DBMS CHAP 6

Transaction, Recovery and Concurrency Control

6	**Transaction, Recovery and Concurrency Control** **Transaction Management:** Transaction Concept, Transaction States, ACID Properties of Transaction, Serial and Concurrent Executions, Conflict and View Serializability. **Concurrency Control:** Lock Based Protocols, Deadlock Handling **Recovery:** Failure Classification, Log based recovery, Checkpoint, Shadow Paging.	9	L1, L2

Transaction Concept

- A **transaction** is a *unit* of program execution that accesses and possibly updates various data items.
- E.g., transaction to transfer Rs50 from account A to account B:
  1. **read**(*A*)
  2. *A* := *A* – 50
  3. **write**(*A*)
  4. **read**(*B*)
  5. *B* := *B* + 50
  6. **write**(*B*)
- Two main issues to deal with:
  - Failures of various kinds, such as hardware failures and system crashes
  - Concurrent execution of multiple transactions

ACID Properties

A **transaction** is a unit of program execution that accesses and possibly updates various data items. To preserve the integrity of data the database system must ensure:

- **Atomicity.** Either all operations of the transaction are properly reflected in the database or none are.
- Example: Airline Reservation:

Check seat availability, airline confirms seat, reduces no of available seats, charges credit card, increases no of meals.

So here either all changes should be reflected successfully or not.

Example

- Transaction to transfer 50 from account A to account B:
    1. **read**(*A*)
    2. *A* := *A* – 50
    3. **write**(*A*)
    4. **read**(*B*)
    5. *B* := *B* + 50
    6. **write**(*B)*

- **Atomicity requirement**
    - If the transaction fails after step 3 and before step 6, money will be “lost” leading to an inconsistent database state
        - Failure could be due to software or hardware
    - The system should ensure that updates of a partially executed transaction are not reflected in the database

ACID Properties

A **transaction** is a unit of program execution that accesses and possibly updates various data items. To preserve the integrity of data the database system must ensure:

- **Consistency.** Execution of a transaction in isolation preserves the consistency of the database.
- Example: Balance in ACC A is 1000 and B is 5000, so sum = 6000. If 500 is deducted from A then A is 500 so 5500 total but before saving A failure occurs the proper reflection is not done.

Example

- **Consistency requirement** in above example:
    - The sum of A and B is unchanged by the execution of the transaction

- In general, consistency requirements include
    - Explicitly specified integrity constraints such as primary keys and foreign keys
    - Implicit integrity constraints
        - e.g., sum of balances of all accounts, minus sum of loan amounts must equal value of cash-in-hand
    - A transaction must see a consistent database.
    - During transaction execution the database may be temporarily inconsistent.
    - When the transaction completes successfully the database must be consistent
        - Erroneous transaction logic can lead to inconsistency

ACID Properties

A **transaction** is a unit of program execution that accesses and possibly updates various data items. To preserve the integrity of data the database system must ensure:

- **Isolation.** Although multiple transactions may execute concurrently, each transaction must be unaware of other concurrently executing transactions. Intermediate transaction results must be hidden from other concurrently executed transactions.
  - That is, for every pair of transactions T_i and T_j, it appears to T_i that either T_j, finished execution before T_i started, or T_j started execution after T_i finished.

Example

- **Isolation requirement** — if between steps 3 and 6, another transaction T2 is allowed to access the partially updated database, it will see an inconsistent database (the sum *A* + *B* will be less than it should be).

T1	**T2**
1. **read**(*A*)	
2. *A* := *A* – 50	
3. **write**(*A*)	
	read(A), read(B), print(A+B)
4. **read**(*B*)	
5. *B* := *B* + 50	
6. **write**(*B*	

- Isolation can be ensured trivially by running transactions **serially**
  - That is, one after the other.

