Unit III – DATABASE STORAGE, PROCESSING AND TRANSACTION

Prof. Sonali Deshpande

- Query processing and query optimization, Basic concept of a Transaction, Transaction Management, ACID Properties of Transactions,
- Concept of Schedule, Serial and Concurrent Schedule, Serializability: Conflict and View, Cascaded Aborts, Recoverable and Non-recoverable Schedules,
- Concurrency Control: Need, Locking based Protocol, Deadlocks-Prevention, Detection Techniques, Recovery methods : Shadow Paging and Log Based Recovery, Checkpoints, Introduction to RDD, RDD operations.

Query processing and query optimization

- A query is processed in three general steps:
  - Parsing and Translation
  - Query Optimization or planning the execution strategy
  - Execution in the runtime database processor

Query Optimization or Planning the Execution Strategy

- For any given query, there may be a number of different ways to execute it.
- Each operation in the query (SELECT, JOIN, etc.) can be implemented using one or more different *Access Routines*.
- For example, an access routine that employs an index to retrieve some rows would be more efficient that an access routine that performs a full table scan.
- The goal of the **query optimizer** is to find a *reasonably efficient* strategy for executing the query (not quite what the name implies) using the access routines.
- Optimization typically takes one of two forms: *Heuristic Optimization* or *Cost Based Optimization*
- In **Heuristic Optimization**, the query execution is refined based on *heuristic rules* for reordering the individual operations.
- With **Cost Based Optimization**, the overall cost of executing the query is systematically reduced by estimating the costs of executing several different execution plans.

- **Query Code Generator (interpreted or compiled)**
- Once the query optimizer has determined the execution plan (the specific ordering of access routines), the code generator writes out the actual access routines to be executed.
- With an interactive session, the query code is interpreted and passed directly to the runtime database processor for execution.
- It is also possible to *compile* the access routines and store them for later execution.
- **Execution in the runtime database processor**
- At this point, the query has been scanned, parsed, planned and (possibly) compiled.
- The runtime database processor then executes the access routines against the database.
- The results are returned to the application that made the query in the first place.
- Any runtime errors are also returned.

Transaction Concept

A **transaction** is a *unit* of program execution that accesses and possibly updates various data items.

E.g. 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*)

Two main issues to deal with:

- **Failures** of various kinds, such as hardware failures and system crashes
- **Concurrent** execution of multiple transactions

Example of Fund Transfer

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**

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.

Example of Fund Transfer

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*)

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., min**sum of balances of all accounts**us 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

Example of Fund Transfer

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.

However, executing multiple transactions concurrently has **significant benefits**, as we will see later.

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.
- **Consistency.**
  - Execution of a transaction in isolation preserves the consistency of the database.
- **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.
- **Durability.**
  - After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.

Transaction State

Active – the initial state; the transaction stays in this state while it is executing

