Unit 4

Concurrency Control

* Concurrency control: a procedure to manage multiple transactions without conflicting with each other

→ helps enforce isolation through mutual exclusion

→ helps w/ consistency preservation of transactions

→ resolve read - write and write - write conflict issues

→ enforces serializability

* Types of concurrency control protocols

(i) locking-based protocols

(ii) timestamp - based protocols

(iii) validation - based protocols

* Locks

→ an operation which secures permission to Read or Write a data item for transaction

→ each lock is associated w/ a data item

LOCK(X): Data item X locked by requesting transaction

UNLOCK(X): Data item X is made available to all other transactions

* Types of Locks

(i) Binary Locks :

→ has only 2 states, locked or unlocked
(ie 1 and 0)

Lock (X) = 1 ⇒ cannot be accessed

Lock (X) = 0 ⇒ can be accessed

Rules to follow (Enforced by lock manager)

1. Must issue Lock(X) before read / write operations
2. must issue unlock(X) after all read / write operations are complete
3. cannot issue a lock / unlock ~~if it~~ unless it already holds the lock on X.

Algorithm

LOCK

B: if Lock(X) = 0
then Lock(X) ← 1
else
begin
wait until lock(X) = 0
go to B
end

UNLOCK

Lock (X) ← 0
if any transactions are waiting, wake them up

Disadvantage

→ at most one transaction can hold a lock on a given item
→ not practically useful

(ii) Shared / Exclusive locks

Shared locks - (Read) when a read operation is to be done

Exclusive locks - (Read (write) when a read/write operation is to be done.

→ With read locks, other transactions are also allowed to read the item

→ With write locks, a single transaction exclusively holds the lock on the item

request →

grant ↓	S	X
S	Yes	No
X	No	No

Rules

1. Issues the read-lock(X) or write-lock(X) before read/write operations.
2. Issue the unlock(X) after all read & write operations are completed.
3. cannot issue a read-lock(X) if it already holds a read/write lock (exception - downgrade from write to read)
4. cannot issue a write-lock(X) if it already holds a read lock a write lock (exception - upgrade lock from read to write)

* Lock Manager

→ stores the identity of a transaction locking a data item, lock mode and the pointer to the next data item

→ implemented using a linked list

Pseudocode

```
read_lock(X):
B:  if   Lock(X) = unlocked
        then   begin
                    lock(X)  ← read_locked
                    no_of_reads(X) ← 1
               end
    else  if  Lock(X) = read_locked
        then
              no_of_reads (X)  ← no_of_reads(X)+1
    else
          begin
               wait until lock(X) = 'unlocked'
               wake up transaction
               go to B
          end
```

write_lock(x):

Pseudocode:

```
B: if LOCK(x) = unlocked
       then LOCK(x) ← write-locked

   else
       begin
       wait until LOCK(x) = unlocked
       go to B
   end;
```

```
UNLOCK(x)

   if LOCK(x) = write-locked
   then
       begin
       wakeup one of the waiting transactions
       end
else if LOCK(x) = read-locked
then
       begin
          no_of_reads(x) ← no_of_reads(x) - 1
          if no_of_reads(x) = 0
             then
                begin
                   LOCK(x) = unlocked
                   wake up transaction
                end
       end
```

* Conversion of locks

→ A transaction that already holds a lock on an item X is allowed under certain conditions to convert the lock from one locked state to another.

→ Upgrading a lock: can upgrade if T is the only transaction holding a read lock, otherwise must wait

* 2PL

→ all locking operations precede the first unlock operation

→ ensures serializability

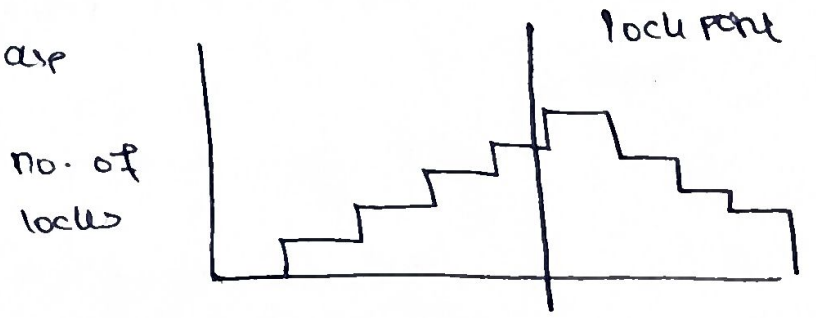
2 phases

Expanding or growing phase - new locks can be acquired, none can be released.

Shrinking - existing locks can be released, no new locks can be acquired.

upgradation can happen during the expanding phase

downgrading during shrinking phase



Enforcing Serializability

X must remain locked by T until all r/w ops are done

Another transaction seeking to access X may be forced to wait

Also if an item Y is locked earlier than it is needed, another transaction seeking access to Y is forced to wait, even though T is not using Y yet

∴ 2PL suffers from deadlock & cascading rollback

→ Types of 2PL

1. **Basic 2PL** – Transaction locks data items incrementally; results in deadlock

2. **Strict 2PL**

→ does not release any write locks until after a commit

→ not deadlock free, but no cascading rollback

→ possible to enforce recoverability

3 **Rigorous 2PL**

→ does not release any of its locks until after commit/abort

→ more restrictive than strict 2PL, but easier to implement

→ no cascading rollback or deadlock

4. **Conservative 2PL**

→ requires a transaction to lock all the items it access before the transaction begins execution, by predeclaring its read & write set

→ Transaction is in shrinking phase after it starts

→ is deadlock free

→ is difficult to use because of difficulty in predeclaring the read and write set.

Recovery Subsystems

* **Recovery Process** : restore database to the most recent consistent state

* **Recovery Strategies**:

  (i) restore backed-up copy of database from archival storage

  (ii) identify any changes that may cause an inconsistency and apply **undo** or **redo**

* Recovery is done w/ log files (refer structure in transactions notes)

The old value of an item = Before Image = BFIM → for UNDO

new value = After Image = AFIM → for REDO

* **Caching**

→ db items being read & written in disk blocks

→ buffer has modified bit, pin/unpit bit

↓

dirty bit, 0 if buffer has been modified

↓

set to 1 if it cannot be written back into the disk yet (usually waiting for a commit)

* Strategies for Flushing the Buffer

(i) In-place updating

→ write buffer back to og location

→ overwrites old value

→ recovery requires a log

(ii) Shadowing

→ write updated buffer to a diff. location

→ old value = BFIM

new value = AFIM

* Write-Ahead Log (WAL)

(i) undo-type log entries - include old values - BFIM

(ii) redo-type log entries - include the new values - AFIM

→ When a data item is updated, entries are made in the appropriate log entry

→ With WAL, info needed for recovery must be first written onto the log file on disk, before changes are made to the database on disk.

WAL Rules

(i) BFIM cannot be overwritten by AFIM, until all UNDO type log records have been force-written onto the disk

(ii) Commit of a transaction cannot be completed until all the REDO & UNDO log records for that transaction have been force-written to disk (STABLE STORE)

* Techniques used to move a page from cache to memory

(i) no-steal approach: can't be written to disk before commit

(ii) steal approach: can be written to disk before commit

(iii) force approach: all pages updated by a transaction are immediately written to disk

(iv) no-force approach: all pages updated after commit

* Check pointing?

checkpoint - a type of entry in the log

Due to system failure, and when the system restarts

If the order of operations is

(i) Transaction → commit → Checkpoint : no action to be taken

(ii) Transaction → Checkpoint → commit : redo

(iii) Transaction → Checkpoint
or : undo
Checkpoint → Transaction

see also on Pg. 16

* Recovery Policies for Non-Catastrophic Transaction

Failures.

(1) Deferred Update

→ do not physically update until commit

→ till commit, updates are recorded in the log file

→ if failure, no changes in db so no UNDO

→ after commit - REDO operations → change AFIM value

called the NO-UNDO/REDO algorithm

For concurrent users:

→ maintain 2 lists → commit list → REDO

→ active list → submit again, since they were effectively

(2) Immediate Update

→ updated for every transaction, even before commit

→ in case of failure - rollback

maintain 2 lists → committed transactions → REDO

→ active transactions → UNDO - set to BFIM

do in reverse order of log entry

2PL and Recovery Mechanism Problems

(1) Consider the 2 transactions T_1 and T_2

T_1 : $r_1(y)$; $r_1(x)$; $w_1(x)$,

T_2 : $r_2(x)$; $w_2(x)$;

Write a serializable schedule that follows basic 2PL.

