Unit5 Concurrency Control: Lock-Based Protocols – Locks, Granting of Locks, The Two-Phase Locking Protocol, Timestamp-Based Protocols – Timestamps, The Timestamp-Ordering Protocol, Thomas Write Rule, Deadlock handling – Deadlock Prevention, Deadlock Detection and Recovery.

Recovery System: Failure Classification, Storage Structure, Recovery and Atomicity, Log-Based Recovery, Shadow Paging Technique.

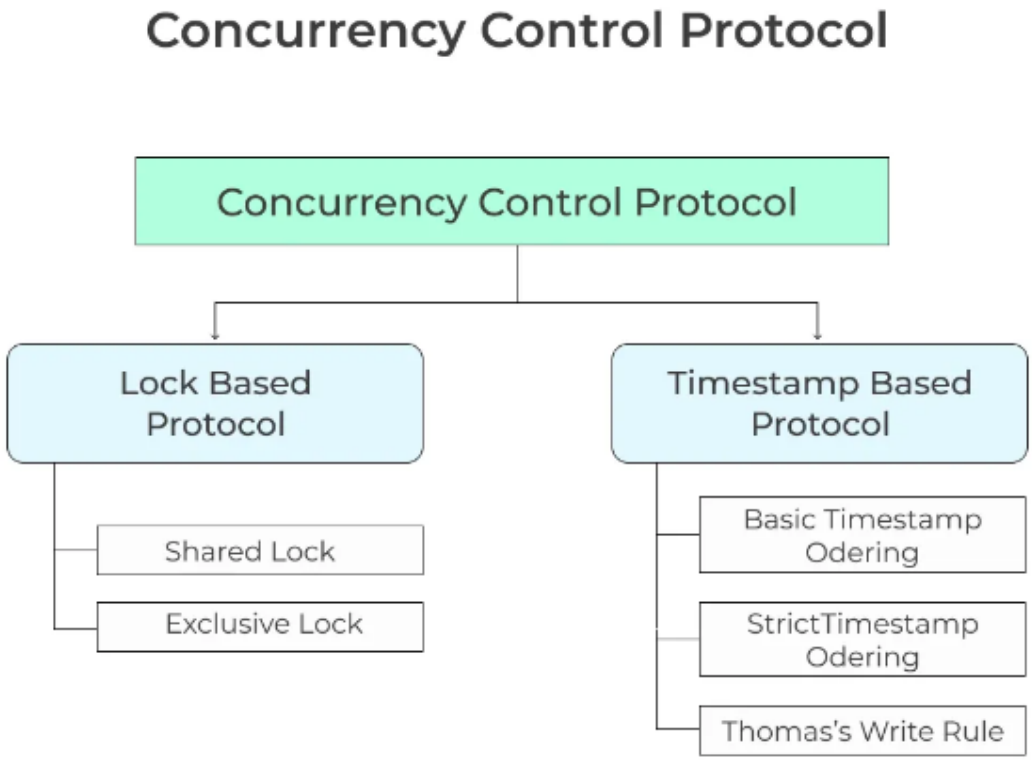
Concurrency Control Protocols

Concurrency control in a DBMS is a technique for managing concurrent transactions and ensuring their atomicity, isolation, consistency, and serializability. Many problems that arise when a large number of transactions are randomly executed at the same time are called concurrency control problems.

- Concurrency Control Protocols in DBMS are procedures that are used for managing multiple simultaneous operations without conflict with each other
- Concurrency control ensures the speed of the transactions but at the same time we should address conflicts occurring in a multi-user system and make sure the database transactions are performed concurrently without violating the Data integrity of the respective databases.

Concurrency control can be broadly divided into two protocols

- Lock Based Protocol
- Timestamp Based Protocol



Lock-Based Protocol

- Lock based protocol mechanism is very crucial in concurrency control which controls concurrent access to a data item.
- It *e*nsures that one transaction should not retrieve and update record while another transaction is performing a write operation on it.

Example

In traffic light signal that indicates stop and go, when one signal is allowed to pass at a time and other signals are locked, in the same way in a database transaction, only one transaction is performed at a time meanwhile other transactions are locked.

- If this locking is not done correctly then it will display inconsistent and incorrect data
- It maintains the order between the conflicting pairs among transactions during execution
- There are two lock modes,
    1. Shared Lock(S)
    2. Exclusive Lock(X)

Shared lock(S)

- Shared locks can only read without performing any changes to it from the database
- Shared Locks are represented by S.
- S – lock is requested using lock – s instruction.

