Concurrency Control: A database must provide a mechanism that will ensure that all possible schedules are either conflict or view serializable, and are recoverable and preferably cascadeless.

Concurrency Control Protocols: Concurrency-control protocols allow concurrent schedules, but ensure that the schedules are conflict/view serializable, and are recoverable and cascadeless. These protocols generally do not examine the precedence graph as it is being created. Instead a protocol imposes a discipline that avoids nonseralizable schedules.

Types of Concurrency Protocols :

i) Lock-Based Protocols ii) Timestamp-Based Protocols
iii) Validation-Based Protocols

Lock-Based Protocols

1. Two phase locking
2. Graph based protocol
3. Timeout based protocol

- A lock is a mechanism to control concurrent access to a data item

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.

- Lock requests are made to concurrency-control manager. Transaction can proceed only after request is granted.

Lock-compatibility matrix

	S	X
S	true	false
X	false	false

- 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

- 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.
- 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.

Example of a transaction performing locking:

T_2: **lock-S**(*A*);
read (*A*);
unlock(*A*);
lock-S(*B*);
read (*B*);
unlock(*B*);
display(*A*+*B*);

T1 : lock-x(B);
read(B);
B = B - 50;
write(B);
unlock(B);
lock-x(A);
read(A);
A = A + 50;
write(A); unlock(A);

Locking as above is not sufficient to guarantee serializability — if *A* and *B* get updated in-between the read of *A* and *B*, the displayed sum would be wrong. (ex)

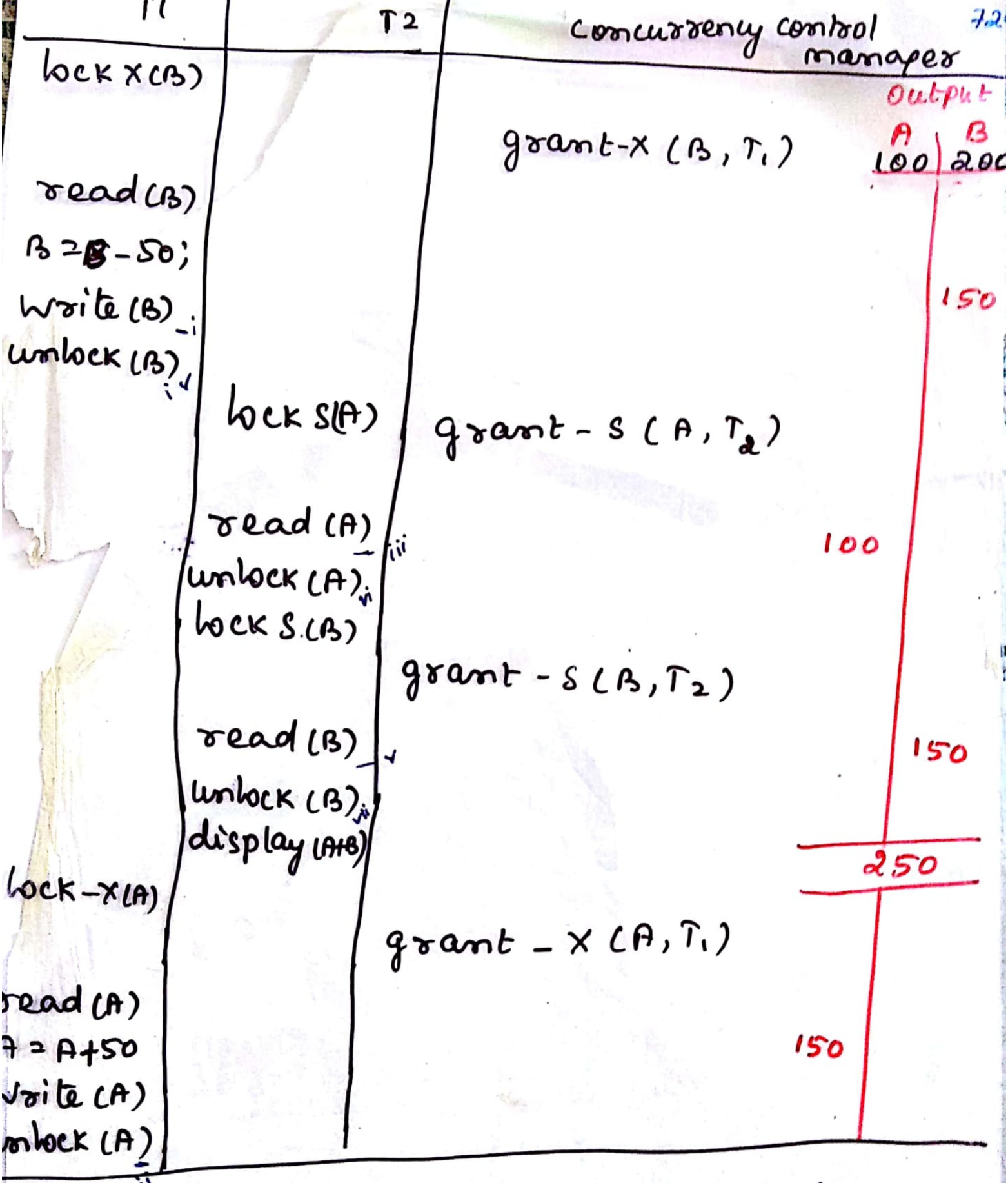
A locking **protocol** is a set of rules followed by all transactions while requesting and releasing locks. Locking protocols restrict the set of possible schedules.

Pitfalls of Lock-Based Protocols

Consider the partial schedule

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*)	

Neither T_3 nor T_4 can make progress — executing lock-S(*B*) causes T_4 to wait for T_3 to release its lock on *B*, while executing **lock-**



* T_1 unlocked data item B too early, so T_2 saw an inconsistent state.

X(A) causes T_3 to wait for T_4 to release its lock on A. Such a situation is called a **deadlock**. To handle a deadlock one of T_3 or T_4 must be rolled back and its locks released. The potential for deadlock exists in most locking protocols. Deadlocks are a necessary evil.

Starvation is also possible if 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.

The Two-Phase Locking Protocol

This is a protocol which 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

This protocol assures conflict serializability. It can be proved that the transactions can be serialized in the order of their **lock points** (i.e. the point where a transaction acquired its final lock).

Drawbacks:

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

ii) Cascading roll-back is possible under two-phase locking. To avoid this, follow a modified protocol called **strict two-phase locking**. Here a transaction must hold all its exclusive locks till it commits/aborts. **Rigorous two-phase locking** is even stricter: here

Example for Two phase locking which leads to cascading rollback:

T_1	T_2	T_3
lock-X(A)		
read(A)		
lock-S(B)		
read(B)		
write(A) ←		
unlock(A)		
unlock(B)		
	lock-X(A)	
	read(A)	
	write(A) ←	
	unlock(A)	
		lock-S(A)
		read(A)
		unlock(A)

* Failure of T_1 after the read(A) of T_2 leads to cascading rollback of T_2 and T_3.

* cascading rollbacks are avoided by making modifications in two phase locking.

1. Strict two phase locking
2. Rigorous two phase locking

both shared & exclusive

all locks are held till commit/abort. In this protocol transactions can be serialized in the order in which they commit.

iii) There can be conflict serializable schedules that cannot be obtained if two-phase locking is used. However, in the absence of extra information (e.g., ordering of access to data), two-phase locking is needed for conflict serializability in the following sense:

Given a transaction T_i that does not follow two-phase locking, we can find a transaction T_j that uses two-phase locking, and a schedule for T_i and T_j that is not conflict serializable.

Two-phase locking with lock conversions — generates conflict serializable schedules

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.

Automatic Acquisition of Locks

A transaction T_i issues the standard read/write instruction, without explicit locking calls.

The operation **read**(D) is processed as:

```
if Ti has a lock on D
 then
      read(D)
 else begin
      if necessary wait until no other
         transaction has a lock-X on D
      grant Ti a lock-S on D;
```

read(D)
end

The operation **write**(D) is processed as:

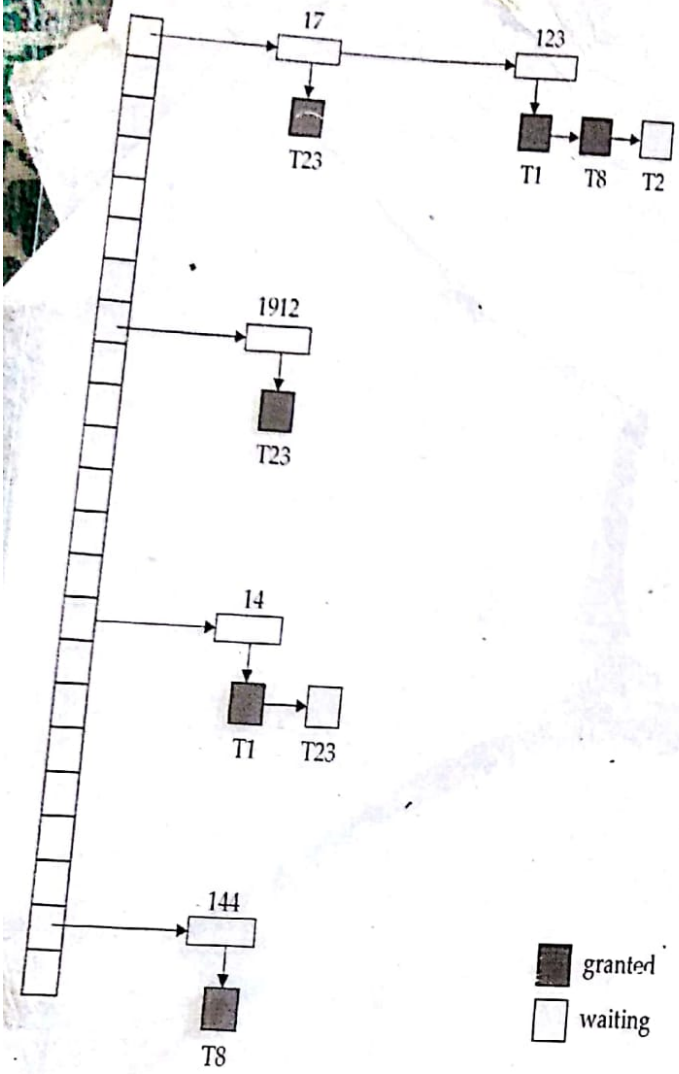
if T_i has a **lock-X** on D
then
write(D)
else begin
if necessary wait until no other trans, has any lock on D,
if T_i has a **lock-S** on D
then
upgrade lock on D to **lock-X**
else
grant T_i a **lock-X** on D
write(D)
end;