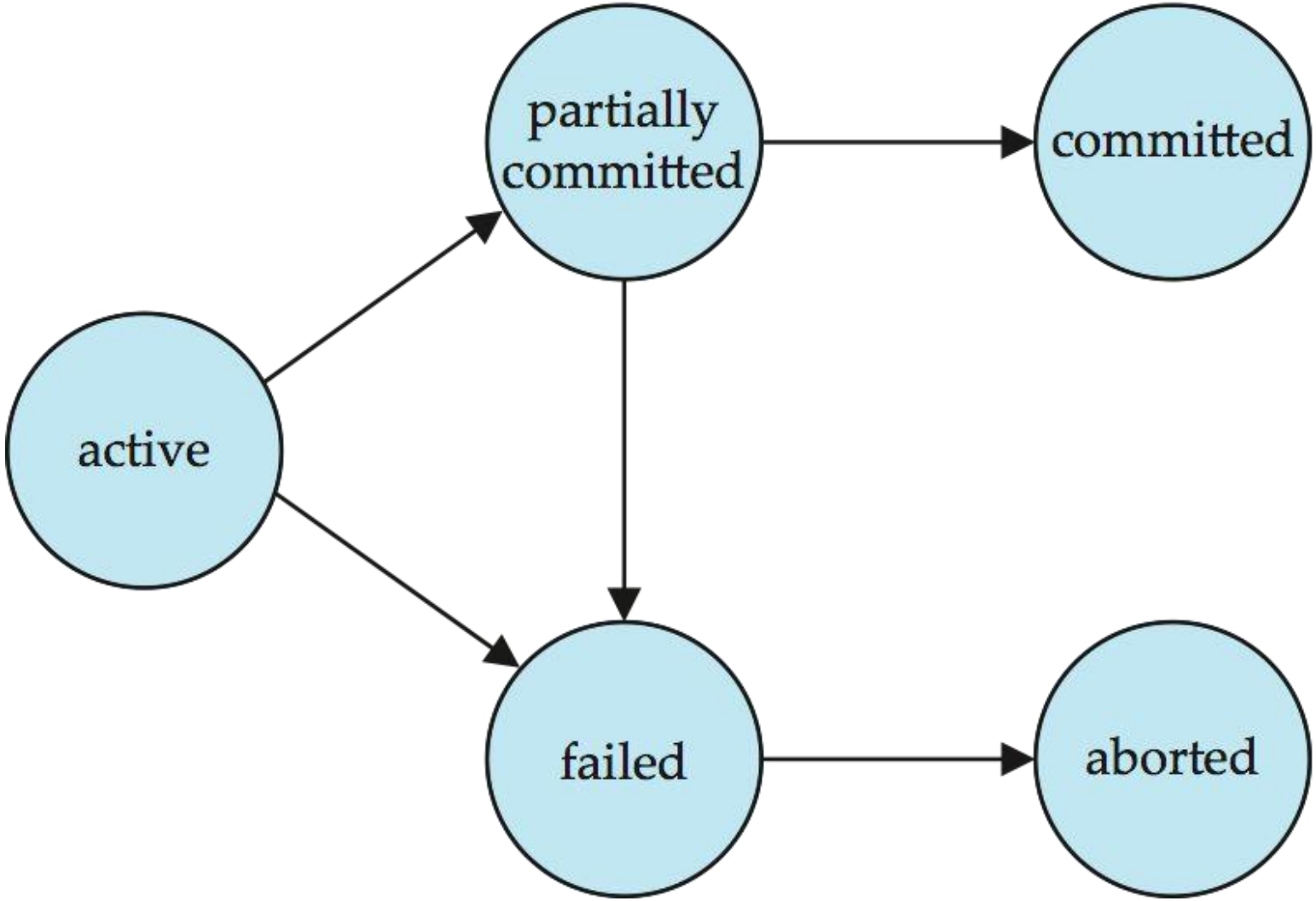
Partially committed – after the final statement has been executed.

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.


partially committed
committed
active
failed
aborted

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.

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	
A = 100 A = 50 Write A = 50	read (A) $A := A - 50$ write (A)		
		read (A) $temp := A * 0.1$ $A := A - temp$ write (A)	A = 50 temp = 5 A = 45 **Write A = 45**
B = 10 B = 60 Write B = 60	read (B) $B := B + 50$ write (B) commit		
		read (B) $B := B + temp$ write (B) commit	B = 60 B = 65 **Write B = 65**

A + B = 110

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

Serializability

- Serializability is a concept that helps us to check which schedules are serializable.
- A serializable schedule is the one that always leaves the database in consistent state.
- A serial schedule is always a serializable schedule because in serial schedule, a transaction only starts when the other transaction finished execution.
- Non-serial schedule may leave the database in inconsistent state so we need to check these non-serial schedules for the Serializability.

- **Types of Serializability**

- 1. **Conflict Serializability**-A schedule is called conflict serializable if we can convert it into a serial schedule after swapping its non-conflicting operations.