Exclusive Lock(X)

- The data items accessed using this instruction can perform both read and write operations
- Exclusive Locks are represented by X.
- X – lock is requested using lock – X instruction.

Lock Compatibility Matrix

	Shared	Exclusive
Shared	True	False
Exclusive	False	False

- Lock compatibility Matrix controls whether these multiple transactions can acquire locks on the same resource at a time or not
- If a resource is already locked by another transaction, then a new lock request can be granted only if the mode of the requested lock is compatible with the mode of the existing lock.
- There can be any number of transactions for holding shared locks on an item but if any transaction holds exclusive lock, then item no other transaction may hold any Lock on the item.

Timestamp Based Protocol

- It is the most commonly used concurrency protocol.
- Timestamp based protocol helps DBMS to identify transactions and determines the serializability order.

- It has a unique identifier where each transaction is issued with a timestamp when it is entered into the system.
- This protocol uses the system time or a logical counter as a timestamp which starts working as soon as the transaction is created.

Timestamp Ordering Protocol

This protocol ensures serializability among transactions in their conflicting read/write operations

- ***TS(T)***: transaction of timestamp (T)
- ***R–timestamp(X)***: Data item (X) of read timestamp
- ***W–timestamp(X):*** Data item (X) of write timestamp

Timestamp Ordering Algorithms

1. Basic Timestamp ordering

- It ensures transaction execution is not violated by comparing the timestamp of T with Read_TS(X) and Write_TS(X).
- When it finds that transaction execution sequence is violated transaction T is aborted and resubmitted to the system as a new transaction with a new timestamp.

2. Strict Timestamp ordering

It makes sure that the schedules are both strict for easy recoverability and conflict serializability.

3.Thomas's Write Rule

- It does not enforce conflict serializability
- It rejects some write operations, by modifying the checks for the write_item(X) operation as follows.

Read TS(X) > TS (T)

- Abort and rollback transaction T and reject the operation when read timestamp is greater than timestamp transaction.

Write TS(X) > TS(T)

- Whenever write timestamp is greater than the timestamp of the entire transaction then do not execute the write operation but continue processing
- Because sometimes transaction with set and timestamp may be greater than TS(T) and after T in the timestamp has already written the value of X.

If neither 1 nor 2 occurs As mentioned above

- Execute the Write item(X) operation of transaction T and set Write_TS(X) to TS(T).