T_1	T_2
S_LOCK(Y)	
$r_1(y)$	
	X_LOCK(X)
	$r_2(x)$
	$w_2(x)$
	UNLOCK(X)
X_LOCK(X)	
$r_1(x)$	
$w_1(x)$	
UNLOCK(X)	
UNLOCK(Y)	

(2) Implement strict 2PL on the following transactions

T_1 : $r_1(y)$; $r_1(x)$; $w_1(x)$; commit

T_3 : $r_3(x)$; $r_3(y)$; $w_3(x)$; commit

T_1	T_2
S_LOCK(Y)	
$r_1(y)$	
	XLOCK(X)
	$r_3(x)$
	SLOCK(Y)
	$r_3(y)$
	$w_3(x)$
	commit
	unlock(x)
	unlock(y)

Xlock(x)
Unlock(x)
$r_1(x)$
$w_1(x)$
commit
Unlock(x)
~~Unlock(y)~~

(3) Apply basic 2PL and strict 2PL to the following schedule

T_3	T_4
$r_1(x)$	$r_2(x)$
$w_1(x)$	$w_2(x)$
$r_1(y)$	
$w_1(y)$	

Basic 2PL

T_3	T_4
XLOCK(X)	
$r_1(x)$	
$w_1(x)$	
XLOCK(Y)	
UNLOCK(X)	
	XLOCK(X)
	$r_2(x)$
$r_1(y)$	$w_2(x)$
$w_1(y)$	
UNLOCK(Y)	

Strict 2PL

T_3	T_4
XLOCK(X)	
$r_1(x)$	
$w_1(x)$	
XLOCK(Y)	
	XLOCK(X)
	wait
$r_3(y)$	
$w_3(y)$	
commit	
	XLOCK(X)
	$r_2(x)$
	$w_2(x)$
	commit

(4) Consider a schedule of 4 transactions T_1, T_2, T_3, T_4 and its corresponding log entry at the point of a system crash.

[start - transaction, T_1]

Initial values : A = 30, B = 10, C = 30, D = 20

[read - item, T_1, A]

[read - item, T_1, D]

[write - item, T_1, D, 20, 25]

[commit, T1]

checkpoint

[start - transaction, T_2]

[read - item, T_2, B]

[start transaction T_4]

[read - item, T_4, D]

[write - item, T_4, D, 25, 15]

[write - item, T_2, B, 12, 18]

[commit, T_2]

[start - transaction, T_3]

[write - item, T_3, C, 30, 40]

[read - item, T_4, A]

[write - item, T_4, A, 30, 20]

[commit, T_4]

[read - item, T_3, D]

[write - item, T_3, D, 15, 25] → System Crash

A. Use the deferred update protocol & apply the necessary redo operations

Solution : REDO type entries contain only AFIM values in the write-item operations :

[start_transaction, T_1]

[write_item, T_1, D, 25]

[commit, T_1]

[checkpoint]

[start_transaction, T_2]

[start_transaction, T_4]

[write_item, T_4, D, 15]

[write_item, T_2, B, 18]

[commit, T_2]

[start_transaction, T_3]

[write_item, T_3, C, 40]

[write_item, T_4, A, 20]

[commit, T_4]

[write_item, T_3, D, 25]

For Redo operations, for everything after the checkpoint, maintain 2 lists

REDO RIST

T_4, T_2

rewrite items in the same order.

ACTIVE LIST

T_3 → push again

↳ is uncommitted

[write_item, T_4, D, 15]

[write_item, T_2, B, 18]

[write_item, T_4, A, 20]

B. Remove the [commit, T_2] entry. Use the immediate update policy to make the necessary undo/redo operations

commit list

T_4

redo T_4 operations in the same order

[write_item, T_4, D, 15] (25)

[write_item, T_4, A, 30, 20]

active list

T_2, T_3

undo operations in the backward order

[write_item, T_3, D, 15, 25]

[write_item, T_3, C, 30, 40]

[write_item, T_2, B, 12, 18]

Transaction Processing

* **Transaction**: provides a mechanism for describing the logical units of a database processing

* **Classification of database systems**

(i) single-user system: at most one user at a time

(ii) multi-user system: many users can access the system concurrently

* **Interleaved and Parallel Processing**

A. **Interleaved Processing**

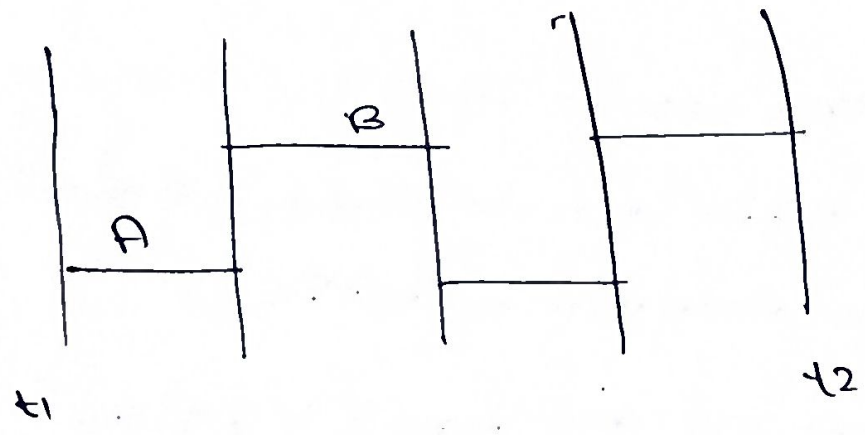
→ concurrent execution of processes is interleaved in a single CPU

→ keeps CPU busy when reading I/O operations, ie minimum idle time during I/O

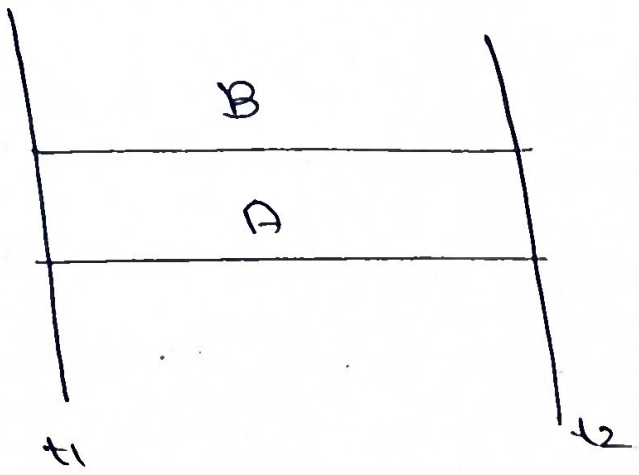
→ prevents long processes from delaying other process

B. **Parallel Processing**

→ processes are concurrently executed in multiple CPUs



Interleaved Processing



Parallel Processing

* Granularity

→ granularity is the size of the data item

→ granularity of data = a field, record or a whole block

→ Transaction processing concepts are independent of granularity and apply to data items in general

* Read and Write Operations on Database

(1) Read_item (X): reads a database item named X into a program variable X

Steps:

→ find address of disk block that contains memory item X

→ copy that disk block into a buffer in main memory

→ copy item X from buffer to program variable X

(2) Write_item (X): writes the value of program variable X into the database item named X.

Steps

→ find address of disk block that contains item X

→ copy that disk block into a buffer in main memory (if that disk block is not already in some main memory buffer)

→ copy item X from the program variable named X into the correct location on the buffer

→ store the updated block from buffer back to disk

* Transaction Notation

Ex1 T_1

```
read_item(X);
X := X-N
write_item(X);
read_item(Y);
Y: Y+N;
write_item(Y);
```

Ex2 T_2

```
read_item(X);
X := X+M;
write(X);
  item
```

T_1: $b_1; r_1(X); w_1(X); r_1(Y); w_1(Y); e_1$

T_2: $b_2; r_2(X); w_2(X); e_2;$

* Concurrency control & database recovery mechanism → deal with the database commands in a transaction.

→ uncontrolled concurrent execution would lead to database Inconsistency

* Sources of Database Inconsistency

→ uncontrolled execution of database transactions in a multi-user environment can lead to db inconsistency.

Source of db inconsistency

(i) Lost Update Problem

(ii) Dirty Read Problem

(iii) Incorrect summary problem

(iv) Unrepeatable Read problem

① Lost Update Problem

→ This occurs when two transactions that access the same database items have their operations interleaved in a way that makes the value of the same database item incorrect.

T_1	T_2
$r_1(X)$	
$X := X - N$	
	$r_2(X)$
	$X := X + M$
$w_1(X)$	
$R_1(Y)$	
	$w_2(X)$ ← X has an incorrect value.
$Y := Y + N$	
$w_1(Y)$	

Suppose $X = 100$

$N = 5$

and $M = 4$

→ The final value of X should be 99

However T_2 reads the value of X before T_1 changes it in the database ⇒ The value of X would be 104 instead, because the update in T_1 that removed 5 was lost.