ACID Properties

A **transaction** is a unit of program execution that accesses and possibly updates various data items. To preserve the integrity of data the database system must ensure:

- **Durability.** After a transaction completes successfully, the changes it has made to the database persist (permanent), even if there are system failures.

Example

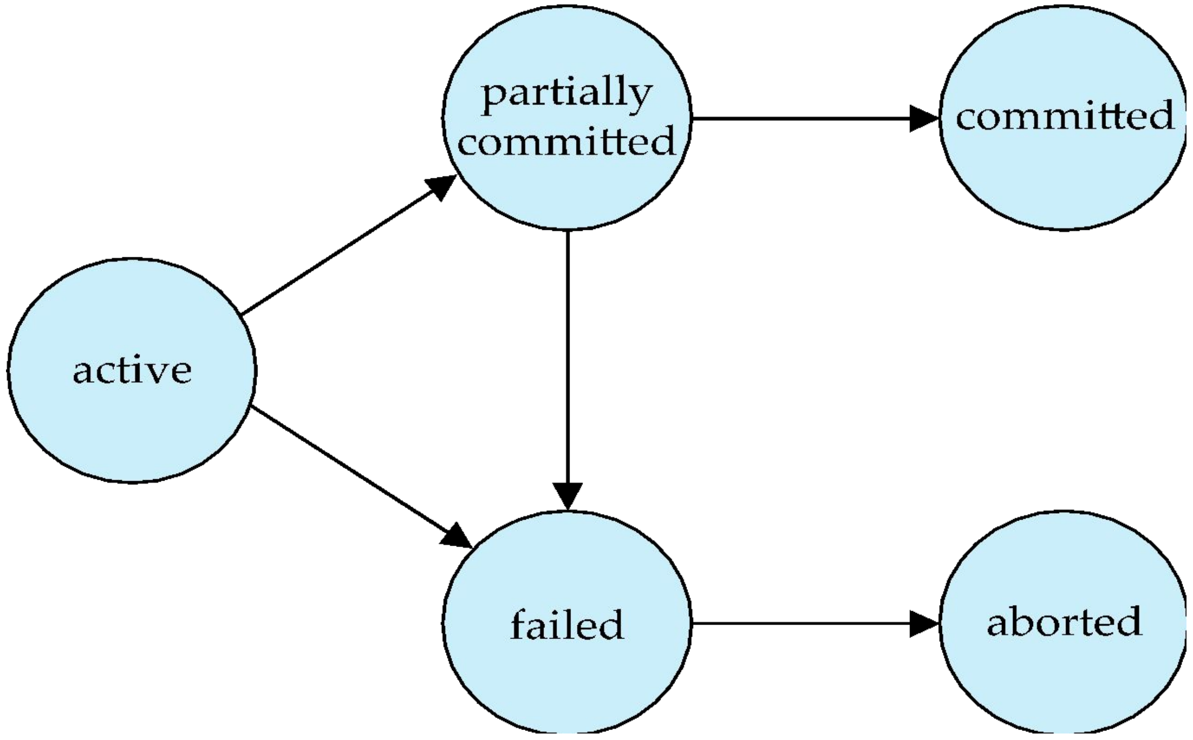
- Transaction to transfer 50 from account A to account B:
  1. **read**(*A*)
  2. *A* := *A* – 50
  3. **write**(*A*)
  4. **read**(*B*)
  5. *B* := *B* + 50
  6. **write**(*B*)

- **Durability requirement** — once the user has been notified that the transaction has completed (i.e., the transfer of the 50 has taken place), the updates to the database by the transaction must persist even if there are software or hardware failures.

Transaction State

- **Active** – the initial state; the transaction stays in this state while it is executing
- **Partially committed** – after the final statement has been executed.
- **Failed** -- after the discovery that normal execution can no longer proceed.
- **Aborted** – after the transaction has been rolled back and the database restored to its state prior to the start of the transaction. Two options after it has been aborted:
  - Restart the transaction
    - Can be done only if no internal logical error
  - Kill the transaction
- **Committed** – after successful completion.