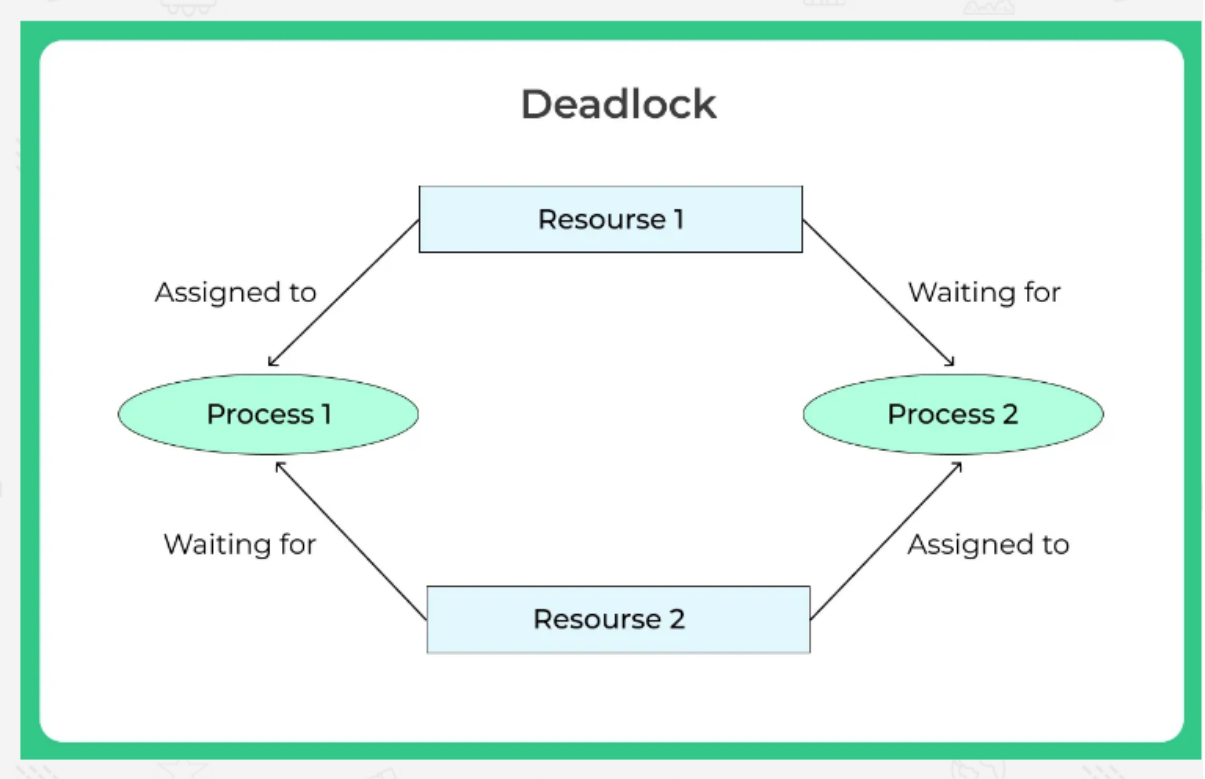
Deadlocks in DBMS

- A deadlock in a database management system (DBMS) is a situation where two or more processes are blocked, waiting for each other to release a resource that they need to proceed with their execution.

Example Deadlock in DBMS

- Every process need some resource for its execution and these resources are granted in sequential order

1. First the process *request some resource*.
2. OS grants the resource to the process if it is available or else it places the request in the wait queue.
3. The process uses it and releases on the completion so that it can be granted to another process
4. Deadlock is a situation where two or more transactions are waiting indefinitely for each other to give up their locks.



Example:

Transaction ***T1 is waiting for transaction T2*** to release the lock and similarly transaction ***T2 was waiting for transaction T1*** to release Its lock, as a result, all activities come to halt state and remain at standstill .This continues until the DBMS detects that ***deadlock has occurred and abort some of the transactions.***

Necessary conditions for Deadlocks

Mutual Exclusion

It implies if **two processes cannot use the same resource at the same time.**

Hold and Wait

A process ***waits for some resources while holding another resource*** at the same time.

No pre-emption

The process which ***once scheduled will be executed till the completion***. No other process can be scheduled by the scheduler meanwhile.

Circular Wait

All the ***processes must be waiting for the resources in a cyclic manner*** so that the last process is waiting for the resource which is being held by the first process.

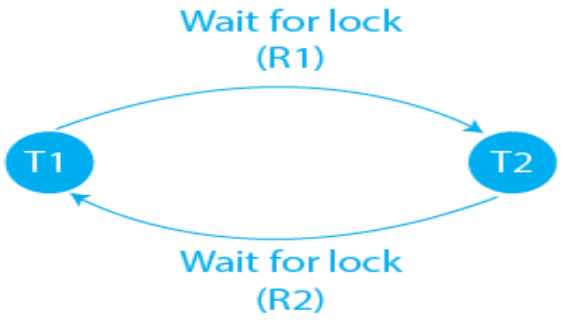
Deadlock avoidance

- Whenever a database is ***stuck in a deadlock*** it's ***better to abort the transactions*** that are ***resulting deadlock*** but this is a waste of resources and time
- Deadlock avoidance mechanism used ***to detect any deadlock situation in advance*** by using some techniques like WEIGHTS FOR GRAPH but this wait for graph method is suitable only for smaller databases, for large databases other prevention techniques are used.
- Deadlock avoidance in DBMS is a technique **used to prevent deadlocks from occurring**. This is achieved by introducing a policy for allocating resources that avoids the conditions for a deadlock.
- There are two main methods for deadlock avoidance in DBMS are **Resource Allocation Graph (RAG) Algorithm** and **Wait-For Graph Algorithms**. Both of these approaches aim to ensure that there are no circular wait conditions in the system, which is the root cause of deadlocks.

Deadlock Detection (Wait for Graph)

Deadlock detection is a technique used in DBMS to identify and resolve deadlocks. In this technique, the system continuously checks for deadlocks and, if found, takes appropriate action to resolve the deadlock.

Wait-for Graph: This method uses a graph-based model to represent the allocation of resources. The graph is used to detect the existence of cycles, which indicate a deadlock has occurred. The wait-for graph is updated whenever a resource is requested or released. For the above example of the Employee and Salary table, the Wait-For Graph can be constructed as shown below:



Deadlock detection is an important technique for ensuring the correct functioning of a DBMS. It allows the system to identify and resolve deadlocks in a timely manner, reducing the risk of system failure. However, it can also lead to decreased system performance as a result of the additional overhead required for monitoring and controlling resource allocation.