② The Temporary Update / Dirty Read Problem

→ This occurs when one transaction updates a database item and then the transaction fails for some reason.

→ The updated item is accessed by another transaction before it is changed back to its original value.

T_1	T_2
$r_1(X)$	
$X := X - N$	
$w_1(X)$	
	$r_2(X)$
	$X := X + M$
	$w_2(X)$
$r_1(Y)$	
T_1 fails	

Transaction T_1 fails and must change the value of X back to its old value, meanwhile T_2 has read the temporary incorrect value of X.

(3) Incorrect Summary Problem

→ If one transaction is calculating an aggregate summary function on a number of records, while other transactions are updating some of these records, the aggregate fn. may calculate some values before they are updated and others after they are updated.

T_1	T_3
	$sum := 0$
	$r_3(A)$
	$sum := sum + A$
$r_1(X)$	
$X := X - N$	
$w_1(X)$	
	$r_3(X)$
	$sum := sum + X$
	$r_3(Y)$
	$sum := sum + Y$
$r_1(Y)$	
$Y := Y + N$	
$w_1(Y)$	

T_3 reads after N is subtracted and reads before N is added to Y.

⇒ The sum is off by N.

(4) Unrepeatable Read Problem

→ When a transaction reads the same item twice and the item is changed by another transaction T between the 2 reads.

→ T receives a different for 2 reads of the same item

T_1	T_2
$R_1(X)$	
	$R_2(X)$
$X := X + 50$	
$W_1(X)$	
	$R_2(X)$

if $X = 100$

→ The initial $R_2(X)$ reads 100, but the second $R_2(X)$ reads 150.

* Why is recovery needed?

→ If the transaction fails after executing some of the operations but before executing all of them, the operations already executed must be undone and must have no lasting effect.

* Types of Failures

classified as transaction, system and media failures

(i) Computer failure (system crash): hardware / software / network error during transaction

(ii) Transaction or system error : operations that cause overflow, division by 0, logical programming errors etc.

(iii) Local errors or exception conditions detected by the transaction.

cancellation of transaction due to lack of data (insufficient fund)

(iv) concurrency control enforcement : transactions aborted due to deadlock, timeout, serializability violations

(v) disk failure : read / write malfunction

(vi) physical problems / catastrophe : power failure, theft etc

DBMS has a Recovery Subsystem to protect db against system failures

* Transaction States

→ A transaction is an atomic unit of work that is either completed in its entirety or not done at all.

→ The different transaction states are:

(i) Active State : Transaction goes into active state immediately after it starts execution

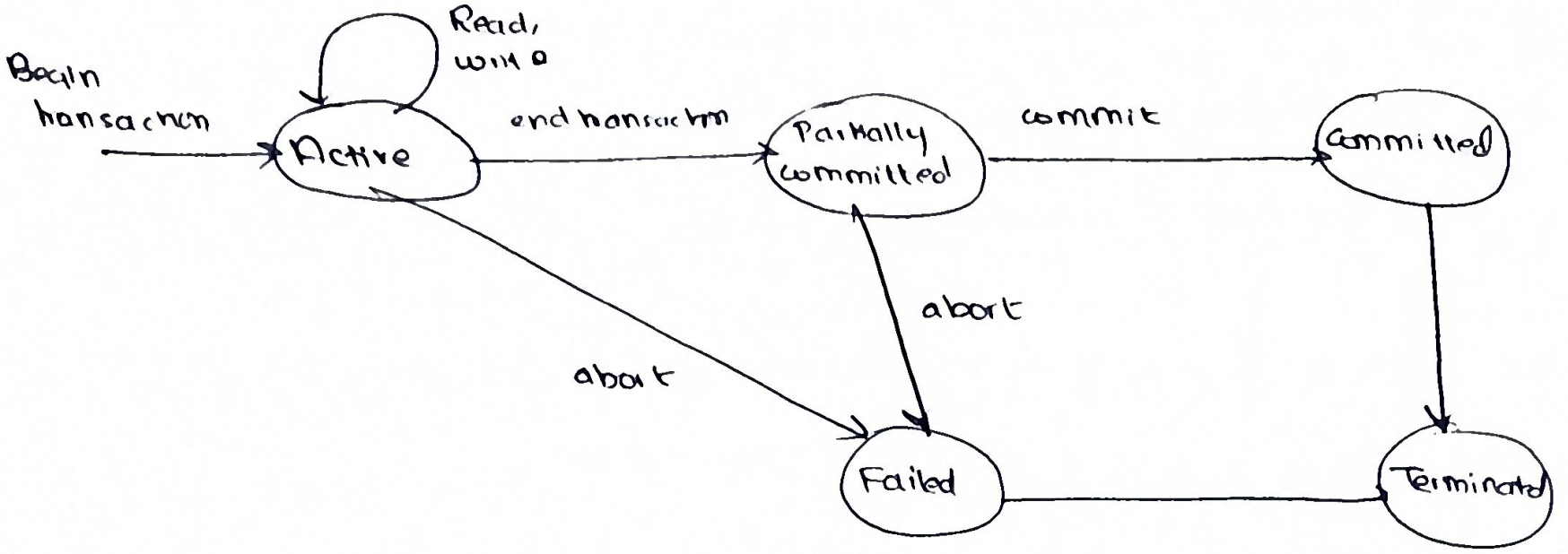
(ii) Partially Committed State : when the transaction ends, it moves to the partially committed state

(iii) Committed State : If the transaction reaches its commit point, the transaction enters the committed state

(iv) Failed State : If the transaction is aborted during its active state, it enters into the failed state.

(v) Terminated State : corresponds to the transaction leaving the system

* State Transition Diagram



The recovery manager keeps track of the following operations

(i) Begin_transaction : marks beginning of transaction execution

(ii) Read / write : the specified read / write operations on the db items that are executed as part of a transaction

(iii) end_transaction : marks that read & write ops have completed,

(iv) commit_transaction : signals a successful end of transaction, any changes can be safely committed to the db.

(v) rollback / abort : transaction ended unsuccessfully, any changes or effects must be undone.

* System Log

→ To be able to recover from transaction failure, the system maintains a log.

→ The log keeps track of all transaction operations that affect the value of db items, as well as info needed to permit recovery

→ The log is a sequential, append-only file

→ The log is kept on a disk, so it is not affected by any type of failure, also periodically backed up to archival storage.

* Contents of Log File

A log file has the following contents:

T refers to a unique transaction ID

(i) [start_transaction, T] : records that transaction T has started execution

(ii) [write_item, T, X, old_value, new_value] : records that T has changed the value of database item X from old_value to new_value

(iii) [read_item, T, X] : records that transaction T has read the value of database item X

(iv) [commit, T] : records that transaction T has completed successfully, and affirms that its effect can be committed to the db

(v) [abort, T] : records that transaction T has been aborted.

* UNDO & REDO Operations

UNDO: reset all items changed by a write operation of T to their old-values by tracing backwards through the log records

REDO: by setting all items changed by a write operation of T to their new-values. Redo may be necessary when a transaction

has been written into the log, but a failure occurs before the new-values have been written to the actual database.

* Writing of log Files

→ The log file must be kept on disk.

→ It is common to keep one or more blocks of the log file in the main memory buffers, until they are filled with log entries, and then to write them back to the disk only once, rather than writing to the disk every time a log entry is added.

→ At the time of system crash, only the log entries that have been written back to the disk are considered in the recovery process

Force Writing a log : Before a transaction reaches its commit point, any portion of the log that has not been written to the disk yet must now be written to the disk. This is called force-writing the log file before committing a transaction.

* ACID Properties

→ A transaction should possess several properties called ACID properties that should be enforced by the concurrency control recovery methods

(1) Atomicity

→ a transaction is an atomic unit of processing

→ it is either performed in its entirety, or not performed at all

→ responsibility of the transaction recovery subsystem.

- (ii) Consistency Preservation

→ every transaction should execute w/o the interference of other transactions

→ transaction must take db from one consistent state to another

→ programmer's responsibility or the DBMS module to enforce integrity constraints

(iii) Isolation

→ transaction should not make its update visible to other transactions until it is committed

→ solves the temporary update problem & eliminates cascading rollbacks

Level 0 : no dirty read
Level 1 : no lost updates
Level 2 : no lost updates & no dirty read
Level 3 : level 2 + repeatable reads

→ enforced by concurrency control subsystem of DBMS

(iv) Durability or permanency

→ once a transaction changes the db and the changes are committed these changes must never be lost because of subsequent failure

→ responsibility of the recovery subsystem

Recovery protocols enforce atomicity & durability

Example Transfer ₹50 from account a to b

read (a) : a := a - 50 write(a)

read (b) : b := b + 50 ; write (b)

Atomicity: a must not be debited w/o crediting b

Consistency: Sum of a & b remains constant

Isolation: if another transaction reads a and b, it must see one of the 2 states: before or after the transaction, otherwise a+b is wrong

Durability: if b is notified of credit, it must persist even if the database crashes.