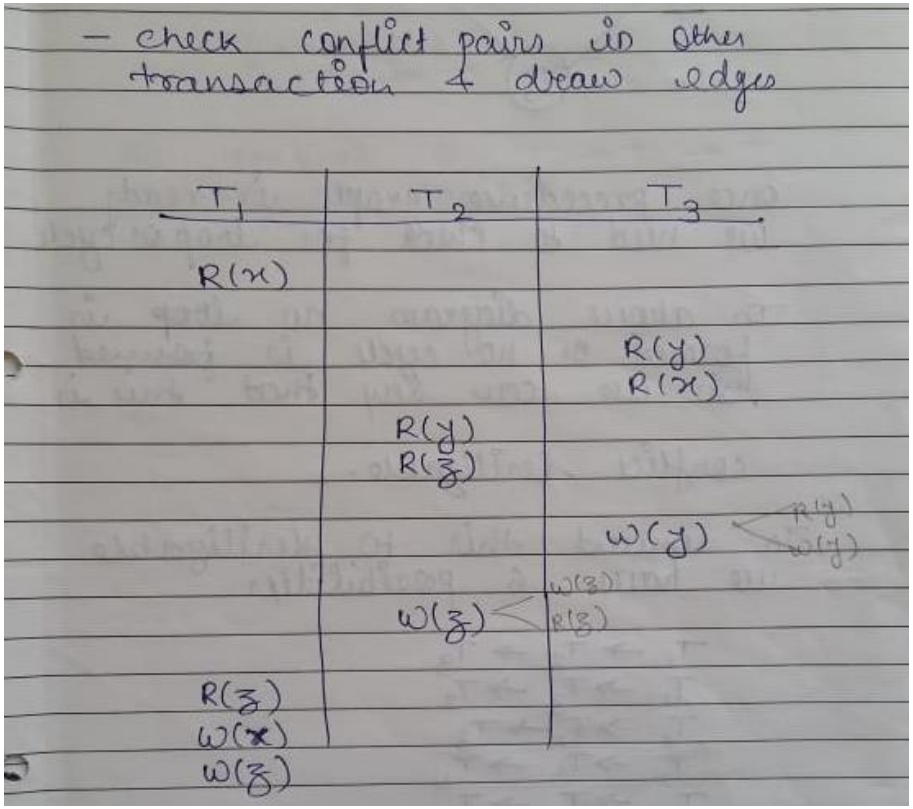
2. **View Serializability**-View Serializability is a process to find out that a given schedule is view serializable or not.

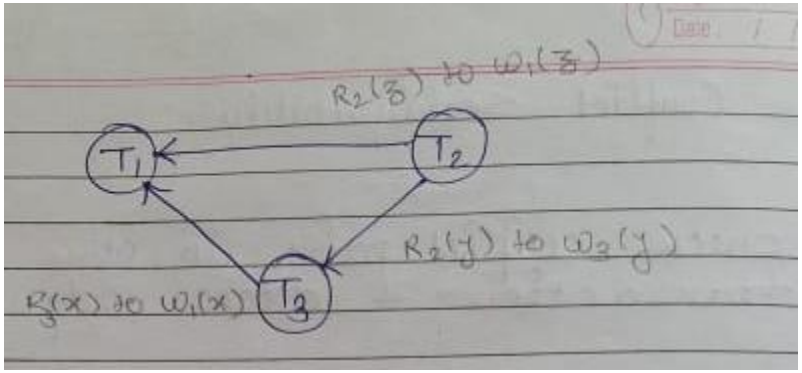
- We need to check whether given schedule is view equivalent to its serial schedule or not.
- To check we need to follow following rules

1. Read initial value of data item.
2. Write final value of data item.
3. Check W->R conflict

Conflict Serializability

A schedule is called conflict serializable if it can be transformed into a serial schedule by swapping non-conflicting operations.