All locks are released after commit or abort

Implementation of Locking

- A **lock manager** can be implemented as a separate process to which transactions send lock and unlock requests
- The lock manager replies to a lock request by sending a lock grant messages (or a message asking the transaction to roll back, in case of a deadlock)
- The requesting transaction waits until its request is answered
- The lock manager maintains a data-structure called a **lock table** to record granted locks and pending requests
- The lock table is usually implemented as an in-memory hash table indexed on the name of the data item being locked

Lock Table

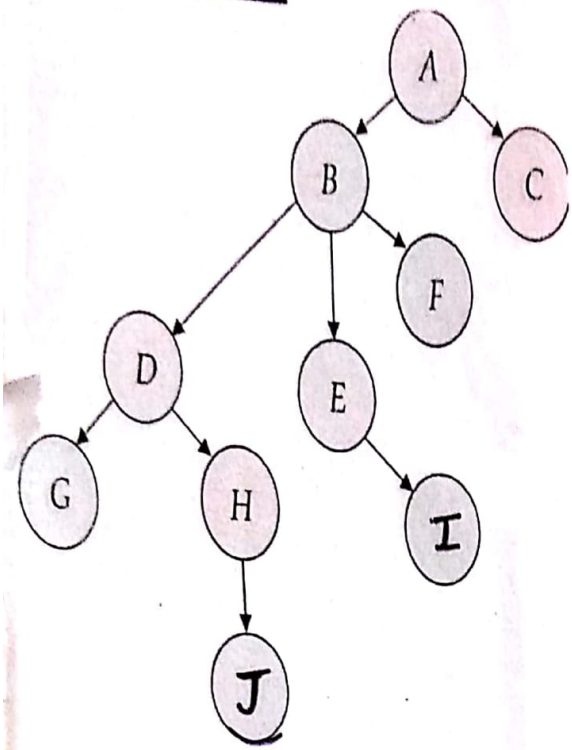


- Filled rectangles indicate granted locks, empty ones indicate waiting requests
- Lock table also records the type of lock granted or requested
- New request is added to the end of the queue of requests for the data item, and granted if it is compatible with all earlier locks
- Unlock requests result in the request being deleted, and later requests are checked to see if they can now be granted
- If transaction aborts, all waiting or granted requests of the transaction are deleted
- lock manager may keep a list of locks held by each transaction, to implement this efficiently

Graph-Based Protocols

- Graph-based protocols are an alternative to two-phase locking
- Impose a partial ordering $\rightarrow$ on the set $\mathbf{D} = \{d_1, d_2, ..., d_h\}$ of all data items.
- If $d_i \rightarrow d_j$, then any transaction accessing both d_i and d_j must access d_i before accessing d_j.
- Implies that the set **D** may now be viewed as a directed acyclic graph, called a *database graph*. The *tree-protocol* is a simple kind of graph protocol.

Tree Protocol



1. Only exclusive locks are allowed.
2. The first lock by T_i may be on any data item. Subsequently, a data Q can be locked by T_i only if the parent of Q is currently locked by T_i.
3. Data items may be unlocked at any time.
4. A data item that has been locked and unlocked by T_i cannot subsequently be relocked by T_i

The tree protocol ensures conflict serializability as well as freedom from deadlock.Unlocking may occur earlier in the tree-locking protocol than in the two-phase locking protocol.

Advanatages:

i) Shorter waiting times, and increase in concurrency

ii) Protocol is deadlock-free, no rollbacks are required

iii) Schedules not possible under two-phase locking are possible under tree protocol, and vice versa.

Drawbacks

i) Protocol does not guarantee recoverability or cascade freedom

- Need to introduce commit dependencies to ensure recoverability

ii) Transactions may have to lock data items that they do not access.

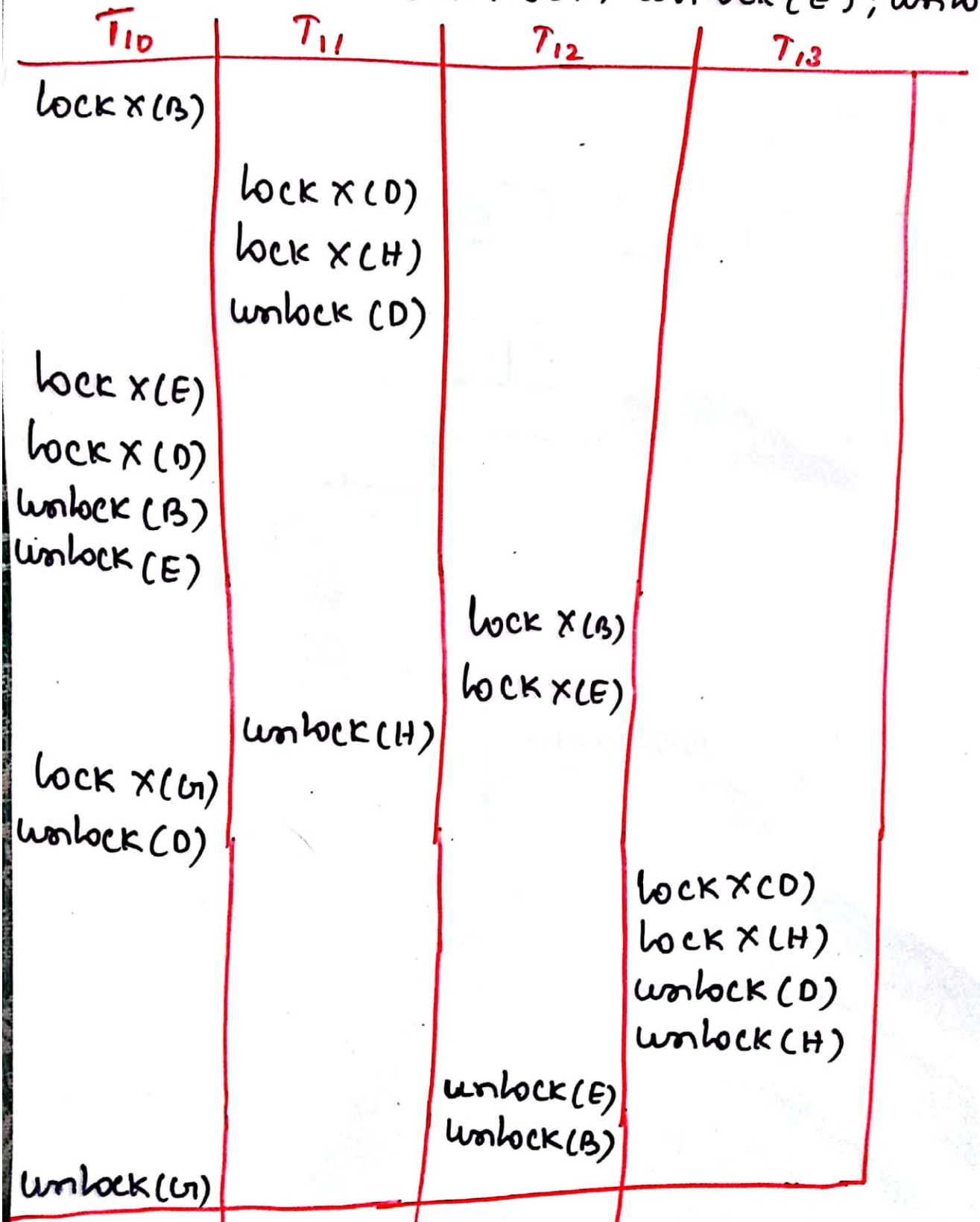
- increased locking overhead, and additional waiting time
- potential decrease in concurrency

Problem: Generate a conflict serializable schedule using a tree protocol for the given transactions T_{10}, T_{11}, T_{12}, T_{13}

T_{10} : lock X(B); lock X(E); lock X(D); unlock(B);
unlock(E); lock X(G); unlock(D); unlock(G);

T_{11} : lock X(D); lock X(H); unlock(D); unlock(H);

T_{12} : lock X(B); lock X(E); unlock(E); unlock(B);



T_{13} : lock X(D); lock X(H); unlock(D); unlock(H);

Deadlock Handling

Consider the following two transactions:

T_1: write (X)
write(Y)

T_2: write(Y)
write(X)

Schedule with deadlock

T_1	T_2
lock-X on A	
write (A)	
	lock-X on B
	write (B)
	wait for **lock-X** on A
wait for **lock-X** on B	

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 (graph-based protocol).