Schedules

* Schedule = history : ordering of operations of transactions

* Conflicting Operations in a Schedule:

  (i) belong to different transactions

  (ii) access same data item X

  (iii) at least one operation is a write_item(X)

  : can be read-write & write-write

eg. Find conflicting operations

S: $r_1(X), r_2(X), w_1(X), r_1(Y), w_2(Y), w_1(Y)$;

ans $r_1(X), w_2(X)$

$r_2(X), w_1(X)$

$w_1(X), w_2(X)$

complete schedulle

operations in S are those in $T_1, T_2, T_3 \ldots$

relative ordering is the same

for conflicting operations - one must happen before the other.

conflicting operations can happen because of a change in order resulting in a different outcome

(i) read-write conflict $r_1(X); w_2(X) \; != \; w_2(X); r_1(X)$

(ii) write-write conflict $w_1(Y); w_2(Y) \; != \; w_2(Y); w_1(Y)$

* Characterizing Schedules based on Recoverability

Recoverable Schedule : make sure that schedule w/ write ops, (from which another schedule has read from)) commits first

aka w commits first

r commits next

* Cascading Rollback

→ an uncommitted transaction has to be rolled back because it read an item from a transaction that failed

S: $r_1(X)$; $w_1(X)$; $r_2(X)$; $r_1(Y)$; $w_2(X)$; $w_1(Y)$; a_1; a_2

T_1	T_2
$r_1(X)$	
$w_1(X)$	
	$r_2(X)$
$r_1(Y)$	
	$w_2(X)$
$w_1(Y)$	
a_1	
	a_2

T_2 should also abort

T_1 aborts

* Cascadeless Schedule:

read items only from committed transactions (blind writes allowed)

* Strict Schedules:

read and write only allowed from committed transactions

Ex1

T_1	T_2
$r_1(X)$	
$w_1(X)$	
$r_1(Y)$	
$w_1(Y)$	
c_1	
	$r_2(X)$
	$w_2(X)$

cascadeless & strict

Ex2

T_1	T_2	T_3
$w_1(X)$		$w_3(X)$
$r_1(Y)$	read	
$w_1(Y)$		
c_1		
	$r_2(X)$	cascadeless
	$w_2(X)$	not strict
		($w_3(X)$ writes from uncommitted T_1)

Ex 3

T_1	T_2	T_3
$r_1(X)$		
$w_1(X)$		
	↓	
$r_1(Y)$		
$w_1(Y)$		
↓		
		$w_3(X)$.
	$r_2(X)$	
	$w_2(X)$	

neither cascadeless nor strict

T_3 should commit,

then only T_2 can read (cascadeless)

for T_2 to write, T_3 should commit (strict)

* Serializability Theory → determines which schedules are correct

A schedule S is said to be serializable if it is equivalent to some serial schedule of the same n transactions.

considered equivalent when

(i) result equivalent → produces same result at the final state.