Transaction State (Cont.)



Concurrent Executions

- Multiple transactions are allowed to run concurrently in the system. Advantages are:
  - **Increased processor and disk utilization**, leading to better transaction *throughput*
    - E.g., one transaction can be using the CPU while another is reading from or writing to the disk
  - **Reduced average response time** for transactions: short transactions need not wait behind long ones.
- **Concurrency control schemes** – mechanisms to achieve isolation
  - That is, to control the interaction among the concurrent transactions in order to prevent them from destroying the consistency of the database
    - Will study in Chapter 15, after studying notion of correctness of concurrent executions.

Schedules

- **Schedule** – a sequences of instructions that specify the chronological order in which instructions of concurrent transactions are executed
  - A schedule for a set of transactions must consist of all instructions of those transactions
  - Must preserve the order in which the instructions appear in each individual transaction.

- A transaction that successfully completes its execution will have a commit instructions as the last statement
  - By default transaction assumed to execute commit instruction as its last step

- A transaction that fails to successfully complete its execution will have an abort instruction as the last statement

Schedule 1

- Let T_1 transfer \$50 from *A* to *B*, and T_2 transfer 10% of the balance from *A* to *B.*
- A serial schedule in which T_1 is followed by T_2 :

T_1	T_2
read (A)	
$A := A - 50$	
write (A)	
read (B)	
$B := B + 50$	
write (B)	
commit	
	read (A)
	$temp := A * 0.1$
	$A := A - temp$
	write (A)
	read (B)
	$B := B + temp$
	write (B)
	commit

Schedule 2

- A serial schedule where T_2 is followed by T_1

T_1	T_2
	read (A) $temp := A * 0.1$ $A := A - temp$ write (A) read (B) $B := B + temp$ write (B) commit
read (A) $A := A - 50$ write (A) read (B) $B := B + 50$ write (B) commit	

Schedule 3

- Let T_1 and T_2 be the transactions defined previously. The following schedule is not a serial schedule, but it is *equivalent* to Schedule 1

T_1	T_2
read (A) $A := A - 50$ write (A)	
	read (A) $temp := A * 0.1$ $A := A - temp$ write (A)
read (B) $B := B + 50$ write (B) commit	
	read (B) $B := B + temp$ write (B) commit

- In Schedules 1, 2 and 3, the sum A + B is preserved.

Schedule 4

- The following concurrent schedule does not preserve the value of (*A* + *B*).

T_1	T_2
read (*A*)	
A := *A* – 50	
	read (*A*)
	temp := *A* * 0.1
	A := *A* - *temp*
	write (*A*)
	read (*B*)
write (*A*)	
read (*B*)	
B := *B* + 50	
write (*B*)	
commit	
	B := *B* + *temp*
	write (*B*)
	commit

Serializability

- **Basic Assumption** – Each transaction preserves database consistency.
- Thus, serial execution of a set of transactions preserves database consistency.
- A (possibly concurrent) schedule is serializable if it is equivalent to a serial schedule. Different forms of schedule equivalence give rise to the notions of:
  1. **Conflict serializability**
  2. **View serializability**

Conflicting Instructions

- Instructions l_i and l_j of transactions T_i and T_j respectively, **conflict** if and only if there exists some item *Q* accessed by both l_i and l_j, and at least one of these instructions wrote *Q*.
    1. l_i = **read**(*Q*), l_j = **read**(*Q*). l_i and l_j don't conflict.
    2. l_i = **read**(*Q*), l_j = **write**(*Q*). They conflict.
    3. l_i = **write**(*Q*), l_j = **read**(*Q*). They conflict
    4. l_i = **write**(*Q*), l_j = **write**(*Q*). They conflict
- Intuitively, a conflict between l_i and l_j forces a (logical) temporal order between them.
- If l_i and l_j are consecutive in a schedule and they do not conflict, their results would remain the same even if they had been interchanged in the schedule.