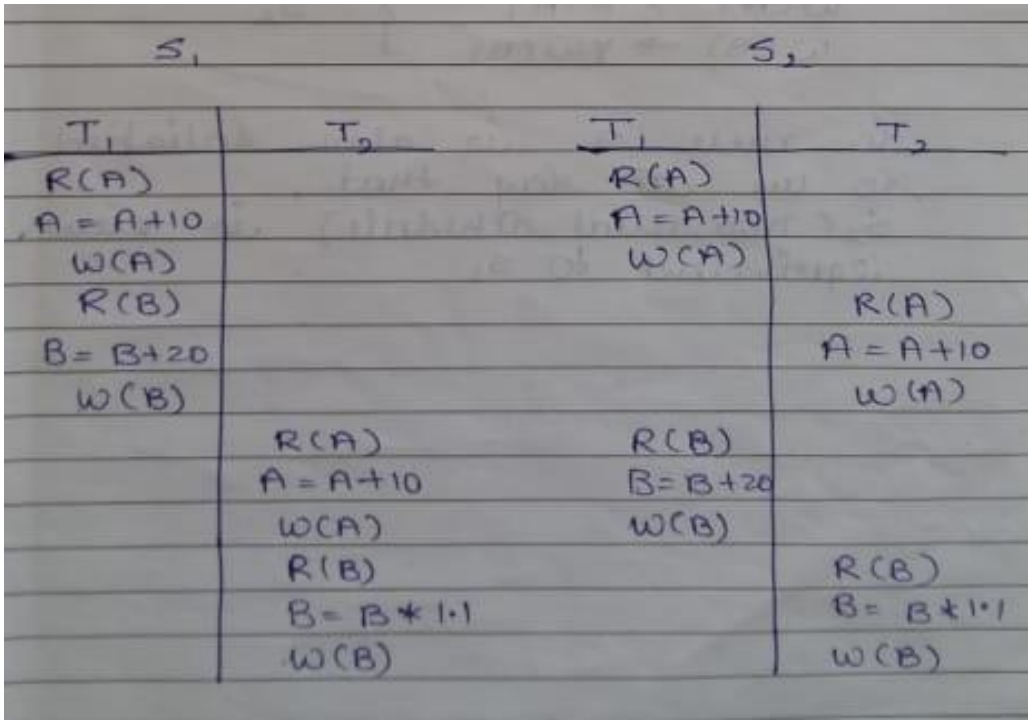
Conflict Serializability

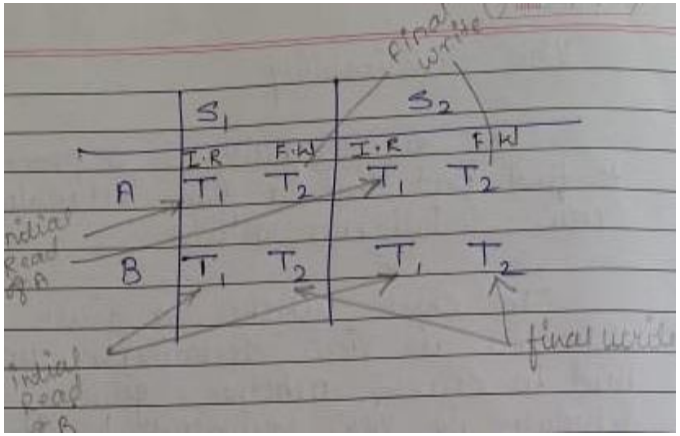
- T1->T2->T3
- T1->T3->T2
- T2->T1->T3
- T2->T3->T1
- T3->T2->T1
- T3->T1->T2.

View Serializability

- View serializability is a process to find out that a given schedule is view serializable or not
- To check we need to follow following rules

1. Read initial value of data item
2. Write final value of data item
3. Check W->R conflict





Recoverable Schedules

- Schedules in which transactions commit only after all transactions, whose changes they read commit are called recoverable schedules.
- In other words, if some transaction T_j is reading value updated or written by some other transaction T_i, then the commit of T_j must occur after the commit of T_i.

- There can be three types of recoverable schedule:
  1. Cascading Schedule
  2. Cascadeless Schedule
  3. Strict Schedule

Cascading Rollbacks

- **Cascading rollback** – When there is a failure in one transaction and this leads to the rolling back or aborting other dependent transactions, then such scheduling is referred to as Cascading rollback or cascading abort.