More Deadlock Prevention Strategies

1.wait-die scheme — non-preemptive

- Older transaction may wait for younger one to release data item. Younger transactions never wait for older ones; they are rolled back instead. A transaction may die several times before acquiring needed data item

2.wound-wait scheme — preemptive

- Older transaction *wounds* (forces rollback) of younger transaction instead of waiting for it. Younger transactions

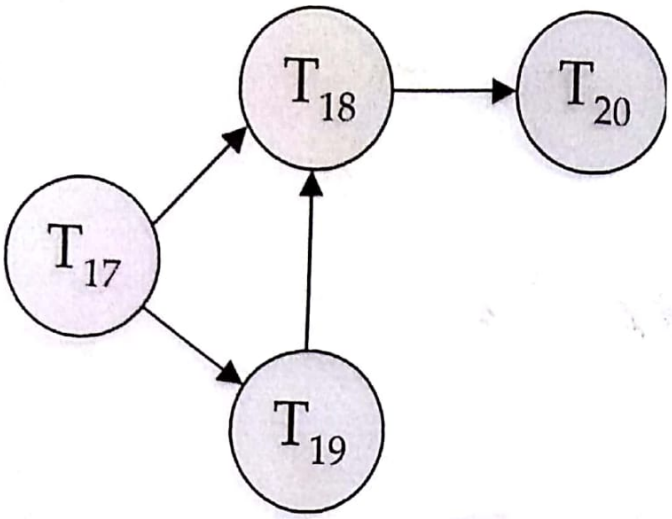
may wait for older ones. May be fewer rollbacks than *wait-die* scheme.

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.

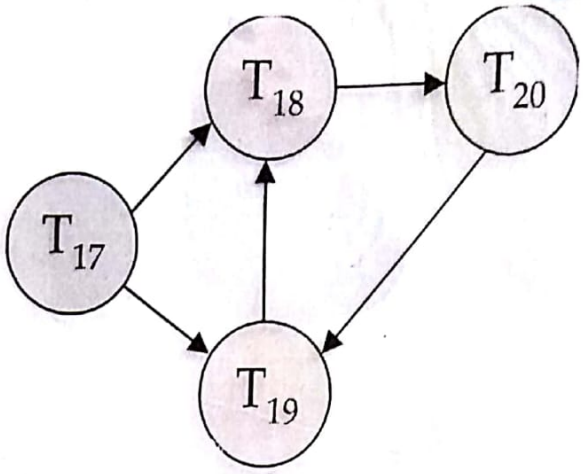
3.Timeout-Based Schemes: A transaction waits for a lock only for a specified amount of time. After that, the wait times out and the transaction is rolled back. Thus deadlocks are not possible. Simple to implement; but starvation is possible. Also difficult to determine good value of the timeout interval.

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.



Wait-for graph without a cycle



Wait-for graph with a cycle

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

Multiple Granularity

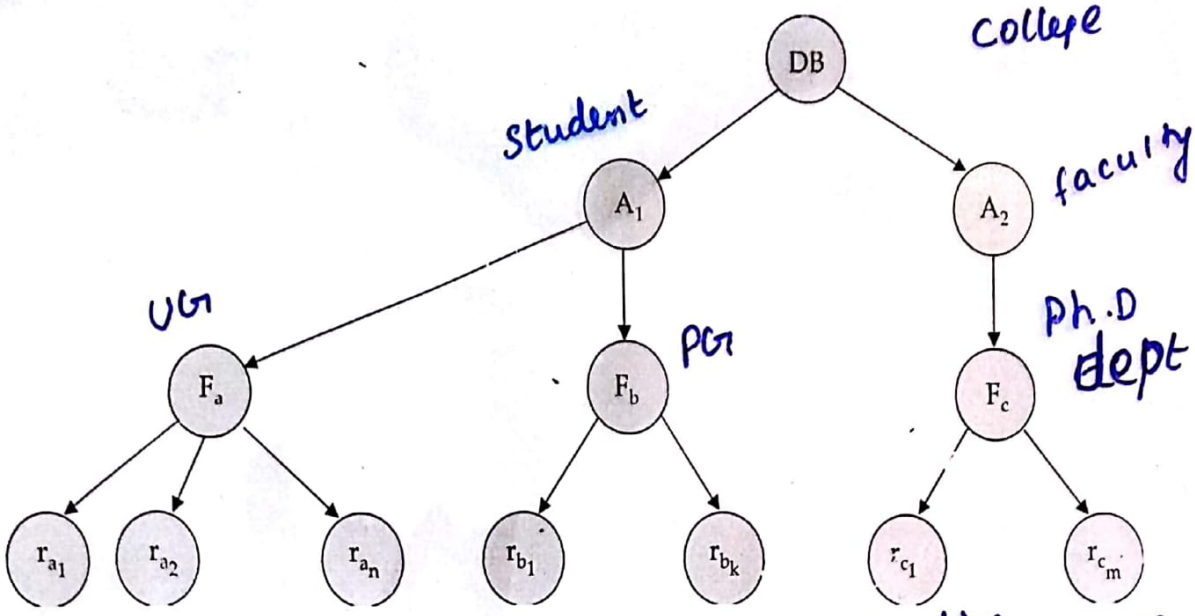
Allow data items to be of various sizes and define a hierarchy of data granularities, where the small granularities are nested within larger ones.Can be represented graphically as a tree. When a transaction locks a node in the tree *explicitly*, it *implicitly* locks all the node's descendents in the same mode.

Granularity of locking (level in tree where locking is done):

fine granularity (lower in tree): high concurrency, high locking overhead

coarse granularity (higher in tree): low locking overhead, low concurrency

Example of Granularity Hierarchy



The levels, starting from the coarsest (top) level are

- *database*
- *area*
- *file*
- *record*

Intention Lock Modes

In addition to S and X lock modes, there are three additional lock modes with multiple granularity:

- **intention-shared (IS):** indicates explicit locking at a lower level of the tree but only with shared locks.
- **intention-exclusive** (IX): indicates explicit locking at a lower level with exclusive or shared locks
- **shared and intention-exclusive** (SIX): the subtree rooted by that node is locked explicitly in shared mode and explicit locking is being done at a lower level with exclusive-mode locks.

intention locks allow a higher level node to be locked in S or X mode without having to check all descendent nodes.

Compatibility Matrix with Intention Lock Modes

	IS	IX	S	SIX	X
IS	true	true	true	true	false
IX	true	true	false	false	false
S	true	false	true	false	false
SIX	true	false	false	false	false
X	false	false	false	false	false

Multiple Granularity Locking Scheme

Transaction T_i can lock a node Q, using the following rules:

1. The lock compatibility matrix must be observed.
2. The root of the tree must be locked first, and may be locked in any mode.
3. A node Q can be locked by T_i in S or IS mode only if the parent of Q is currently locked by T_i in either IX or IS mode.
4. A node Q can be locked by T_i in X, SIX, or IX mode only if the parent of Q is currently locked by T_i in either IX or SIX mode.
5. T_i can lock a node only if it has not previously unlocked any node (that is, T_i is two-phase).
6. T_i can unlock a node Q only if none of the children of Q are currently locked by T_i.

Observe that locks are acquired in root-to-leaf order, whereas they are released in leaf-to-root order.

Lock granularity escalation: in case there are too many locks at a particular level, switch to higher granularity S or X lock.

Timestamp-Based Protocols

- Each transaction is issued a timestamp when it enters the system. If an old transaction T_i has time-stamp TS(T_i), a new transaction T_j is assigned time-stamp TS(T_j) such that TS(T_i) <TS(T_j).
- The protocol manages concurrent execution such that the time-stamps determine the serializability order.
- In order to assure such behavior, the protocol maintains for each data Q two timestamp values:

  W-timestamp(Q) is the largest time-stamp of any transaction that executed **write**(Q) successfully.

  R-timestamp(Q) is the largest time-stamp of any transaction that executed **read**(Q) successfully.

The timestamp ordering protocol ensures that any conflicting **read** and **write** operations are executed in timestamp order.

Suppose a transaction T_i issues a read(Q)

1. If TS(T_i) < **W**-timestamp(Q), then T_i needs to read a value of Q that was already overwritten. Hence, the **read** operation is rejected, and T_i is rolled back.
2. If TS(T_i) ≥ **W**-timestamp(Q), then the **read** operation is executed, and R-timestamp(Q) is set to **max**(R-timestamp(Q), TS(T_i)).