(ii) conflict equivalent → ordering of conflict equivalence is the same

(iii) view equivalent

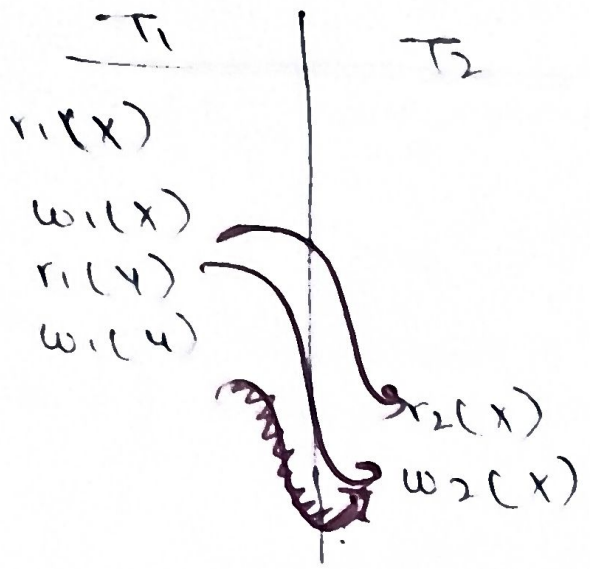
done using

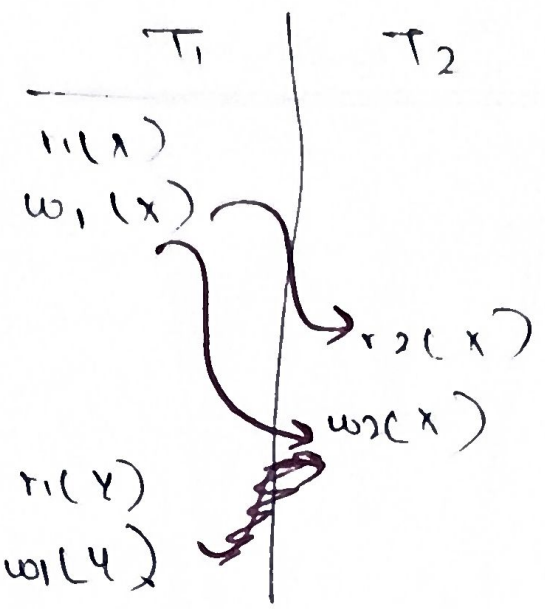
(i) ordering conflict ops

(ii) swapping non-conflict ops

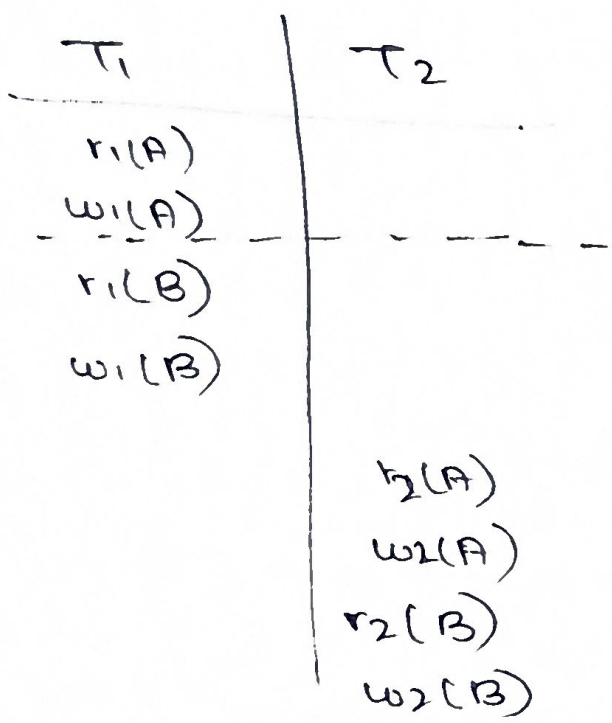
(iii) precedence graph

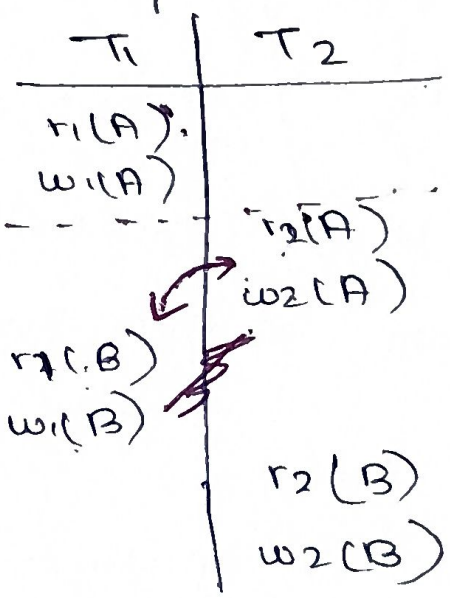
Ordering of Conflict Operations

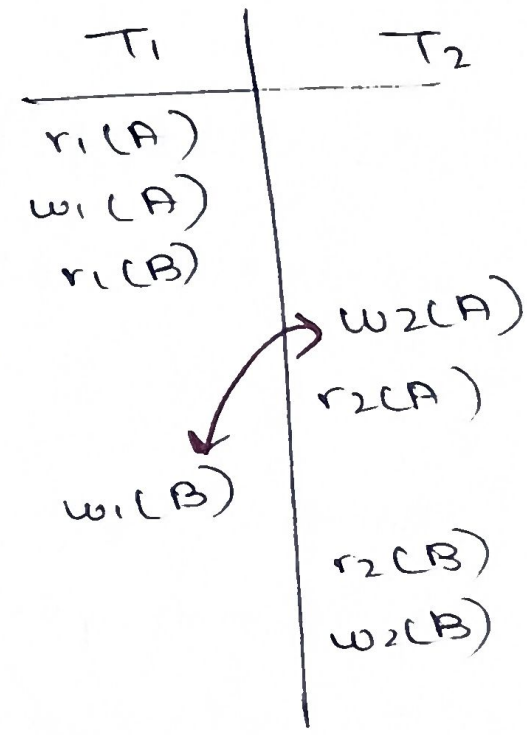


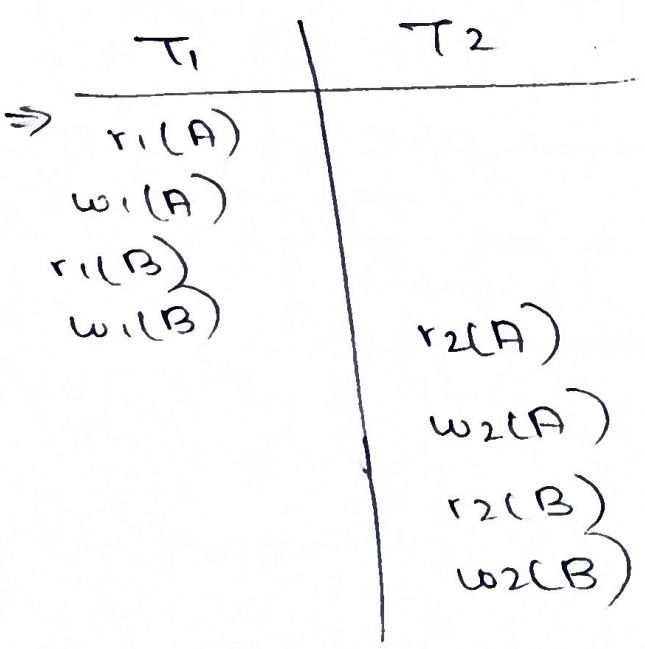


Swapping of non-conflict operations









View Serializable

UNIT 4

PRACTICE QUESTIONS ON SERIALIZABILITY

(1) Classify the following schedules as cascadeless, strict etc.

A. Schedule C: $r_1(X)$; $w_1(X)$; $r_1(Y)$; $w_1(Y)$; c_1 ; $r_2(X)$; $w_2(X)$

T_1	T_2
$r_1(X)$	
$w_1(X)$	
$r_1(Y)$	
$w_1(Y)$	
c_1	
	$r_2(X)$
	$w_2(X)$

cascadeless => commit before read operation

strict → commit before read & write

B. Schedule D: $r_1(X)$; $w_1(X)$; $w_3(X)$; $r_1(Y)$; $w_1(Y)$; c_1 ; $r_2(X)$; $w_2(X)$

T_1	T_2	T_3
$r_1(X)$		
$w_1(X)$		
		$w_3(X)$
$r_1(Y)$		
$w_1(Y)$		
c_1		
	$r_2(X)$	
	$w_2(X)$	

cascadeless ⇒ read happens after a commit

not strict ⇒ T_1 not committed, $w_3(X)$ is written before that (blind writes not allowed)

C. Schedule E: $r_1(X)$; $w_1(X)$; $r_1(Y)$; c_1 ; $w_3(X)$; $r_2(X)$; $w_2(X)$

T_1	T_2	T_3
$r_1(X)$		
$w_1(X)$		
$r_1(Y)$		
c_1		
	$r_2(X)$	$w_3(X)$
	$w_2(X)$	

not cascadeless ⇒ T3 not committed before write operation

not strict ⇒ T3 not committed before $w_2(X)$

(2) Check if the non-serial schedule is serializable based on the ordering of conflict operations

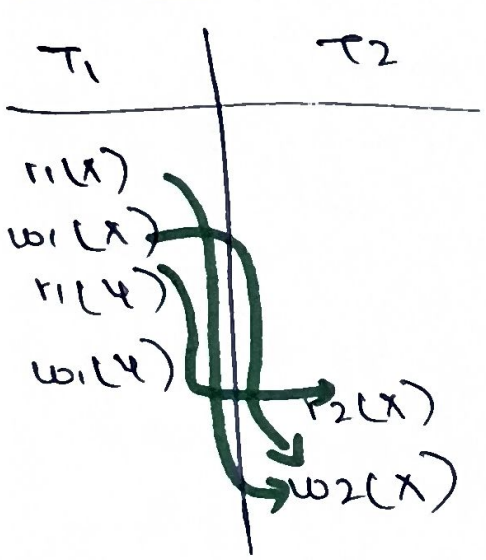
Serial

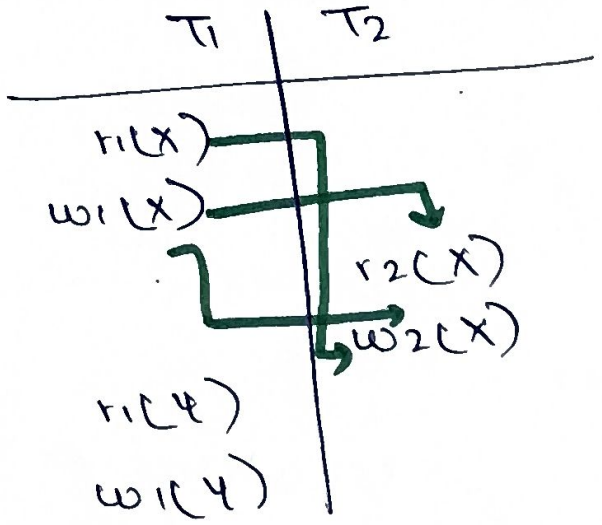
T_1	T_2
$r_1(X)$	
$w_1(X)$	
$r_1(Y)$	
$w_1(Y)$	
	$r_2(X)$
	$w_2(X)$

Non - Serial

T_1	T_2
$r_1(X)$	
$w_1(X)$	
	$r_2(X)$
	$w_2(X)$
$r_1(Y)$	
$w_1(Y)$	

Saution





Ordering of conflict operations

$r_1(X) \rightarrow w_2(X)$

$w_1(X) \rightarrow r_2(X)$

$w_1(X) \rightarrow w_2(X)$

are the same in the serializable & non-serializable schedules.

∴ The schedules are conflict ~~operati~~ equivalent.

(3) Chak if the 2 schedules are conflict equivalent by (3) the swapping of non-conflict operations

Serial

T_1	T_2
$r_1(A)$	
$w_1(A)$	
$r_1(B)$	
$w_1(B)$	
	$r_2(A)$
	$w_2(A)$
	$r_2(B)$
	$w_2(B)$

Non-Serial

T_1	T_2
$r_1(A)$	
$w_1(A)$	
	$r_2(A)$
	$w_2(A)$
$r_1(B)$	
$w_1(B)$	
	$r_2(B)$
	$w_2(B)$

Solution

T_1	T_2
$r_1(A)$	
$w_1(A)$	
	$r_2(A)$
	$w_2(A)$
$r_1(B)$	
$w_1(B)$	
	$r_2(B)$
	$w_2(B)$

$\Rightarrow$

T_1	T_2
$r_1(A)$	
$w_1(A)$	
$r_1(B)$	
	$w_2(A)$
	$r_2(A)$
$w_1(B)$	
	$r_2(B)$
	$w_2(B)$

T_1	T_2
$r_1(A)$	
$w_1(A)$	
$r_1(B)$	
$w_1(B)$	
	$r_2(A)$
	$w_2(A)$
	$r_2(B)$
	$w_2(B)$

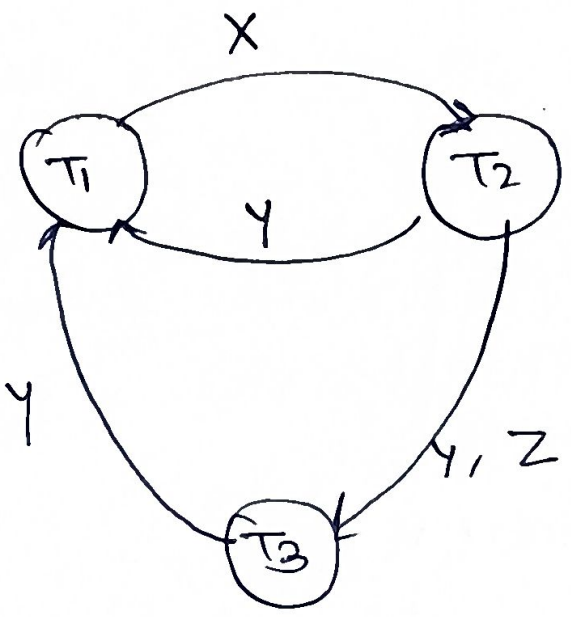
$\Rightarrow$, which is the equivalent serial schedule.