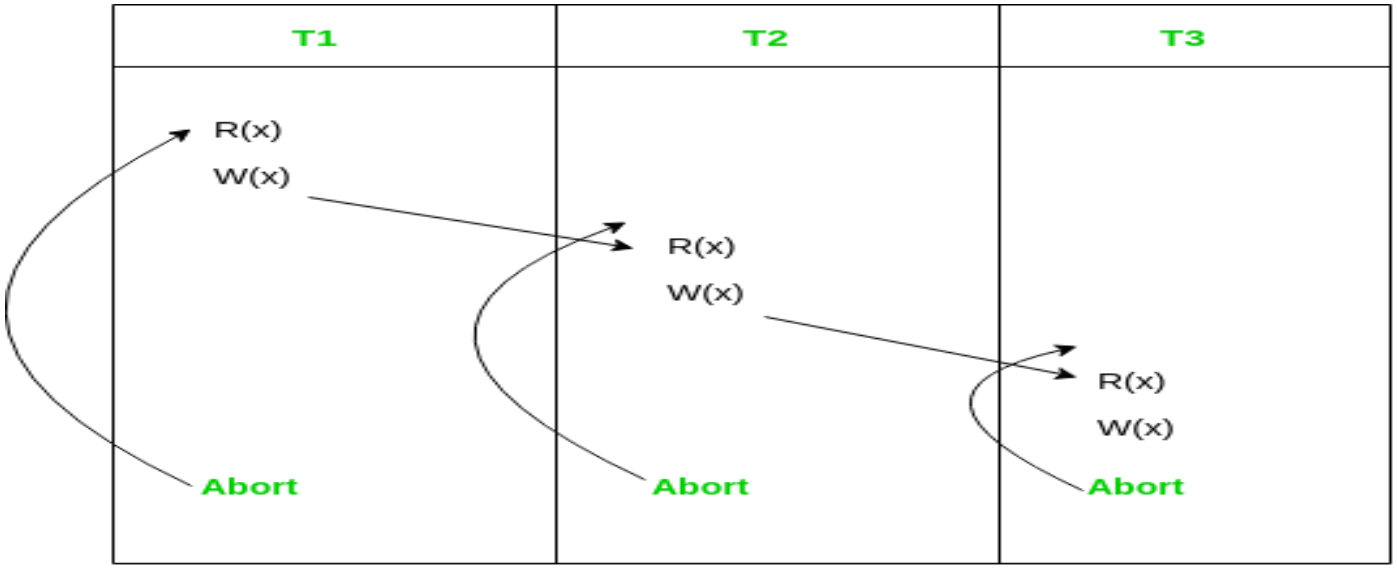


Figure - Cascading Abort

Cascadeless Schedules

- **Cascadeless schedules** — Schedules in which transactions read values only after all transactions whose changes they are going to read commit are called cascadeless schedules.
- A strategy to prevent cascading aborts is to disallow a transaction from reading uncommitted changes from another transaction in the same schedule.

T1	T2
R(A)	
W(A)	
	W(A)
Commit	
	R(A)
	Commit

- This schedule is cascadeless. Since the updated value of **A** is read by T_2 only after the updating transaction i.e. T_1 commits.

Strict Schedule

- A schedule is strict if a value written by a transaction can not be read or overwritten by other transactions until the transaction either commit or abort.
- Or
- if schedule contains no **read** or **write** before commit then it is known as strict schedule. Strict schedule is strict in nature.

T1	T2	T3
R1(x)		
	R2(x)	
R1(z)		
		R3(x)
		R3(y)
W1(x)		
C1;		
		W3(y)
		C3;
	R2(y)	
	W2(z)	
	W2(y)	
	C2;	

Figure - Strict Schedule

In this schedule no read-write or write-write conflict arises before commit hence its strict schedule

Non recoverable Schedules

- Consider the following schedule involving two transactions T_1 and T_2.

T1	T2
R(A)	
W(A)	
	R(A)
	Commit
Abort	

- T_2 read the value of A written by T_1, and committed. T_1 later aborted, therefore the value read by T_2 is wrong, but since T_2 committed, this schedule is **non-recoverable**

Concurrency Control

- Concurrency control is the procedure in DBMS for managing simultaneous operations without conflicting with each another.
- In the concurrency control, the multiple transactions can be executed simultaneously.
- It may affect the transaction result. It is highly important to maintain the order of execution of those transactions.

Problems of concurrency control

Several problems can occur when concurrent transactions are executed in an uncontrolled manner.

Following are the three problems in concurrency control.

- Lost updates
- Dirty read
- Unrepeatable read

Concurrency Control (Cont.)

Lost update problem

- When two transactions that access the same database items contain their operations in a way that makes the value of some database item incorrect, then the lost update problem occurs.

- If two transactions T1 and T2 read a record and then update it, then the effect of updating of the first record will be overwritten by the second update.

Dirty Read

- The dirty read occurs in the case when one transaction updates an item of the database, and then the transaction fails for some reason. The updated database item is accessed by another transaction before it is changed back to the original value.

- A transaction T1 updates a record which is read by T2. If T1 aborts then T2 now has values which have never formed part of the stable database.

- **Inconsistent Retrievals Problem / Unrepeatable read**
- Inconsistent Retrievals Problem is also known as unrepeatable read. When a transaction calculates some summary function over a set of data while the other transactions are updating the data, then the Inconsistent Retrievals Problem occurs.

- A transaction T1 reads a record and then does some other processing during which the transaction T2 updates the record. Now when the transaction T1 reads the record, then the new value will be inconsistent with the previous value.