Suppose that transaction T_i issues write(Q)

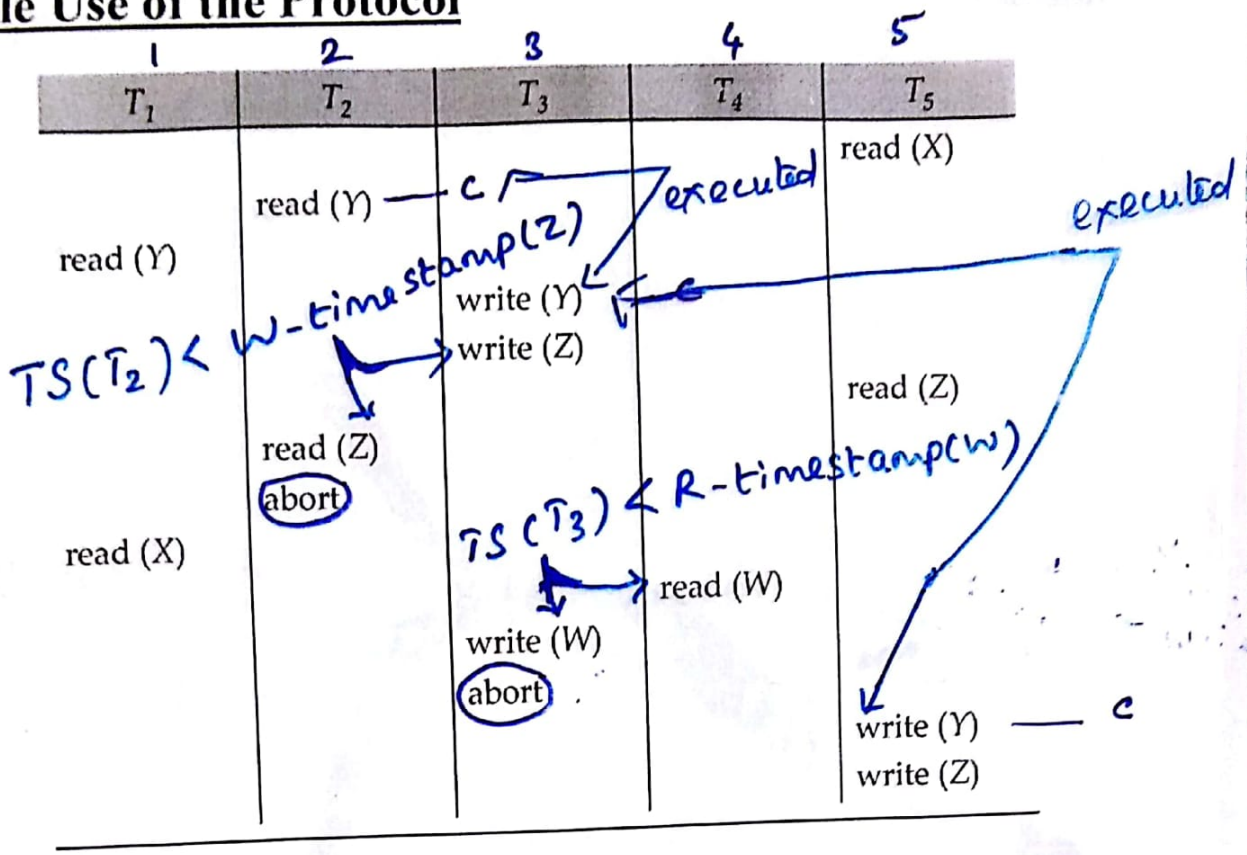
a. If TS(T_i) < R-timestamp(Q), then the value of Q that T_i is producing was needed previously, and the system assumed that that value would never be produced. Hence, the **write** operation is rejected, and T_i is rolled back.

* Timestamp is assigned by the database system
* Two simple methods are
  1. System clock
  2. Logical counter

b. If TS(T_i) < W-timestamp(Q), then T_i is attempting to write an obsolete value of Q. Hence, this **write** operation is rejected, and T_i is rolled back.

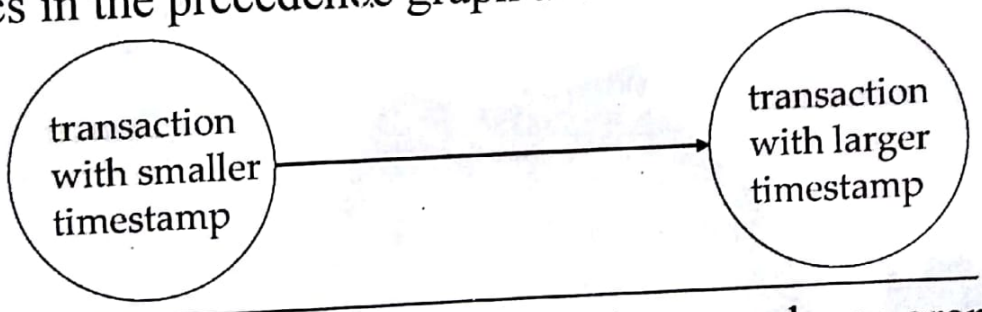
c. Otherwise, the **write** operation is executed, and W-timestamp(Q) is set to TS(T_i).

Example Use of the Protocol



Correctness of Timestamp-Ordering Protocol:

The timestamp-ordering protocol guarantees serializability since all the arcs in the precedence graph are of the form:



- Thus, there will be no cycles in the precedence graph
- Timestamp protocol ensures freedom from deadlock as no transaction ever waits.

Adv:

Timestamp Protocol : Ensure both serializability and deadlock free.

DisAdv : NO cascade free and NO recoverable schedules.

- But the schedule may not be cascade-free, and may not even be recoverable.

Recoverability and Cascade Freedom (ex) – previous problem read cy)

Problem with timestamp-ordering protocol:

- Suppose T_i aborts, but T_j has read a data item written by T_i
- Then T_j must abort; if T_j had been allowed to commit earlier, the schedule is not recoverable.
- Further, any transaction that has read a data item written by T_j must abort
- This can lead to cascading rollback --- that is, a chain of rollbacks

Solution 1:

- A transaction is structured such that its writes are all performed at the end of its processing
- All writes of a transaction form an atomic action; no transaction may execute while a transaction is being written
- A transaction that aborts is restarted with a new timestamp

Solution 2: Limited form of locking: wait for data to be committed before reading it

Solution 3: Use commit dependencies to ensure recoverability

Thomas' Write Rule

- Modified version of the timestamp-ordering protocol in which obsolete **write** operations may be ignored under certain circumstances.
- When T_i attempts to write data item Q, if $TS(T_i) <$ W-timestamp(Q), then T_i is attempting to write an obsolete value of $\{Q\}$.
- Rather than rolling back T_i as the timestamp ordering protocol would have done, this {**write**} operation can be

ignored. Otherwise this protocol is the same as the timestamp ordering protocol.

- Thomas' Write Rule allows greater potential concurrency.
- Allows some view-serializable schedules that are not conflict-serializable.

View Serializability

- Let S and S' be two schedules with the same set of transactions. S and S' are **view equivalent** if the following three conditions are met, for each data item Q,
  1. If in schedule S, transaction T_i reads the initial value of Q, then in schedule S' also transaction T_i must read the initial value of Q.
  2. If in schedule S transaction T_i executes **read**(Q), and that value was produced by transaction T_j (if any), then in schedule S' also transaction T_i must read the value of Q that was produced by the same **write**(Q) operation of transaction T_j.
  3. The transaction (if any) that performs the final **write**(Q) operation in schedule S must also perform the final **write**(Q) operation in schedule S'.
- View equivalence is also based purely on **reads** and **writes** alone.
  - A schedule S is **view serializable** if it is view equivalent to a serial schedule.
  - Every conflict serializable schedule is also view serializable.

Below is a schedule which is view-serializable but *not* conflict serializable.

1 2 3

T_3	T_4	T_6
read(Q)		
	write(Q)	
write(Q)		
		write(Q)

Every view serializable schedule that is not conflict serializable has **blind writes.**

In this schedule as per Timestamp based Protocol T_3 is roll back because of write(Q) [TS[T_3] < W-timestamp(Q)] But as per Thomas write rule, T_3 of write[Q] is ignored instead of roll back.

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. Assume for simplicity that the validation and write phase occur together, atomically and serially.(ie) only one transaction executes validation/write at a time.
- Also called as **optimistic concurrency control** since transaction executes fully in the hope that all will go well during validation
- 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. Because the serializability order is not pre-decided, and relatively few 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) ⟶ Same data item (finish-write)

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. (different data items)

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

Justification: Either the first condition is satisfied, and there is no overlapped execution, or the second condition is satisfied and

- the writes of T_j do not affect reads of T_i since they occur after T_i has finished its reads.
- the writes of T_i do not affect reads of T_j since T_j does not read any item written by T_i.

Schedule Produced by Validation

Example of schedule produced using validation

T_{25}	T_{26}
read (B)	
	read (B)
	$B := B - 50$
	read (A)
	$A := A + 50$
read (A)	
⟨ *validate* ⟩	
display ($A + B$)	
	⟨ *validate* ⟩
	write (B)
	write (A)

Recovery System

Failure Classification

1. Transaction failure:

i) Logical errors: transaction cannot complete due to some internal error condition, such as bad input, data not found, over flow or resource limit exceeded.

ii) System errors: the database system must terminate an active transaction due to an error condition (e.g., deadlock)

2. System crash: a power failure or other hardware or software failure causes the system to crash.

Fail-stop assumption: non-volatile storage contents are assumed to not be corrupted by system crash

- Database systems have numerous integrity checks to prevent corruption of disk data

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

Destruction is assumed to be detectable: disk drives use checksums to detect failures

Recovery Algorithms

Consider transaction T_i that transfers \$50 from account A to account B

Two updates: subtract 50 from A and add 50 to B

Transaction T_i requires updates to A and B to be output to the database.

- A failure may occur after one of these modifications have been made but before both of them are made.
- Modifying the database without ensuring that the transaction will commit may leave the database in an inconsistent state

- Not modifying the database may result in lost updates if failure occurs just after transaction commits

Recovery algorithms have two parts :

i) Actions taken during normal transaction processing to ensure enough information exists to recover from failures

ii) Actions taken after a failure to recover the database contents to a state that ensures atomicity, consistency and durability

Storage Structure

Volatile storage:

- Does not survive system crashes
- examples: main memory, cache memory

Nonvolatile storage:

- survives system crashes
- examples: disk, tape, flash memory
- 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

Stable-Storage Implementation

- Maintain multiple copies of each block on separate disks — RAID
- Copies can be at remote sites to protect against disasters such as fire or flooding — remote backup systems
- Failure during data transfer can still result in inconsistent copies: Block transfer can result in

i) Successful completion ii) Partial failure: destination block has incorrect information iii)Total failure: destination block was never updated

- 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.