$\therefore$ It is conflict-serializable.

(4) Draw the precedence graphs for the following schedules, and check whether they are conflict serializable or not.

A.

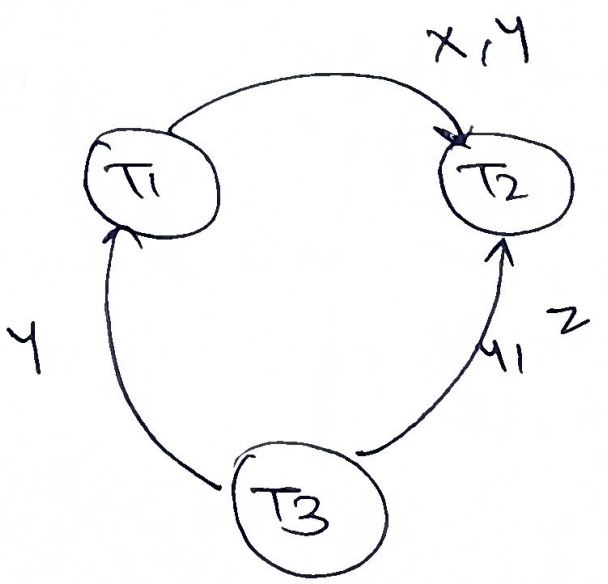
T_1	T_2	T_3
	$r_2(z)$	
	$r_2(y)$	
	$w_2(y)$	
		$r_3(y)$
		$r_3(z)$
$r_1(x)$		
$w_1(x)$		
		$w_3(y)$
		$w_3(z)$
	$r_2(x)$	
$r_1(y)$		
$w_1(y)$		
	$w_2(x)$	



not serializable - there exists a cycle

(5) B.

T_1	T_2	T_3
		$r_3(y)$
		$r_3(z)$
$r_1(x)$		
$w_1(x)$		
		$w_3(y)$
		$w_3(z)$
	$r_2(z)$	
$r_1(y)$		
$w_1(y)$		
	$r_2(y)$	
	$w_2(y)$	
	$r_2(x)$	
	$w_2(x)$	



no cycles ⇒ serializable

Serializable schedule

-> $T_3 \rightarrow T_1 \rightarrow T_2$

(5) Consider the following schedule. Check whether it is serializable or not. (5)

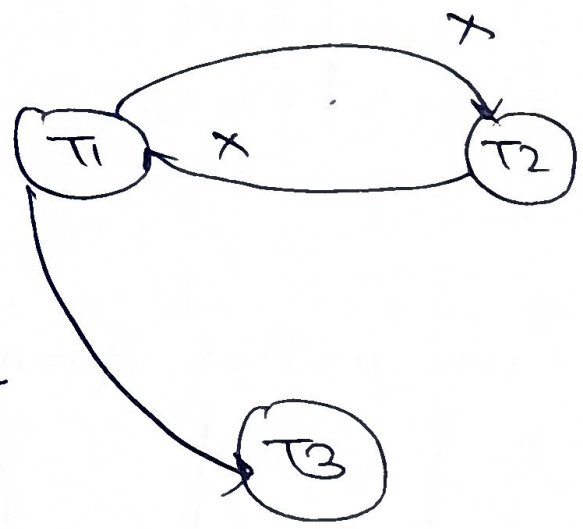
$r_1(X)$; $r_3(Y)$; $r_2(X)$; $w_1(X)$; $w_2(X)$; $w_3(Y)$;

$r_1(Z)$; $w_1(Z)$; $r_3(Z)$; $w_3(Z)$

If not serializable, swap $r_2(X)$ and $w_1(X)$. Swap non-conflicting operations to generate the equivalent serial schedule

Solution

T_1	T_2	T_3
$r_1(X)$		
		$r_3(Y)$
	$r_2(X)$	
$w_1(X)$		
	$w_2(X)$	
		$w_3(Y)$
$r_1(Z)$		
$w_1(Z)$		
		$r_3(Z)$
		$w_3(Z)$



There exists a cycle:

⇒ not serializable

swap $r_2(X)$ and $w_1(X)$

The schedule is now

$r_1(X)$; $r_3(Y)$; $w_1(X)$; $r_2(X)$; $w_2(X)$; $w_3(Y)$;

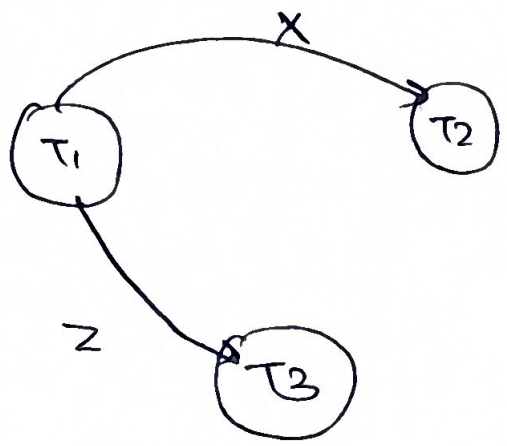
$r_1(Z)$; $w_1(Z)$; $r_3(Z)$; $w_3(Z)$

$r_1(X)$; $r_3(Y)$; $w_1(X)$; $r_2(X)$; $w_2(X)$; $w_3(Y)$; $r_1(Z)$;

$w_1(Z)$; $r_3(Z)$; $w_3(Z)$

T_1	T_2	T_3
$r_1(X)$		
		$r_3(Y)$
$w_1(X)$		
	$r_2(X)$	
	$w_2(X)$	
		$w_3(Y)$
$r_1(Z)$		
$w_1(Z)$		
		$r_3(Z)$
		$w_3(Z)$

Precedence Graph



no - cycle ⇒ serializable

order: $T_1 \rightarrow T_2 \rightarrow T_3$

or $T_1 \rightarrow T_3 \rightarrow T_2$

Swapping non - conflict operations to obtain a serial schedule

T_1	T_2	T_3
$r_1(X)$		
		$r_3(Y)$
$w_1(X)$		
	$r_2(X)$	
	$w_2(X)$	
		$w_3(Y)$
$r_1(Z)$		
$w_1(Z)$		
		$r_3(Z)$
		$w_3(Z)$

⇒

T_1	T_2	T_3
$r_1(X)$		
$r_1(Z)$	[illegible]	[illegible]
$w_1(X)$		
	$r_2(X)$	[illegible]
	$w_2(X)$	
		$w_3(Y)$
[illegible]		$r_3(Y)$
$w_1(Z)$		
		$r_3(Z)$
		$w_3(Z)$

T_1	T_2	T_3
$r_1(X)$		
$r_1(Z)$		
$w_1(X)$		
$w_1(Z)$		
	$w_2(X)$	
		$w_3(Y)$
		$r_3(Y)$
	$r_2(X)$	
		$r_3(Z)$
		$w_3(Z)$

swap $w_2(X)$ and $r_3(Z)$ and $r_2(X)$ and $w_3(Z)$ to get $T_1 \rightarrow T_3 \rightarrow T_2$

$\Rightarrow$

T_1	T_2	T_3
$r_1(X)$		
$r_1(Z)$		
$w_1(X)$		
$w_1(Z)$		
	$w_2(X)$	
	$r_2(X)$	
		$r_3(Y)$
		$w_3(Y)$
		$r_3(Z)$
		$w_3(Z)$

$T_1 \rightarrow T_2 \rightarrow T_3$ equivalent serial schedule.

(6) Consider the following schedule, check if it is serializable or not. If it is, generate the equivalent serial schedule.