Deadlock Prevention in DBMS

- The deadlock prevention strategy is appropriate for huge databases. A stalemate can be avoided by allocating resources in such a way that a deadlock never arises. The DBMS examines the operations to see if they can cause a deadlock. If they do, the transaction is never permitted to be executed.
- Deadlock prevention in DBMS is a technique used to eliminate the possibility of deadlocks occurring in a database management system. It involves making certain changes to the system to reduce the chances of resource contention, which is the main cause of deadlocks.
- Prevention techniques are ***used for larger databases.***
- Resource ***allocation is done in such a way that deadlock never occurs***
- DBMS ***analyse the transactions to determine whether any deadlock situation can arise or not***, if there is any possibility then DBMS never allows such transaction for execution.

There are two main schemes followed for Deadlock Prevention in DBMS namely, Wait Die and Wound Wait Schemes. These are explained below in detail:

Wait-Die scheme

- Whenever there is a conflict of resource between transactions DBMS simply ***check the timestamp of both the transactions***
- There are two transactions Ti and Tj and let TS(T) is a timestamp of any transaction T
- If T2 holds a lock by some other transaction and T1 is requesting for resources held by T2 then the following actions are performed by DBMS

Check if TS(Ti) < TS(Tj) :

- If Ti is older transaction and Tj has some resource held then ***Ti is allowed to wait until data item is available for execution***
- Simply if ***older transaction is waiting for a resource*** which is ***locked by younger transaction*** then ***older transaction is allowed to wait*** for resource until it is available

Check if TS(Ti) < TS(Tj)

- If Ti is older transaction and has held some resource and if Tj is waiting for it, then ***Tj is killed and restarted later with the random delay but with the same timestamp.***

Wound wait scheme

- Whenever an older transaction request for a resource which is held by a younger transaction, ***older transaction forces the younger transaction to kill its transaction and release the resource to the elder one***
- After minute delay ***younger transaction is restarted within the same timestamp***
- Whenever older transaction has held a resource which is requested by the Younger transaction, then the ***younger transaction is asked to wait until older releases it.***

Recovery System

Failure Classification

Transaction failure: – 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., deadlock)

System crash: a power failure or other hardware or software failure causes the system to crash. It is assumed that non-volatile storage contents are not corrupted.

Disk failure: a head crash or similar failure destroys all or part of disk storage.

Storage Structure

Volatile storage: –

does not survive system crashes –

examples: main memory, cache memory

Non-volatile storage: – survives system crashes – examples: disk, tape

Stable storage: – a mythical form of storage that survives all failures – approximated by maintaining multiple copies on distinct non-volatile media.

Stable-Storage Implementation

- Maintain multiple copies of each block on separate disks; copies can be at remote sites to protect against disasters such as fire or flooding.
- Failure during data transfer can result in inconsistent copies
- Protecting storage media from failure during data transfer (one solution): –

Execute output operation as follows (assuming two copies of each block):

1. Write the information onto the first physical block.
2. When the first write successfully completes, write the same information onto the second physical block.
3. The output is completed only after the second write successfully completes.

Protecting storage media from failure during data transfer

Copies of a block may differ due to failure during output operation. To recover from failure:

1. First find inconsistent blocks:

(a) Expensive solution: Compare the two copies of every disk block.

(b) Better solution: Record in-progress disk writes on non-volatile storage. Use this information during recovery to find blocks that may be inconsistent, and only compare copies of these.

2. If either copy of an inconsistent block is detected to have an error (bad checksum), overwrite it by the other copy. If both have no error, but are different, overwrite the second block by the first block.