Conflict Serializability

- If a schedule *S* can be transformed into a schedule *S'* by a series of swaps of non-conflicting instructions, we say that *S* and *S'* are **conflict equivalent***.*

- We say that a schedule *S* is **conflict serializable** if it is conflict equivalent to a serial schedule

Conflict Serializability (Cont.)

- Schedule 3 can be transformed into Schedule 6, a serial schedule where T_2 follows T_1, by series of swaps of non-conflicting instructions. Therefore Schedule 3 is conflict serializable.

T_1	T_2
read (A)	
write (A)	
	read (A)
	write (A)
read (B)	
write (B)	
	read (B)
	write (B)

Schedule 3

T_1	T_2
read (A)	
write (A)	
read (B)	
write (B)	
	read (A)
	write (A)
	read (B)
	write (B)

Schedule 6

Conflict Serializability (Cont.)

- Example of a schedule that is not conflict serializable:

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

- We are unable to swap instructions in the above schedule to obtain either the serial schedule < T_3, T_4 >, or the serial schedule < T_4, T_3 >.

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'*.
- As can be seen, view equivalence is also based purely on **reads** and **writes** alone.

View Serializability (Cont.)

- 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.

T_{27}	T_{28}	T_{29}
read (Q)		
	write (Q)	
write (Q)		
		write (Q)

- What serial schedule is above equivalent to?
- Every view serializable schedule that is not conflict serializable has **blind writes**.

Lock-Based Protocols

- 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 the concurrency-control manager by the programmer. Transaction can proceed only after request is granted.

Lock-Based Protocols (Cont.)

- **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.

Lock-Based Protocols (Cont.)

- 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*)

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

The Two-Phase Locking Protocol

- This protocol ensures conflict-serializable schedules.
- Phase 1: Growing Phase
  - Transaction may obtain locks
  - Transaction may not release locks
- Phase 2: Shrinking Phase
  - Transaction may release locks
  - Transaction may not obtain locks
- The protocol assures serializability. It can be proved that the transactions can be serialized in the order of their **lock points** (i.e., the point where a transaction acquired its final lock).

The Two-Phase Locking Protocol (Cont.)

- 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.

Lock Conversions

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

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

Automatic Acquisition of Locks (Cont.)

- **write**(*D)* is processed as:

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

- All locks are released after commit or abort

Deadlocks

- 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-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.

Deadlocks (Cont.)

- Two-phase locking *does not* ensure freedom from deadlocks.
- In addition to deadlocks**,** there is a possibility of **starvation.**
- **Starvation** occurs if the concurrency control manager is badly designed. For example:
  - A transaction may be waiting for an X-lock on an item, while a sequence of other transactions request and are granted an S-lock on the same item.
  - The same transaction is repeatedly rolled back due to deadlocks.
- Concurrency control manager can be designed to prevent starvation.

Deadlocks (Cont.)

- The potential for deadlock exists in most locking protocols. Deadlocks are a necessary evil.
- When a deadlock occurs there is a possibility of cascading roll-backs.
- Cascading roll-back is possible under two-phase locking. To avoid this, follow a modified protocol called **strict two-phase locking** -- a transaction must hold all its exclusive locks till it commits/aborts.
- **Rigorous two-phase locking** is even stricter. Here, *all* locks are held till commit/abort. In this protocol transactions can be serialized in the order in which they commit.