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

Step 1: First find inconsistent blocks:

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

Better solution:

- Record in-progress disk writes on non-volatile storage (Non-volatile RAM or special area of disk).
- Use this information during recovery to find blocks that may be inconsistent, and only compare copies of these.
- Used in hardware RAID systems

Step2 : 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

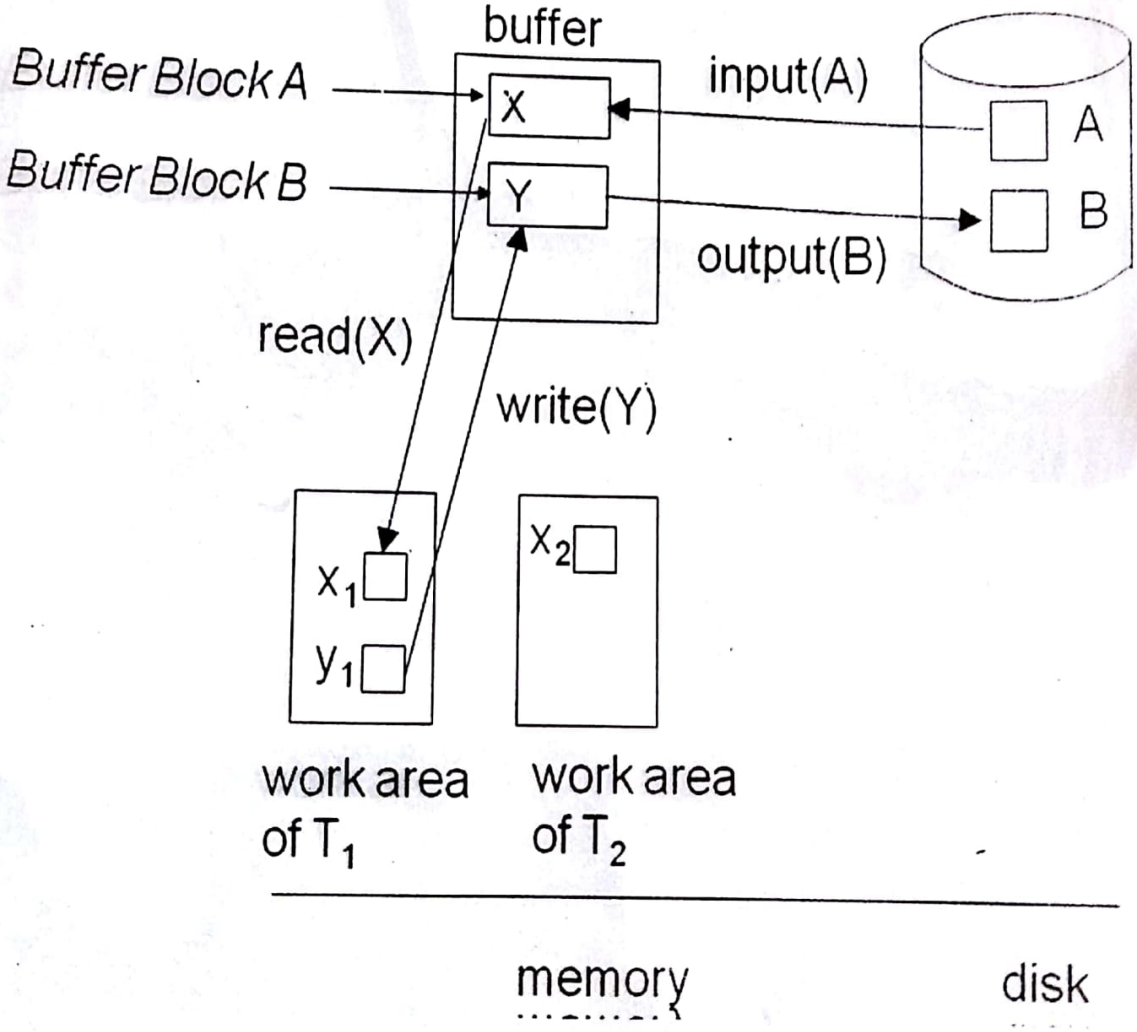
Physical blocks are those blocks residing on the disk.
Buffer blocks are the blocks residing temporarily in main memory.

Block movements between disk and main memory are initiated through the following two operations:

i) **input**(B) transfers the physical block B to main memory.

ii) **output** (B) transfers the buffer block B to the disk, and replaces the appropriate physical block there.

Example of Data Access



Each transaction T_i has its private work-area in which local copies of all data items accessed and updated by it are kept.

- T_i's local copy of a data item X is called x_i.

Transferring data items between system buffer blocks and its private work-area done by:

- **read**(X) assigns the value of data item X to the local variable x_i.
- **write**(X) assigns the value of local variable x_i to data item $\{X\}$ in the buffer block.
- **output**(B_X) need not immediately follow **write**(X). System can perform the **output** operation when it deems fit.

Transactions

- Must perform **read**(X) before accessing X for the first time (subsequent reads can be from local copy)
- **write**(X) can be executed at any time before the transaction commits

Recovery and Atomicity

To ensure atomicity despite failures, we first output information describing the modifications to stable storage without modifying the database itself.

Techniques used for recovery and atomicity

i) Log based recovery ii) Shadow paging(less used)

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 T_i starts, it registers itself by writing a
$<T_i$ **start**$>$log record

Where T_i executes **write**(X), a log record
$<T_i, X, V_1, V_2>$

where T_i – Transaction identifier
X – Data item identifier

Various types of log records are represented as, 93-

$\langle T_i \text{ start} \rangle$: Transaction T_i has started

$\langle T_i, X, V_1, V_2 \rangle$: Transaction T_i has performed a write on data item X. X had value V_1 (old) before the write and will have the value V_2 (new) after the write.

$\langle T_i \text{ Commit} \rangle$: Transaction T_i has committe

$\langle T_i \text{ abort} \rangle$: Transaction T_i has aborted

Problem: Consider two transactions T_1 and T_2 as given below. Write the system log detail and the corresponding database updation

T_1	T_2
read (A)	
A = A - 50;	
Write (A)	
read (B)	
B = B + 50;	
Write (B)	
	read (C)
	C = C - 100;
	Write (C)

Soln The given schedule is a serial schedule. Assume A, B, C have the values of 1000, 2000 and 700 respectively.

System log and database updation

Log	database
$\langle T_1 \text{ start} \rangle$	
$\langle T_1, A, 1000, 950 \rangle$	→ immediate
$\langle T_1, B, 2000, 2050 \rangle$	→
$\langle T_1 \text{ Commit} \rangle$	
	A = 950, B = 2050 } deferred
$\langle T_2 \text{ start} \rangle$	
$\langle T_2, C, 700, 600 \rangle$	
$\langle T_2 \text{ commit} \rangle$	C = 600

is written, where V_1 is the value of X before the write (the **old value**), and V_2 is the value to be written to X (the **new value**).

When T_i finishes it last statement, the log record $<T_i$ **commit**$>$ is written.

Two approaches using logs

- Immediate database modification: database modification occur while the transaction is still active
- Deferred database modification: → does not modify the database until it has committed.

mmediate database modification : This scheme allows pdates of an uncommitted transaction to be made to the buffer, r the disk itself, before the transaction commits

- Update log record must be written *before* database item is written. We assume that the log record is output directly to stable storage
- Output of updated blocks to stable storage 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.

eferred database modification : It performs updates to ffer/disk only at the time of transaction commit

- Simplifies some aspects of recovery
- But has overhead of storing local copy

ansaction Commit

- A transaction is said to have committed when its commit log record is output to stable storage. All previous log records of the transaction must have been output already.

Writes performed by a transaction may still be in the buffer when the transaction commits, and may be output later

Undo and Redo Operations

Undo of a log record $<T_i, X, V_1, V_2>$ writes the **old** value V_1 to X

Redo of a log record $<T_i, X, V_1, V_2>$ writes the **new** value V_2 to X

Undo and Redo of Transactions

undo(T_i) restores the value of all data items updated by T_i to their old values, going backwards from the last log record for T_i

- each time a data item X is restored to its old value V a special log record $<T_i, X, V>$ is written out
- when undo of a transaction is complete, a log record $<T_i$ **abort**$>$ is written out.

redo(T_i) sets the value of all data items updated by T_i to the new values, going forward from the first log record for T_i. No logging is done in this case

Undo and Redo on Recovering from Failure

When recovering after failure:

Transaction T_i needs to be undone if the log

- contains the record $<T_i$ **start**$>$,
- but does not contain either the record $<T_i$ **commit**$>$ *or* $<T_i$ **abort**$>$.

Transaction T_i needs to be redone if the log

- contains the records $<T_i$ **start**$>$
- and contains the record $<T_i$ **commit**$>$ or

- If transaction T_i was undone earlier and the $<T_i$ **abort**> record written to the log, and then a failure occurs, on recovery from failure T_i is redone
- **such a redo redoes all the original actions *including the steps that restored old values***
  - Known as **repeating history**
  - Seems wasteful, but simplifies recovery greatly

mmediate DB Modification Recovery Example

The log as it appears three instances of time.

$<T_0$ 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>
(a)	(b)	(c)

Recovery actions in each case above are:

(a) undo (T_0): B is restored to 2000 and A to 1000, and log records $<T_0$, B, 2000>, $<T_0$, A, 1000>, $<T_0$, **abort**> are written out

(b) redo (T_0) and undo (T_1): A and B are set to 950 and 2050 and C is restored to 700. Log records $<T_1$, C, 700>, $<T_1$, **abort**> are written out.

(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

- Redoing/undoing all transactions recorded in the log can be very slow
  1. processing the entire log is time-consuming if the system has run for a long time

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** L> onto stable storage where L is a list of all transactions active at the time of checkpoint.

