modification.

Shadow paging

- Shadow paging is a technique for providing atomicity and durability in database systems.
- Shadow paging is a copy-on-write technique for avoiding in-place updates of pages. Instead, when a page is to be modified, a shadow page is allocated.

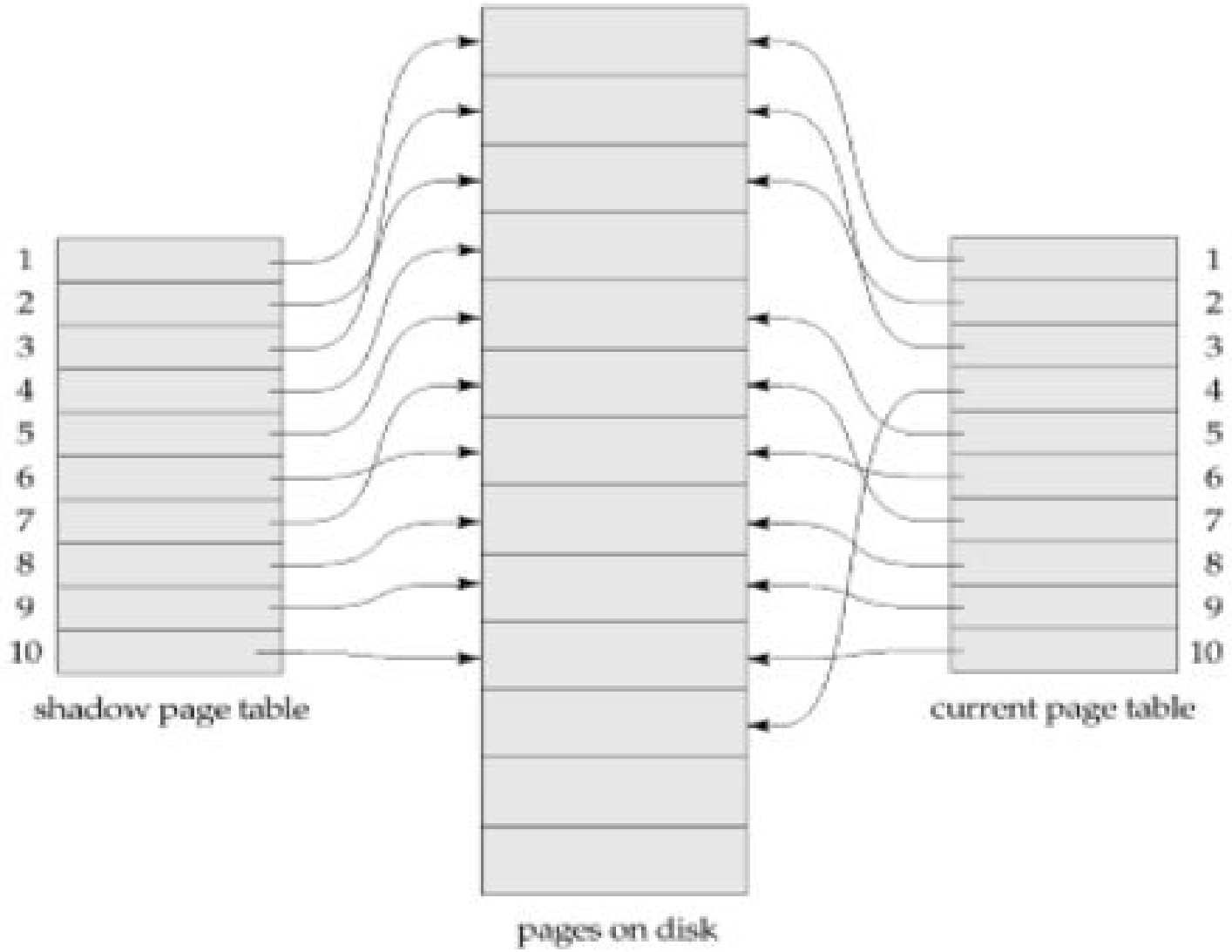
Shadow paging technique:

- The database is partitioned into fixed-length blocks referred to as PAGES.
- Page table has n entries – one for each database page.
- Each contain pointer to a page on disk (1 to 1st page on database and so on…).

Shadow paging

- **The idea is to maintain 2 pages tables during the life of transaction.**
- The current page table
- The shadow page table

- **When transaction starts, both page tables are identical**
- The shadow page table is never changed over the duration of the transaction.
- The current page table may be changed when a transaction performs a write operation.
- All input and output operations use the current page table to locate database pages on disk.


1
2
3
4
5
6
7
8
9
10
shadow page table
pages on disk
1
2
3
4
5
6
7
8
9
10
current page table