Concurrency Control Protocol

- Different concurrency control protocols offer different benefits between the amount of concurrency they allow and the amount of overhead that they impose.
- 1. Lock-Based Protocols
- 2. Two Phase
- 3. Timestamp-Based Protocols
- 4. Validation-Based Protocols

- 1. Lock-Based Protocol
- In this type of protocol, any transaction cannot read or write data until it acquires an appropriate lock on it. All lock requests are made to the concurrency-control manager. Transactions proceed only once the lock request is granted.
- There are two types of lock:
- i. Shared lock
- ii. Exclusive lock

- **i. Shared lock (S Lock) -**
- A shared lock is also called a Read-only lock. With the shared lock, the data item can be shared between transactions. This is because you will never have permission to update data on the data item.

- For example, consider a case where two transactions are reading the account balance of a person. The database will let them read by placing a shared lock. However, if another transaction wants to update that account's balance, shared lock prevent it until the reading process is over.

- **ii. Exclusive lock (X Lock) –**
- In the exclusive lock, the data item can be both reads as well as written by the transaction.
- This lock is exclusive, and in this lock, multiple transactions do not modify the same data simultaneously.

- For example, when a transaction needs to update the account balance of a person. You can allows this transaction by placing X lock on it. Therefore, when the second transaction wants to read or write, exclusive lock prevent this operation.

- 2. Two Phase
- In this type of locking protocol, the transaction should acquire a lock after it releases one of its locks.

- The Two-Phase Locking protocol allows each transaction to make a lock or unlock request in two steps:
- **Growing Phase**: In this phase transaction may obtain locks but may not release any locks.
- **Shrinking Phase**: In this phase, a transaction may release locks but not obtain any new lock

- 3. Timestamp-Based Protocols-
- The timestamp-based algorithm uses a timestamp to serialize the execution of concurrent transactions.
- This protocol ensures that every conflicting read and write operations are executed in timestamp order.
- The protocol uses the **System Time or Logical Count as** a Timestamp.
- The older transaction is always given priority in this method

- 4.Validation-Based Protocols

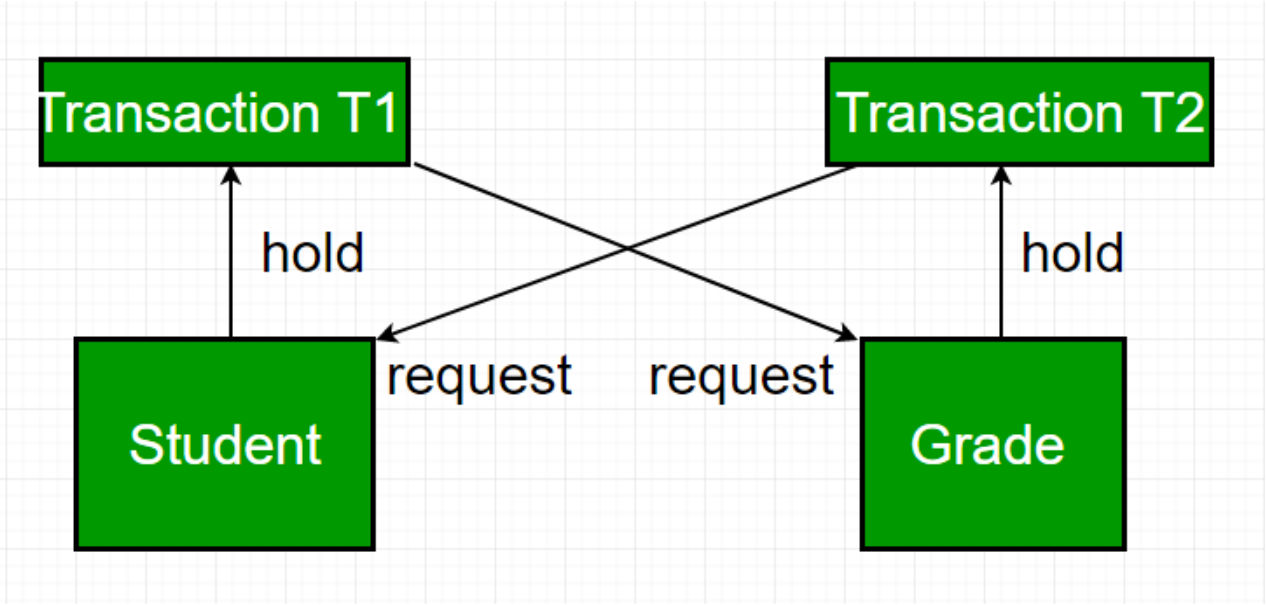
- This protocol is used in DBMS for avoiding concurrency in transactions.
- The three phases for validation based protocol:
- **Read phase:** In this phase, the transaction T is read and executed. It is used to read the value of various data items and stores them in temporary local variables.
- **Validation Phase:**
  In this phase, the temporary variable value will be validated against the actual data, to make sure that there is no violation of serializability.
- **Write phase:** If the validation of the transaction is validated, then the temporary results are written to the database otherwise, the updates are discarded and the transaction is rolled back.