- All updates are stopped while doing checkpointing

During recovery we need to consider only the most recent transaction T_i that started before the checkpoint, and transactions that started after T_i.

1. Scan backwards from end of log to find the most recent <**checkpoint** L> record
2. Only transactions that are in L or started after the checkpoint need to be redone or undone
3. Transactions that committed or aborted before the checkpoint already have all their updates output to stable storage.

Some earlier part of the log may be needed for undo operations

- Continue scanning backwards till a record <T_i **start**> is found for every transaction T_i in L.
- Parts of log prior to earliest <T_i **start**> record above are not needed for recovery, and can be erased whenever desired.

Recovery Algorithm

Logging (during normal operation):

<T_i **start**> at transaction start

<T_i, X_j, V_1, V_2> for each update, and

<T_i **commit**> at transaction end

Transaction rollback (during normal operation)

Let T_i be the transaction to be rolled back

Step 1: Scan log backwards from the end, and for each log record of T_i of the form <T_i, X_j, V_1, V_2>

- perform the undo by writing V_1 to X_j,
- write a log record <T_i, X_j, V_1>
  - such log records are called **compensation log records**

Step 2: Once the record <T_i **start**> is found stop the scan and write the log record <T_i **abort**>

Recovery from failure: Two phases

Redo phase: replay updates of **all** transactions, whether they committed, aborted, or are incomplete

Undo phase: undo all incomplete transactions

Redo phase:

Step 1: Find last <**checkpoint** L> record, and set undo-list to L. (T_0, T_1)

Step 2: Scan forward from above <**checkpoint** L> record

i) Whenever a record <T_i, X_j, V_1, V_2> is found, redo it by writing V_2 to X_j

ii) Whenever a log record <T_i **start**> is found, add T_i to undo-list → (T_0, T_1, T_2)

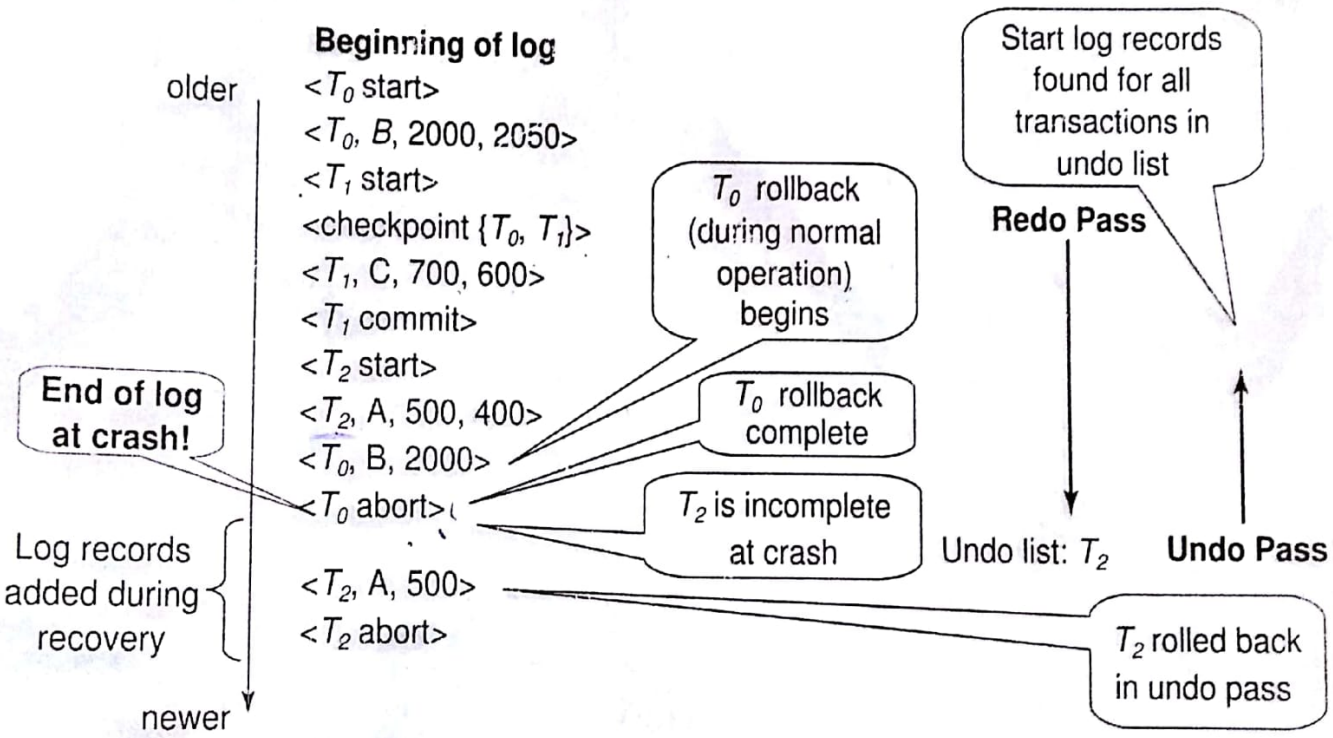
iii) Whenever a log record <T_i **commit**> *or* <T_i **abort**> is found, remove T_i from undo-list

Undo phase:

- Scan log backwards from end

1. Whenever a log record $<T_i, X_j, V_1, V_2>$ is found where T_i is in undo-list perform same actions as for transaction rollback:
   i) Perform undo by writing V_1 to X_j.
   ii) write a log record $<T_i, X_j, V_1>$
2. Whenever a log record $<T_i$ **start**$>$ is found where T_i is in undo-list,
   i) Write a log record $<T_i$ **abort**$>$
   ii) Remove T_i from undo-list
3. Stop when undo-list is empty i.e. $<T_i$ **start**$>$ has been found for every transaction in undo-list

After undo phase completes, normal transaction processing can commence



Log Record Buffering

Log record buffering: log records are buffered in main memory, instead of being output directly to stable storage.

Log records are output to stable storage when a block of log records in the buffer is full, or a **log force** operation is executed.

- Log force is performed to commit a transaction by forcing all its log records (including the commit record) to stable storage.
- Several log records can thus be output using a single output operation, reducing the I/O cost.
- The rules below must be followed if log records are buffered:
    - Log records are output to stable storage in the order in which they are created.
    - Transaction T_i enters the commit state only when the log record <T_i **commit**> has been output to stable storage.
    - Before a block of data in main memory is output to the database, all log records pertaining to data in that block must have been output to stable storage.
    - This rule is called the **write-ahead logging** or **WAL** rule -Strictly speaking WAL only requires undo information to be output

Database Buffering

- Database maintains an in-memory buffer of data blocks
    - When a new block is needed, if buffer is full an existing block needs to be removed from buffer
    - If the block chosen for removal has been updated, it must be output to disk
- The recovery algorithm supports the **no-force policy**: i.e., updated blocks need not be written to disk when transaction commits
    - **force policy**: requires updated blocks to be written at commit, More expensive commit

- The recovery algorithm supports the **steal policy**(i.e), blocks containing updates of uncommitted transactions can be written to disk, even before the transaction commits
- If a block with uncommitted updates is output to disk, log records with undo information for the updates are output to the log on stable storage first.(Write ahead logging)
- No updates should be in progress on a block when it is output to disk. Can be ensured as follows.
    - Before writing a data item, transaction acquires exclusive lock on block containing the data item
    - Lock can be released once the write is completed.
        - Such locks held for short duration are called **latches**.
- **To output a block to disk**
    a. First acquire an exclusive latch on the block
        i. Ensures no update can be in progress on the block
    b. Then perform a **log flush**
    c. Then output the block to disk
    d. Finally release the latch on the block
- **Database buffer can be implemented either**
    in an area of real main-memory reserved for the database, or in virtual memory

Implementing buffer in reserved main-memory has drawbacks:

- Memory is partitioned before-hand between database buffer and applications, limiting flexibility.
- Needs may change, and although operating system knows best how memory should be divided up at

any time, it cannot change the partitioning of memory.

- Database buffers are generally implemented in virtual memory in spite of some drawbacks:
  - ✓ When operating system needs to evict a page that has been modified, the page is written to swap space on disk.
  - ✓ When database decides to write buffer page to disk, buffer page may be in swap space, and may have to be read from swap space on disk and output to the database on disk, resulting in extra I/O. Known as **dual paging** problem.
  - ✓ Ideally when OS needs to evict a page from the buffer, it should pass control to database, which in turn should
    1. Output the page to database instead of to swap space (making sure to output log records first), if it is modified.
    2. Release the page from the buffer, for the OS to use.
  - ✓ Dual paging can thus be avoided, but common operating systems do not support such functionality.

Fuzzy Checkpointing

To avoid long interruption of normal processing during checkpointing, allow updates to happen during checkpointing

Fuzzy checkpointing is done as follows:

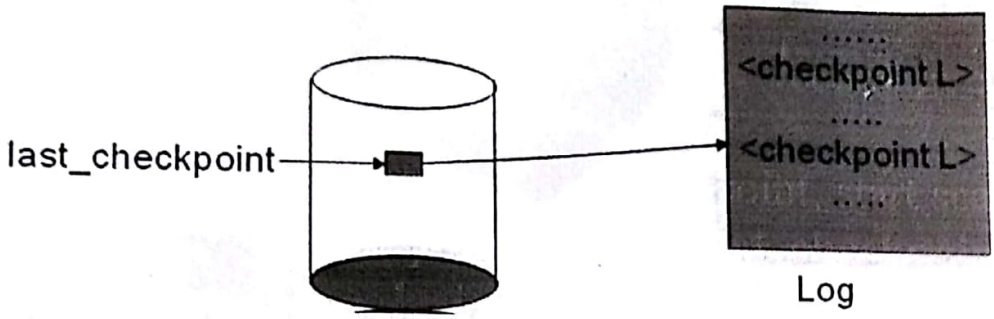
1. Temporarily stop all updates by transactions
2. Write a <**checkpoint** *L*> log record and force log to stable storage
3. Note list *M* of modified buffer blocks
4. Now permit transactions to proceed with their actions
5. Output to disk all modified buffer blocks in list *M*
   - blocks should not be updated while being output
   - Follow WAL: all log records pertaining to a block must be output before the block is output

6. Store a pointer to the **checkpoint** record in a fixed position **last_checkpoint** on disk

✓ When recovering using a fuzzy checkpoint, start scan from the **checkpoint** record pointed to by **last_checkpoint**

✓ Log records before **last_checkpoint** have their updates reflected in database on disk, and need not be redone.