DATA ACCESS

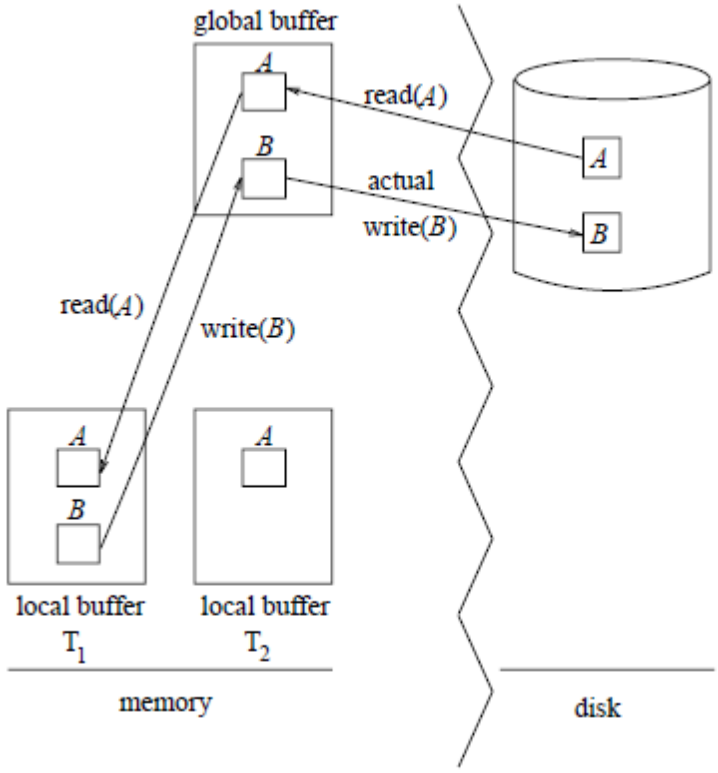
- Physical blocks are those blocks residing on the disk. Buffer blocks are the blocks residing temporarily in main memory.

- Block movements between disk and main memory are initiated through the following two operations: –
  -input(B) transfers the physical block B to main memory.
  – output(B) transfers the buffer block B to the disk, and replaces the appropriate physical block there.
- Each transaction Ti has its private work-area in which local copies of all data items accessed and updated by it are kept. Ti 's local copy of a data item X is called xi.
- We assume, for simplicity, that each data item fits in, and is stored inside, a single block.
- Transaction transfers data items between system buffer blocks and its private work-area using the following operations :
  – read(X) assigns the value of data item X to the local variable xi.
  – write(X) assigns the value of local variable xi to data item X in the buffer block.
  – both these commands may necessitate the issue of an input(BX) instruction before the assignment, if the block BX in which X resides is not already in memory.

Transactions perform **read(X)** while accessing X for the first time; all subsequent accesses are to the local copy. After last access, transaction executes **write(X).**

• output(B_X) need not immediately follow write(X). System can perform the output operation when it deems fit.

Example of Data Access



Recovery and Atomicity

- Consider transaction Ti that transfers $50 from account A to account B; goal is either to perform all database modifications made by Ti or none at all.

- Several output operations may be required for Ti (to output A and B). A failure may occur after one of these modifications have been made but before all of them are made.
- To ensure atomicity despite failures, we first output information describing the modifications to stable storage without modifying the database itself.

We study two approaches:

log-based recovery, and **shadow-paging**

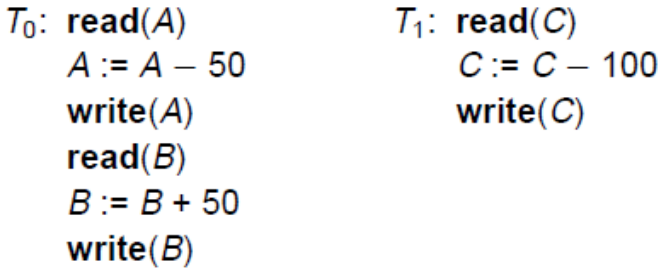
We assume (initially) that transactions run serially, that is, one after the other.

Log Based Recovery

- A log is kept on stable storage. The log is a sequence of log records, and maintains a record of update activities on the database.
- When transaction Ti starts, it registers itself by writing a < Ti **start** > log record
- Before Ti executes **write** (X), a log record < Ti, X, V1, V2 > is written, where V1 is the value of X before the write, and V2 is the value to be written to X.
- When Ti finishes it last statement, the log record < Ti **commit** > is written.
- We assume for now that log records are written directly to stable storage (that is, they are not buffered).