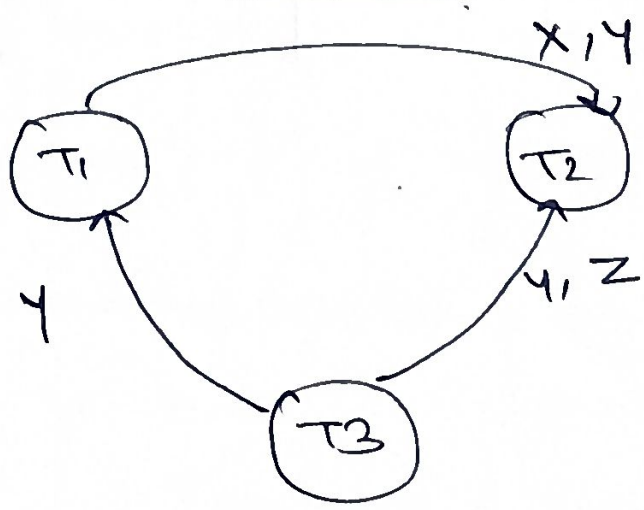
$r_3(Y)$; $r_3(Z)$; $r_1(X)$; $w_1(X)$; $w_3(Y)$; $w_3(Z)$; $r_2(Z)$;

$r_1(Y)$; $w_1(Y)$; $r_2(Y)$; $w_2(Y)$; $r_2(X)$; $w_2(X)$;

Solution

T_1	T_2	T_3
		$r_3(Y)$
		$r_3(Z)$
$r_1(X)$		
$w_1(X)$		
		$w_3(Y)$
		$w_3(Z)$
	$r_2(Z)$	
$r_1(Y)$		
$w_1(Y)$		
	$r_2(Y)$	
	$w_2(Y)$	
	$r_2(X)$	
	$w_2(X)$	

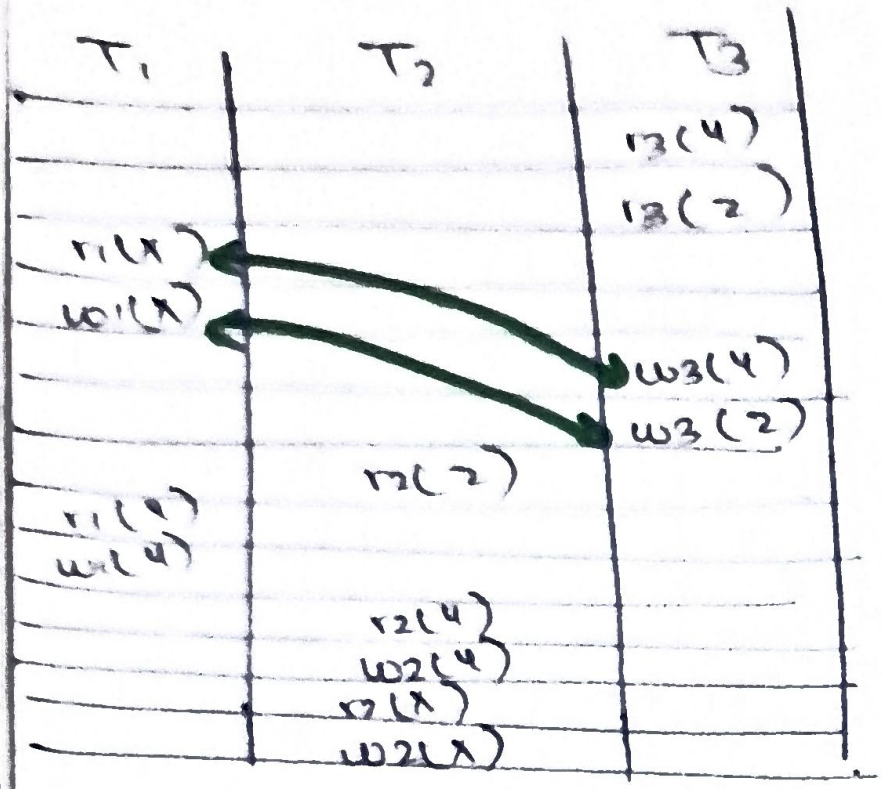
Precedence Graph



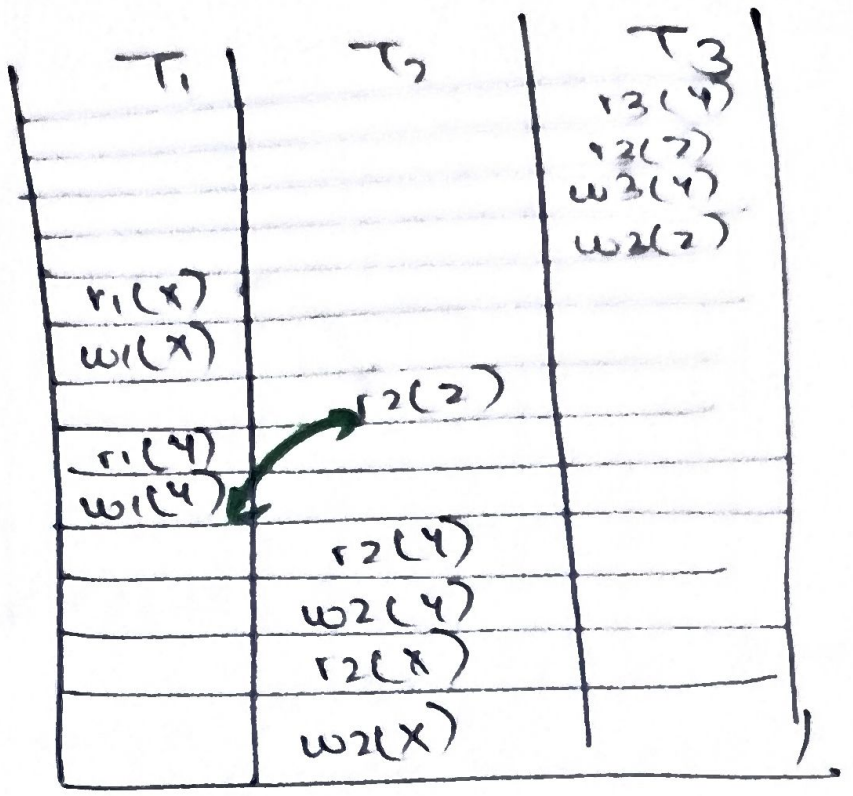
no cycles $\Rightarrow$ serializable

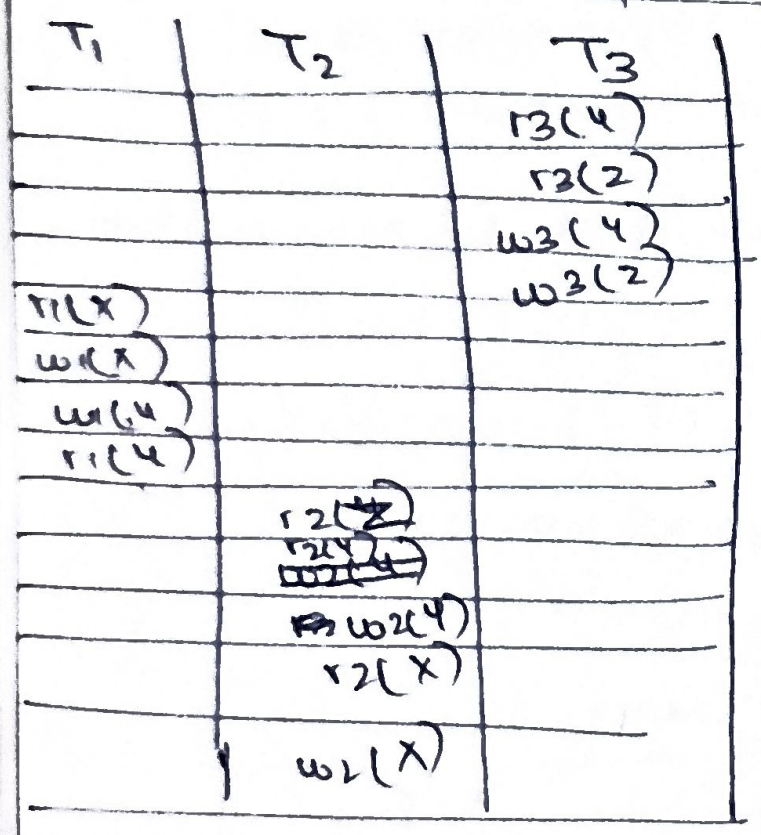
order of serial schedule: $T_3 \rightarrow T_1 \rightarrow T_2$

Swapping of non-conflicting operations to obtain a serial schedule

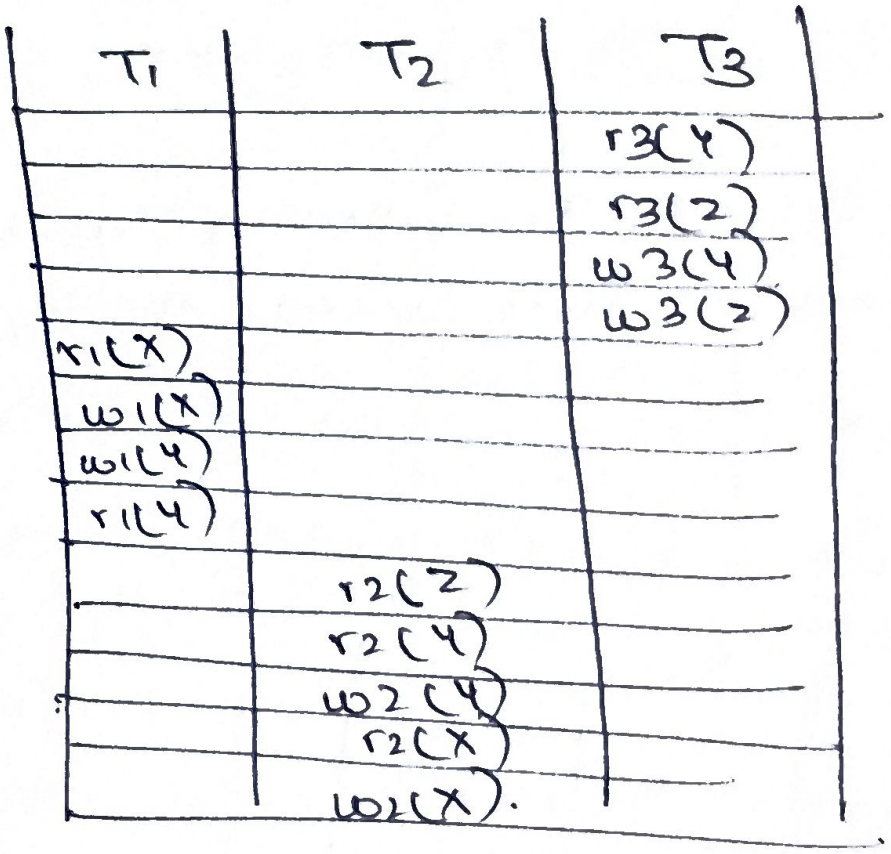


⇒





=



Steps to identify View Serializability?