✓ Incomplete checkpoints, where system had crashed while performing checkpoint, are handled safely



Failure with Loss of Nonvolatile Storage

➢ Technique similar to checkpointing used to deal with loss of non-volatile storage

➢ Periodically **dump** the entire content of the database to stable storage

➢ No transaction may be active during the dump procedure; a procedure similar to checkpointing must take place.

1) • Output all log records currently residing in main memory onto stable storage.

2) • Output all buffer blocks onto the disk.

3) • Copy the contents of the database to stable storage.

4) • Output a record <**dump**> to log on stable storage.

➢ To recover from disk failure

- restore database from most recent dump.

- Consult the log and redo all transactions that committed after the dump
- Can be extended to allow transactions to be active during dump; known as **fuzzy dump** or **online dump and** Similar to fuzzy checkpointing

Recovery with Early Lock Release and Logical Undo Operations

Recovery with Early Lock Release: Support for high-concurrency locking techniques, such as those used for B^+-tree concurrency control, which release locks early, and Supports "logical undo"

Recovery based on "repeating history", whereby recovery executes exactly the same actions as normal processing.

Logical Undo Logging: Operations like B^+-tree insertions and deletions release locks early. They cannot be undone by restoring old values (physical undo), since once a lock is released, other transactions may have updated the B^+-tree. Instead, insertions (resp. deletions) are undone by executing a deletion (resp. insertion) operation (known as logical undo).

For such operations, undo log records should contain the undo operation to be executed. Such logging is called logical undo logging, in contrast to physical undo logging and Operations are called as logical operations

Other examples:

- delete of tuple, to undo insert of tuple
  - allows early lock release on space allocation information
- subtract amount deposited, to undo deposit
  - allows early lock release on bank balance

Physical Redo: Redo information is logged physically (that is, new value for each write) even for operations with logical undo. Logical redo is very complicated since database state on disk may not be "operation consistent" when recovery starts. Physical redo logging does not conflict with early lock release.

Operation Logging:

Operation logging is done as follows:

1. When operation starts, log $<T_i, O_j,$ **operation-begin**$>$. Here O_j is a unique identifier of the operation instance.
2. While operation is executing, normal log records with physical redo and physical undo information are logged.
3. When operation completes, $<T_i, O_j,$ **operation-end**, $U>$ is logged, where U contains information needed to perform a logical undo information.

If crash/rollback occurs before operation completes:

1. the **operation-end** log record is not found, and
2. the physical undo information is used to undo operation.

If crash/rollback occurs after the operation completes:

- the **operation-end** log record is found, and in this case
- logical undo is performed using U; the physical undo information for the operation is ignored.

Redo of operation (after crash) still uses physical redo information.

Transaction Rollback with Logical Undo

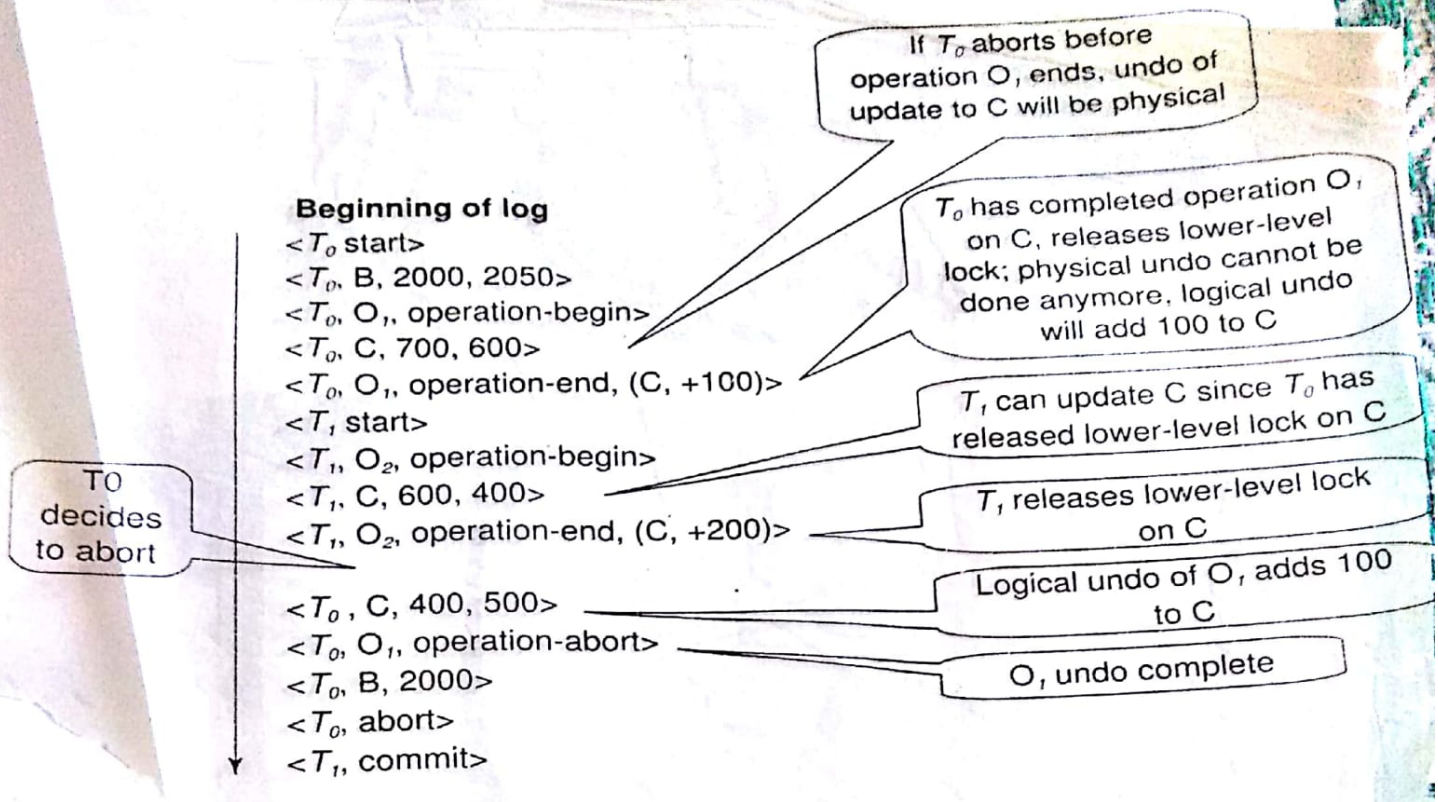
Rollback of transaction T_i is done as follows:

Scan the log backwards

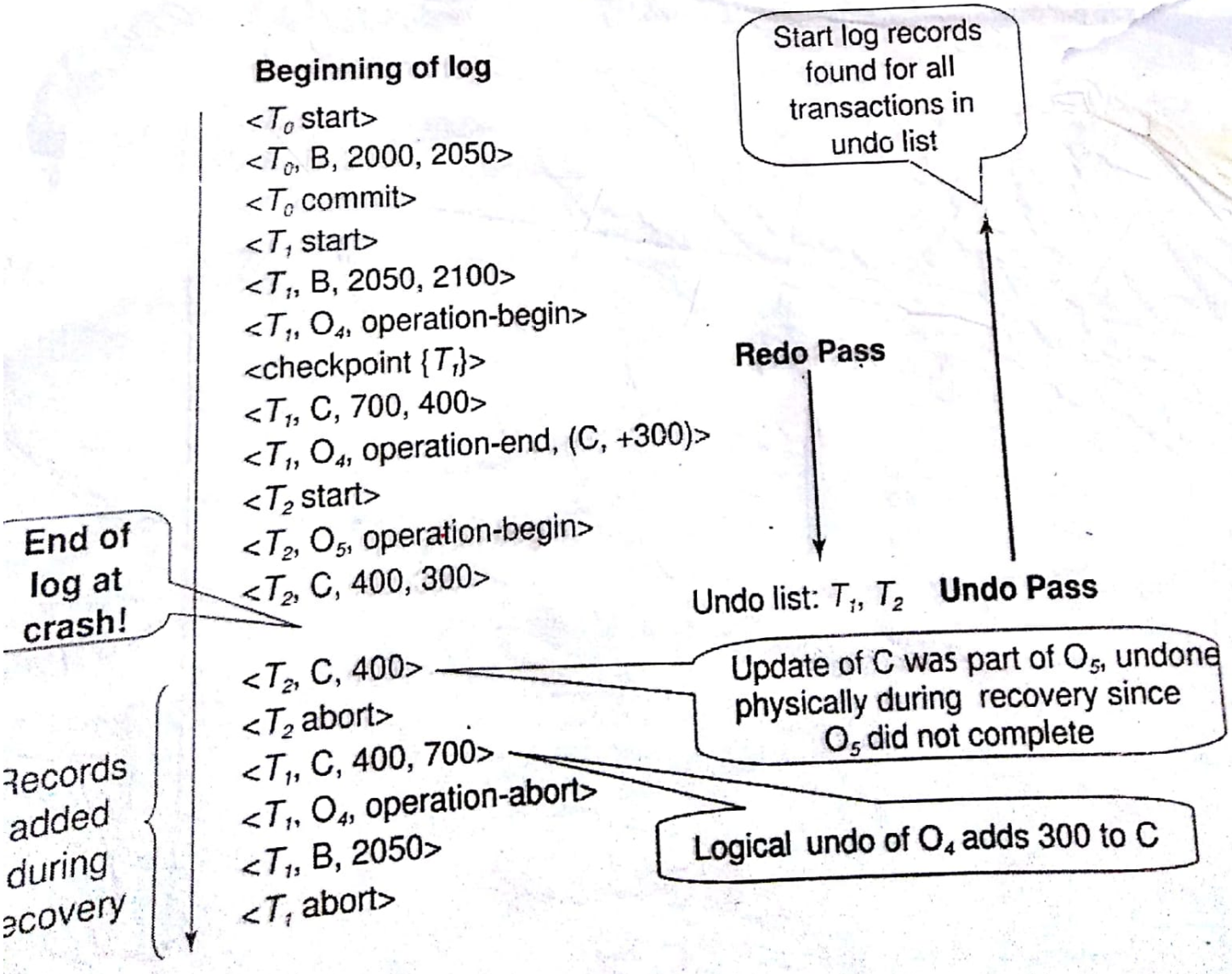
1. If a log record $<T_i, X, V_1, V_2>$ is found, perform the undo and log a al $<T_i, X, V_1>$.
2. If a $<T_i, O_j,$ **operation-end**, $U>$ record is found
   - Rollback the operation logically using the undo information U.
     - Updates performed during roll back are logged just like during normal operation execution.
     - At the end of the operation rollback, instead of logging an **operation-end** record, generate a record $<T_i, O_j,$ **operation-abort**$>$.
   - Skip all preceding log records for T_i until the record $<T_i, O_j$ **operation-begin**$>$ is found
3. If a redo-only record is found ignore it
4. If a $<T_i, O_j,$ **operation-abort**$>$ record is found:
   - skip all preceding log records for T_i until the record $<T_i, O_j,$ **operation-begin**$>$ is found.
5. Stop the scan when the record $<T_i,$ start$>$ is found
6. Add a $<T_i,$ **abort**$>$ record to the log

Cases 3 and 4 above can occur only if the database crashes while a transaction is being rolled back. Skipping of log records as in case 4 is important to prevent multiple rollback of the same operation.

Transaction Rollback with Logical Undo



Failure Recovery with Logical Undo



Recovery Algorithm with Logical Undo

1. (**Redo phase**): Scan log forward from last < **checkpoint** L> record till end of log
    1. **Repeat history** by physically redoing all updates of all transactions,
    2. Create an undo-list during the scan as follows
        - *undo-list* is set to L initially
        - Whenever <T_i **start**> is found T_i is added to *undo-list*
        - Whenever <T_i **commit**> or <T_i **abort**> is found, T_i is deleted from *undo-list*

This brings database to state as of crash, with committed as well as uncommitted transactions having been redone.

Now *undo-list* contains transactions that are **incomplete**, that is, have neither committed nor been fully rolled back.

2. (**Undo phase**): Scan log backwards, performing undo on log records of transactions found in *undo-list*.

    Log records of transactions being rolled back are processed as described earlier, as they are found

    - Single shared scan for all transactions being undone

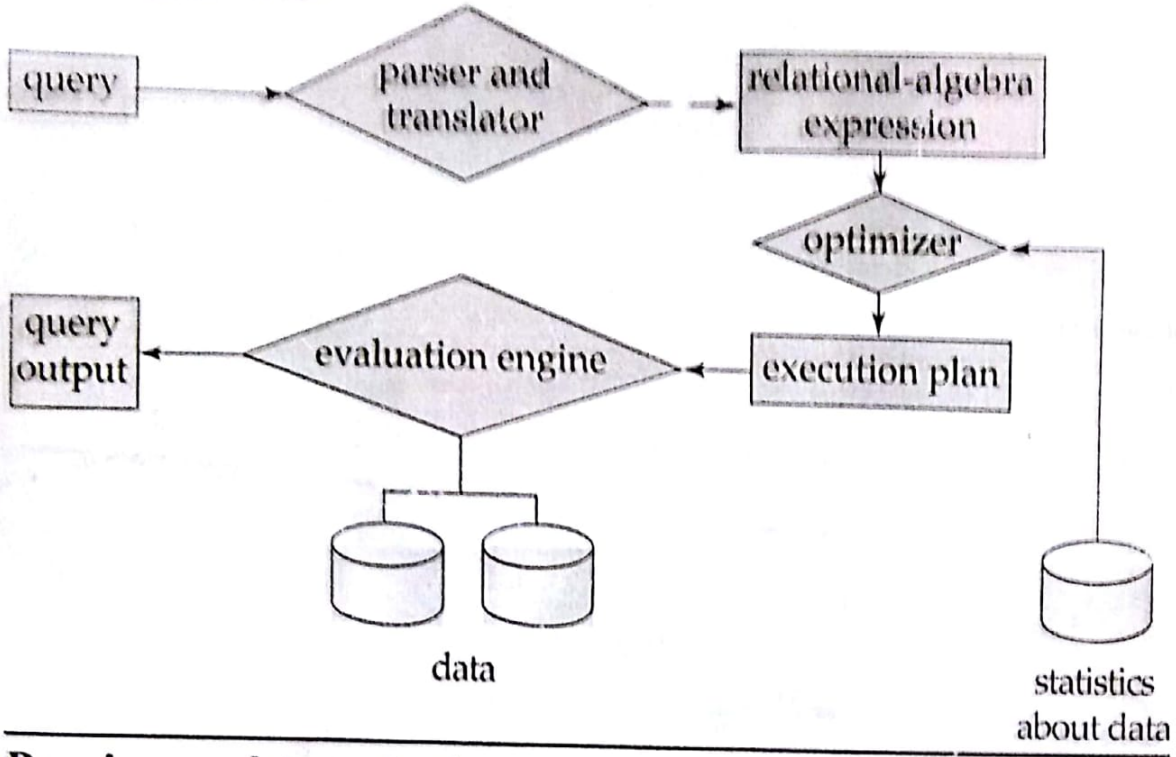
    When <T_i **start**> is found for a transaction T_i in *undo-list*, write a <T_i **abort**> log record.

    Stop scan when <T_i **start**> records have been found for all T_i in *undo-list*

This undoes the effects of incomplete transactions (those with neither **commit** nor **abort** log records). Recovery is now complete.

Basic Steps in Query Processing

1. Parsing and translation
2. Optimization
3. Evaluation



Parsing and translation

Translate the query into its internal form. This is then translated into relational algebra. Parser checks syntax, verifies relations

Evaluation

The query-execution engine takes a query-evaluation plan, executes that plan, and returns the answers to the query.

Optimization

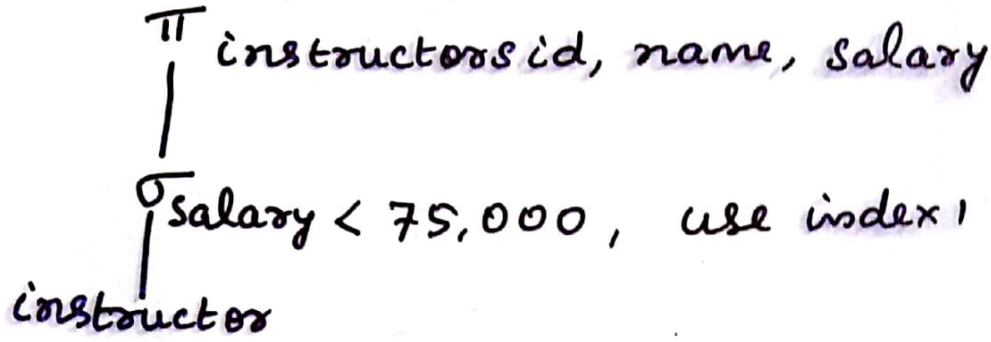
- A relational algebra expression may have many equivalent expressions

E.g., $\sigma_{salary<75000}(\Pi_{salary}(instructor))$ is equivalent to $\Pi_{salary}(\sigma_{salary<75000}(instructor))$

- Each relational algebra operation can be evaluated using one of several different algorithms; correspondingly, a relational-algebra expression can be evaluated in many ways.

Annotated expression specifying detailed evaluation strategy is called an **evaluation-plan**.

E.g., can use an index on *salary* to find instructors with salary < 75000,
or can perform complete relation scan and discard instructors with salary ≥ 75000



Query Optimization: Amongst all equivalent evaluation plans choose the one with lowest cost.
Cost is estimated using statistical information from the database catalog

- e.g. number of tuples in each relation, size of tuples, etc.

Measures of Query Cost

- Cost is generally measured as total elapsed time for answering query. Many factors contribute to time cost like *disk accesses, CPU*, or even network *communication*
- Typically disk access is the predominant cost, and is also relatively easy to estimate. Measured by taking into account

  Number of seeks * average-seek-cost
  Number of blocks read * average-block-read-cost
  Number of blocks written * average-block-write-cost

- Cost to write a block is greater than cost to read a block. Also data is read back after being written to ensure that the write was successful
- For simplicity use the **number of block transfers** *from disk and the* **number of seeks** as the cost measures

Seek – time to access the first block of the file. If blocks are not stored contiguously, extra seeks may be required.