Deferred Database Modification

- This scheme ensures atomicity despite failures by recording all modifications to log, but deferring all the **write**s to after partial commit.
- Assume that transactions execute serially.
- Transaction starts by writing < Ti **start** > record to log.
- A **write**(X) operation results in a log record < Ti, 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 Ti partially commits, < Ti **commit** > is written to the log
- Finally, log records are used to actually execute the previously deferred writes.
- During recovery after a crash, a transaction needs to be redone if and only if both < Ti **start** > and < Ti **commit** > are there in the log.
- Redoing a transaction Ti (**redo**(Ti)) sets the value of all data items updated by the transaction to the new values.
- Crashes can occur while the transaction is executing the original updates, or while recovery action is being taken.
- example transactions T0 and T1 (T0 executes before T1):



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

(a)	(b)	(c)
<T_0 **start**>	<T_0 **start**>	<T_0 **start**>
<T_0, A, 950>	<T_0, A, 950>	<T_0, A, 950>
<T_0, B, 2050>	<T_0, B, 2050>	<T_0, B, 2050>
	<T_0 **commit**>	<T_0 **commit**>
	<T_1 **start**>	<T_1 **start**>
	<T_1, C, 600>	<T_1, C, 600>
		<T_1 **commit**>

If log on stable storage at time of crash is as in case:
(a) No redo actions need to be taken
(b) **redo**(T0) must be performed since <T0 **commit**> is present
(c) **redo**(T0) must be performed followed by **redo**(T1) since
<T0 **commit**> and <T1 **commit**> are present

Immediate Database Modification

- This scheme allows database updates of an uncommitted transaction to be made as the writes are issued; since undoing may be needed, update logs must have both old value and new value
- Update log record must be written before database item is Written
- Output of updated blocks can take place at any time before or after transaction commit
- Order in which blocks are output can be different from the order in which they are written.

Immediate Database Modification Example

Log	Write	Output
<T_0 **start**>		
<T_0, A, 1000, 950>		
<T_0, B, 2000, 2050>		
	A = 950	
	B = 2050	
<T_0 **commit**>		
<T_1 **start**>		
<T_1, C, 700, 600>		
	C = 600	
		B_B, B_C
<T_1 **commit**>		
		B_A

- Note: B_X denotes block containing X.

Recovery procedure has two operations instead of one :

- **undo**(Ti) restores the value of all data items updated by Ti to their old values, going backwards from the last log record for Ti.
- **redo**(Ti) sets the value of all data items updated by Ti to the new values, going forward from the first log record for Ti

When recovering after failure:

- Transaction Ti needs to be undone if the log contains the record < Ti **start**>, but does not contain the record <Ti **commit**>.
- Transaction Ti needs to be redone if the log contains both the record < Ti **start**> and the record <Ti **commit**>.

Undo operations are performed first, then redo operations.

Immediate DB Modification Recovery Example

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

(a)	(b)	(c)
$<T_0$ **start**$>$	$<T_0$ **start**$>$	$<T_0$ **start**$>$
$<T_0, A, 1000, 950>$	$<T_0, A, 1000, 950>$	$<T_0, A, 1000, 950>$
$<T_0, B, 2000, 2050>$	$<T_0, B, 2000, 2050>$	$<T_0, B, 2000, 2050>$
	$<T_0$ **commit**$>$	$<T_0$ **commit**$>$
	$<T_1$ **start**$>$	$<T_1$ **start**$>$
	$<T_1, C, 700, 600>$	$<T_1, C, 700, 600>$
		$<T_1$ **commit**$>$

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 then A and B are set to 950 and 2050 respectively.

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

Checkpoints

Problems in recovery procedure as discussed earlier:

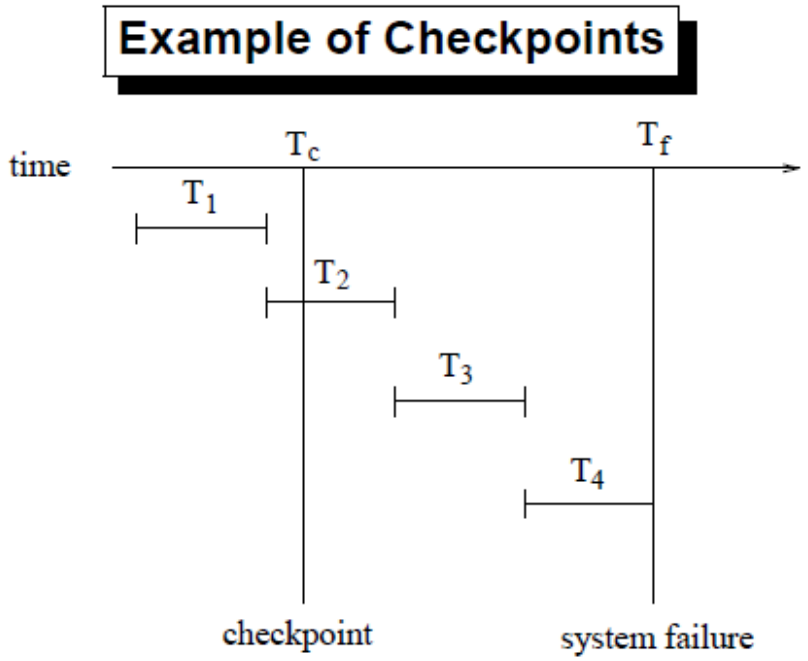
1. searching the entire log is time-consuming
2. we might unnecessarily redo transactions which have already output their updates to the database.