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

Deadlock Handling

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

More Deadlock Prevention Strategies

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

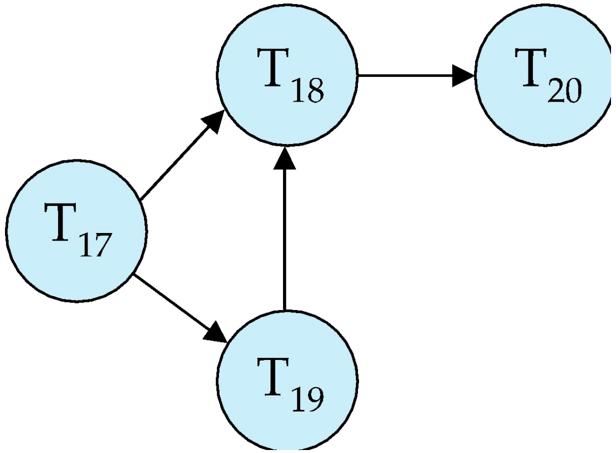
Deadlock prevention (Cont.)

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

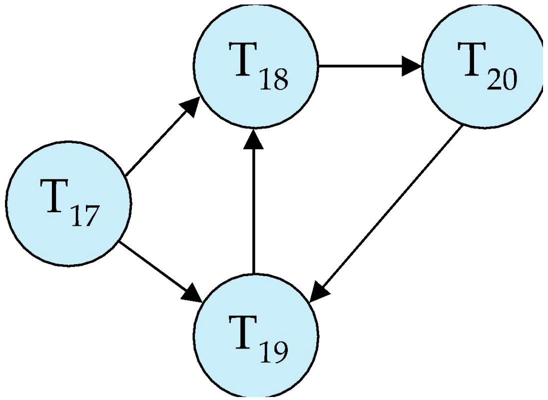
Deadlock Detection

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

Deadlock Detection (Cont.)



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

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.

Timestamp-Based Protocols (Cont.)

- 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)).

Timestamp-Based Protocols (Cont.)

- Suppose that transaction T_i issues **write**(Q).
  1. 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.
  2. 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.
  3. Otherwise, the **write** operation is executed, and W-timestamp(Q) is set to TS(T_i).

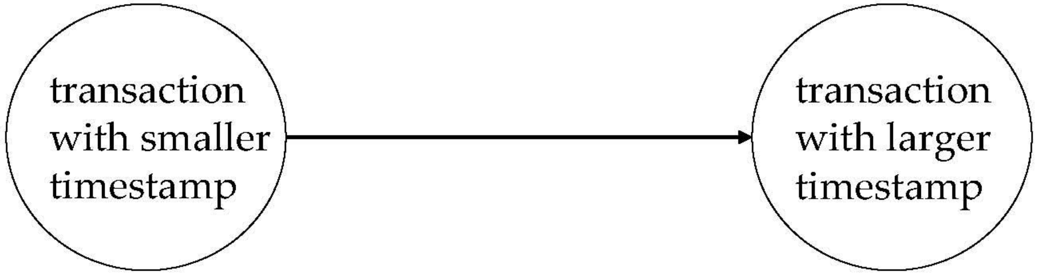
Example Use of the Protocol

A partial schedule for several data items for transactions with timestamps 1, 2, 3, 4, 5

T_1	T_2	T_3	T_4	T_5
				read (X)
	read (Y)			
read (Y)				
		write (Y)		
		write (Z)		
				read (Z)
	read (Z)			
	abort			
read (X)				
			read (W)	
		write (W)		
		abort		
				write (Y)
				write (Z)

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

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.

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
    - I.e., 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

Validation-Based Protocol (Cont.)

- Each transaction T_i has 3 timestamps
  - Start(T_i) : the time when T_i started its execution
  - Validation(T_i): the time when T_i entered its validation phase
  - Finish(T_i) : the time when T_i finished its write phase
- Serializability order is determined by timestamp given at validation time; this is done 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)
  - **start**(T_j) < **finish**(T_i) < **validation**(T_j) **and** the set of data items written by T_i does not intersect with the set of data items read by T_j.