To check if a schedule is view serializable,

check for (i) initial read

(ii) final write

order of update operations must also be the same.

Alternatively,

first check if the schedule is conflict-serializable

If conflict-serializable ⇒ view serializable

If not conflict-serializable ⇒ may or may not be view serializable

(i) check for blind writes ↑ : no blind writes ⇒ not view serializable

a write operation without a read operation

if there are blind writes ⇒ may or may not be view serializable

~~⇒ draw dependency graph~~

write down initial read, update & final read values

Problems

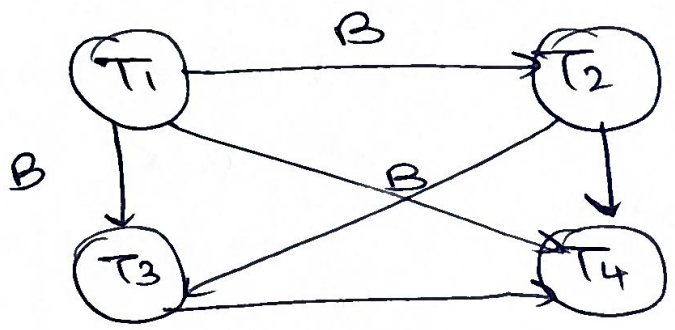
① Check if the following schedule is view serializable or not

T_1	T_2	T_3	T_4
R(A)			
	R(A)		
		R(A	
			R(A)
W(B)			
	W(B)		
		W(B)	
			W(B)

(OR)

	A	B
Initial Read	T_1, T_2 T_3, T_4	—
Update	—	T_1, T_2 T_3 T_4
Final Write	—	T_1 T_2 T_3 T_4

Solution: (i) check for conflict serializability



no cycles ⇒ conflict serializable

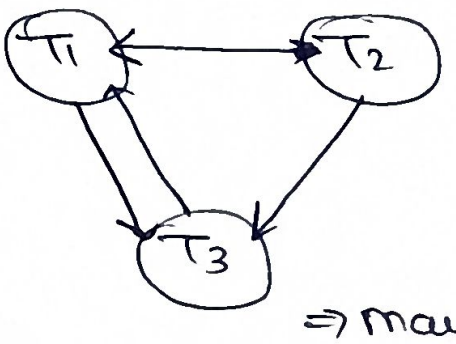
⇒ view serializable

order: $T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow T_4$

② Check whether the given schedule is view serializable or not

T_1	T_2	T_3
R(A)		
	R(A)	
		W(A)
W(A)		

conflict serializability



Not conflict serializable

⇒ may or may not be view serializable

check for blind writes

W3(A) is a blind write

⇒ may or maynot be view serializable

Finding Dependencies

	A
Initial Read	T_1, T_2
Update	T_1, T_3
Final write	T_1, T_3

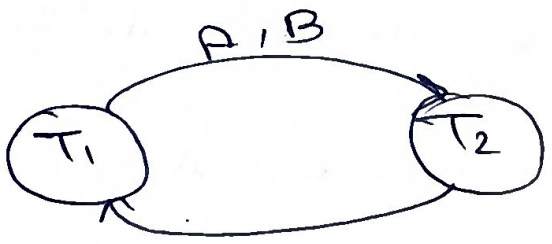
$T_1 \rightarrow T_3$
$T_2 \rightarrow T_1$
$T_3 \rightarrow T_1$
} conflicting

not view - serializable

(3) @heck whether the given schedule is view serializable or not.

T_1	T_2
R(A)	
	R(A)
W(A)	
	W(A)
R(B)	
	R(B)
W(B)	
	W(B)

Solution @heck for conflict serializability



not conflict serializable ⇒ may or may not be view serializable

(ii) check for blind writes - no blind writes ⇒ not view serializable

	A	B
Initial Read	T_1, T_2	T_1, T_2
Update	T_1, T_2	T_1, T_2
Final Write	T_2	T_2

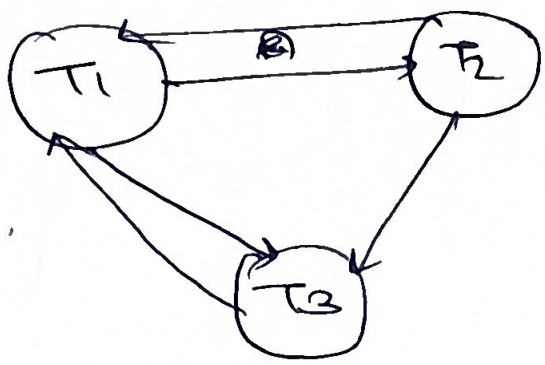
$T_1 \rightarrow T_2$
$T_2 \rightarrow T_1$
} conflicting

(4) Check if the given schedule is view serializable or not. If yes, give the serial schedule.

S: $R_1(A), W_2(A), R_3(A), W_1(A), W_3(A)$

T_1	T_2	T_3
$R_1(A)$		
	$W_2(A)$	
		$R_3(A)$
$W_1(A)$		
		$W_3(A)$

conflict serializability



not conflict serializable ⇒ may or may not be view serializable

Blind write check: $W_2(A)$ in T_2 is a blind write

⇒ may or may not be view serializable.

	A
Initial Read	T_1, T_3
Update	T_1, T_2, T_3
Final Write.	T_1, T_2, T_3

dependencies

~~$T_1 \rightarrow T_2 \rightarrow T_3$~~

T_1 reads before T_2

$\Rightarrow T_1 \rightarrow T_2$

T_3 updates last

$\Rightarrow T_3$ executes last

$\Rightarrow T_1 \rightarrow T_2 \rightarrow T_3$

$\Rightarrow$ **view serializable**

Check if the given schedule is view serializable

S_1: $r_1(x)$; $r_3(4)$; $w_1(x)$; $r_2(x)$; $w_2(x)$; $w_3(4)$;

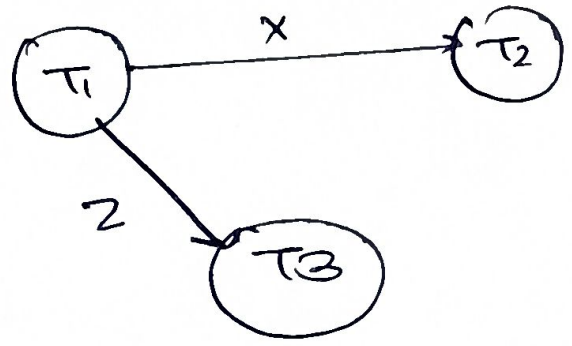
$r_1(z)$; $w_1(z)$; $r_3(z)$; $w_3(z)$

T_1	T_2	T_3
$r_1(x)$		
		$r_3(4)$
$w_1(x)$	~~$w_2(x)$~~	
	$r_2(x)$	
	$w_2(x)$	
		$w_3(4)$
$r_1(z)$		
$w_1(z)$		
		$r_3(z)$
		$w_3(z)$

$T_1 \rightarrow T_2$

$T_1 \rightarrow T_3$

(1) check if conflict serializable



conflict serializable

⇒ it is view serializable

	x	y	z
Initial Read	T_1, T_2	T_3	T_1, T_3
Update	T_1, T_2	T_3	T_1, T_3
Final Write	T_2	T_3	T_3

T_1 before T_3 and T_1 before T_2

possible serial schedules:- $T_1 \rightarrow T_2 \rightarrow T_3$

or $T_1 \rightarrow T_3 \rightarrow T_2$