Streamline recovery procedure by periodically performing checkpointing

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**> onto stable storage.

- During recovery we need to consider only the most recent transaction Ti that started before the checkpoint, and transactions that started after Ti .
- Scan backwards from end of log to find the most recent <**checkpoint**> record
- Continue scanning backwards till a record <Ti **start**> is found.

- Need only consider the part of log following above **start** record. Earlier part of log can be ignored during recovery, and can be erased whenever desired.
- For all transactions (starting from Ti or later) with non <Ti **commit**>, execute **undo**(Ti). (Done only in case of immediate modification.)
- Scanning forward in the log, for all transactions starting from Ti or later with a <Ti **commit**>, execute **redo**(Ti).



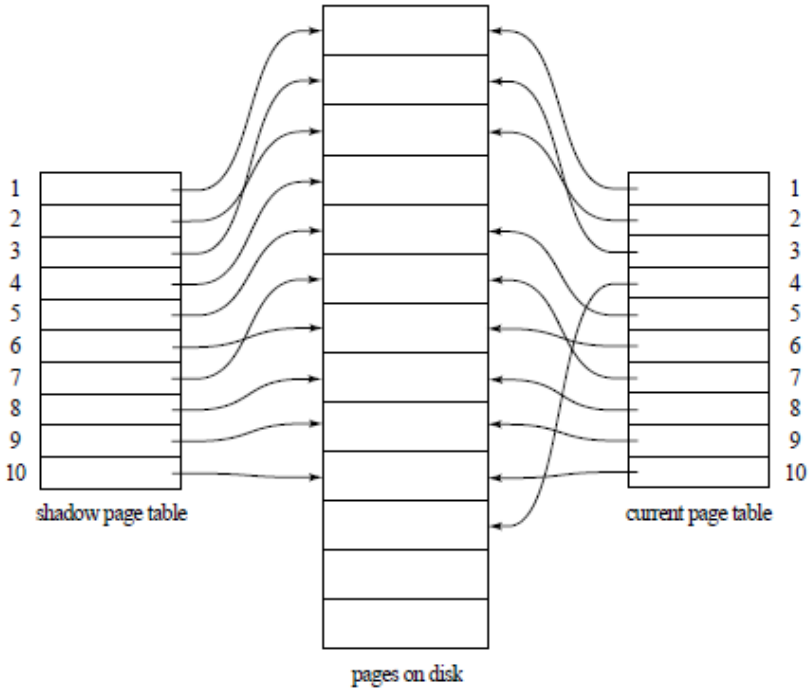
- T_1 can be ignored (updates already output to disk due to checkpoint)
- T_2 and T_3 redone
- T_4 undone

Shadow Paging

Alternative to log-based recovery

- Idea: 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 non-volatile storage, such that state of the database prior to transaction execution may be recovered. Shadow page table is never modified during execution
- To start with, both the page tables are identical. 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, and the update is
- performed on the copy.

Example of Shadow Paging



Shadow and current page tables after write to page 4

To commit a transaction:

1. Flush all modified pages in main memory to disk
2. Output current page table to disk
3. Make the current page the new shadow page table

– keep a pointer to the shadow page table at a fixed (known) location on disk.

– 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.
- No recovery is needed after a crash — new transactions can start right away, using the shadow page table.
- Pages not pointed to from current/shadow page table should be freed (garbage collected).

Advantages of shadow-paging over log-based schemes –

- No overhead of writing log records; recovery is trivial

Disadvantages:

- – Commit overhead is high (many pages need to be flushed)
- – Data gets fragmented (related pages get separated)
- – After every transaction completion, the database pages containing old versions of modified data need to be garbage collected and put into the list of unused pages
- – Hard to extend algorithm to allow transactions to run concurrently.