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

Failures

1. **Hardware Failure / System crash**
   - There is a hardware malfunction that causes the loss of the content of volatile storage, and brings transaction processing to a halt.
   - The content of non volatile storage remains intact, and is not corrupted or changed.
2. **Software Failure**

   The database software or the operating system may be corrupted or failed to work correctly, that may causes the loss of the content of volatile storage, and brings about database failure.
3. **Media failure**
   - A disk block loses its content as a result of either a head crash or failure during a data-transfer operation.
   - Copies of the data on other disks such as tapes, CDs are used to recover from the failure.
4. **Network Failure**
   - The problem with network interface card can cause network failure.
   - There may be problem with network connection.

Failures

5. **Transaction failure**

There are two types of errors that may cause a transaction to fail :

(a) **Logical error**

The transaction can no longer continue with its normal execution because of some internal condition, such as wrong input values, data not found in database, data overflow, or resource limit exceeded etc.

(b) **System error**

The system has entered an undesirable state like deadlock; as a result transaction cannot continue with its normal execution.

6. **Application software error**

- The problem with software accessing the data from database.
- This may cause database failure as data cannot be updated using such application to it.

7. **Physical disasters**

The problem caused due to flood, fire, earthquake etc.

8. **Software error**

These are some logical errors in the program that is accessing database, which cause one or more transactions failure.

Checkpoint

- A database checkpoint is where all committed transactions are written to the redo/audit logs.
- The database administrator determines the frequency of the checkpoints based on volume of transactions.
- When a system failure occurs, we must consult the log to determine those transactions that need to be redone and those that need to be undone using above log files.
- Too frequent checkpoints can affect the performance.

Checkpoint

2. **Problems in this approach**

- The search process is time-consuming.
- Most of the transactions that, according to our algorithm, need to be redone as they have already written their updates into the database.

3. **Need of Checkpoints**

- To reduce these types of overhead, we introduce checkpoints.
- During execution, the system maintains the log, using one of the two techniques.
- The system periodically performs checkpoints, with following sequence of actions :
  1. Output all log records onto stable storage which are currently stored in main memory.
  2. Output to the disk all modified buffer blocks.
  3. Output onto stable storage a log record <checkpoint>.
- Transactions are not allowed to perform any update actions, such as writing to a buffer block or writing a log record, while a checkpoint is in working state.
- The presence of a <checkpoint> record in the log allows the system to restructure its recovery procedure.

Checkpoint

4. **Working**

- Consider a transaction T_n that committed prior to the checkpoint.
- For such a transaction, the <T_n commit> record appears in the log before the <checkpoint> record.
- Any database modifications made by transaction T_n must have been written to the database either prior to the checkpoint or as part of the checkpoint itself. Thus, at recovery time, there is no need to perform a redo operation on T_n.
- After a database failure has occurred, the recovery scheme examines the log to determine the most recent transaction T_n that started executing before the most recent checkpoint took place.
- It can find such a transaction by searching the log backward, from the end of the log, until it finds the first <checkpoint> record (since we are searching backward, the record found is the final <checkpoint> record in the log); then it continues the search backward until it finds the next <T_n start> record. This record identifies a transaction T_n.
- Once the system has identified respective transaction T_n, the redo and undo operations need to be applied to only for transaction T_n and all transactions that started executing after transaction T_n.

Log Based Recovery

- There can be problem in accessing database due to any reason can causes a database system failure.
- The most widely used structure for recording database modifications is **Transaction Log (Log).**
- The log is a sequence of log records, recording all the update activities done on the database by all database users.

Log Based Recovery

There are several types of log records.

a. **Update Log Record**

- An update log record describes a single database write.
- It also includes the value of the bytes of page before and after the page change.

b. **Compensation Log Record**

- Compensation log record the rollback of a particular change to the database.
- Each corresponds with exactly one other Update Log Record.

c. **Commit Record**

Records a decision to commit a transaction.

d. **Abort Record**

Records a decision to abort and hence rollback a transaction.

e. **Checkpoint Record**

- Records a point when checkpoint has been made.
- These are used to speed up recovery.
- It also record information that eliminates the need to read a log's past.
- The time of recording varies according to checkpoint algorithm.

Shadow Paging

- It is not always convenient to maintain logs of all transactions for the purposes of recovery. An alternative is to use a system of shadow paging.
- This is where the database is divided into pages that may be stored in any order on the disk.
- In order to identify the location of any given page, we use something called a **page table**.

Shadow Paging

- This method considers the database is made up of a number of fixed size pages.
- A directory with n entries is constructed which the entry points to the database page on disk.
- This directory is kept in the main memory.
- It is not too large and all references read and write to database pages on disk.

Shadow Paging

a. During the life of a transaction two page tables are maintained as below,

   i. Shadow page table   ii. Current page table.

b. When a transaction begins both of these page tables point to the same locations (are identical).

c. During the lifetime of a transaction the shadow page table doesn't change at all.

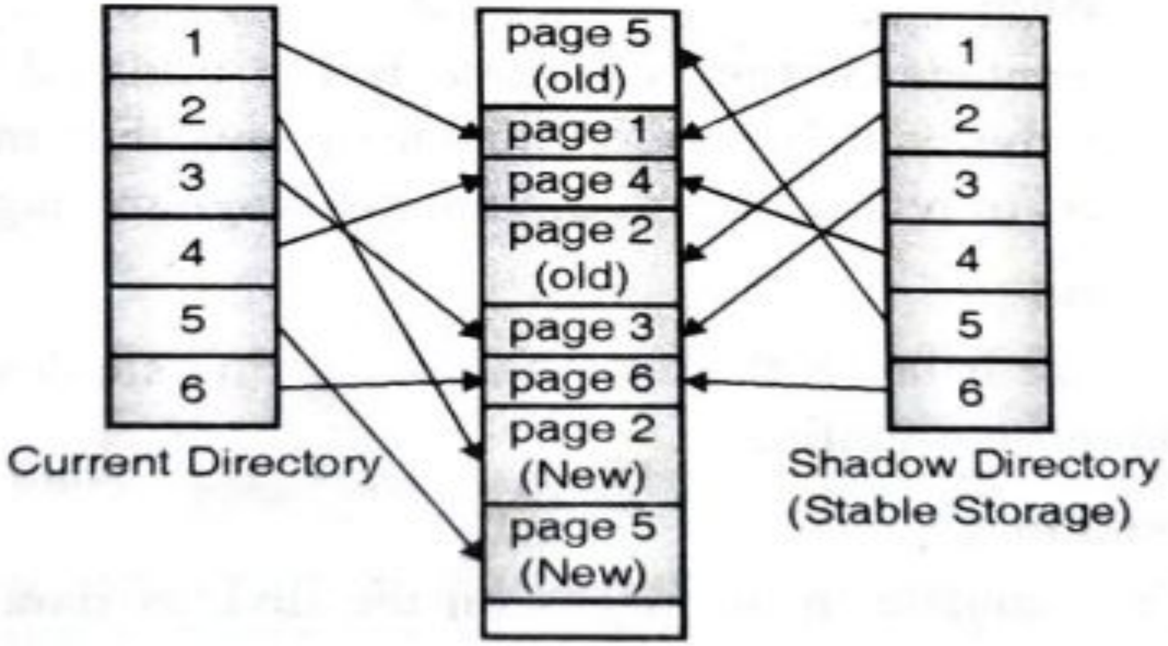
d. However during the lifetime of a transaction update values etc. may be changed.

e. For pages updated by the transaction, two versions are kept. The old version is referenced by the shadow directory and the new version by the current directory.

f. So whenever we update a page in the database we always write the updated page to a new location.

g. This means that when we update our current page table it reflect the changes that have been made by that transaction.

Shadow Paging



h. Looking at below diagram we see how these tables appear during a transaction. As we can see the shadow page table shows the state of the database just prior to a transaction, and the current page table shows the state of the database during or after a transaction has been completed.