Concurrency Control vs. Serializability Tests

- Concurrency-control protocols allow concurrent schedules, but ensure that the schedules are conflict/view serializable, and are recoverable and cascadeless .
- Concurrency control protocols generally do not examine the precedence graph as it is being created
  - Instead a protocol imposes a discipline that avoids nonseralizable schedules.
- Different concurrency control protocols provide different tradeoffs between the amount of concurrency they allow and the amount of overhead that they incur.
- Tests for serializability help us understand why a concurrency control protocol is correct.

Deadlocks

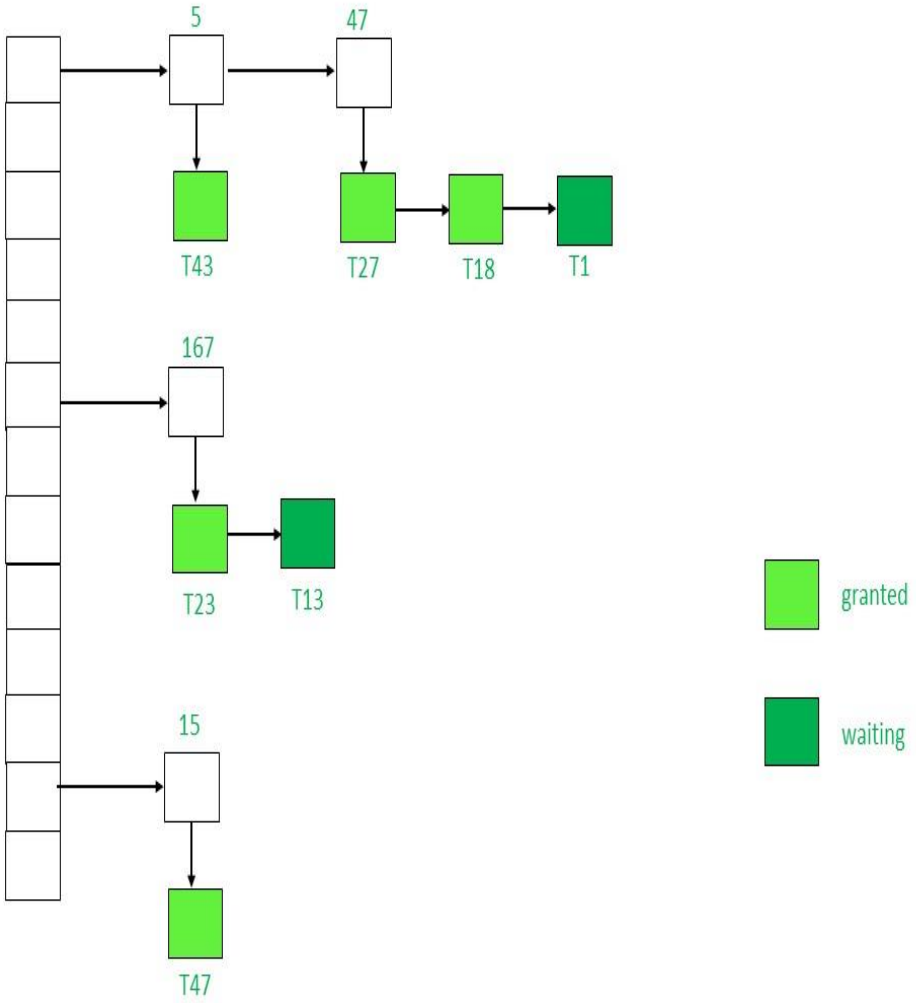
- In a database, a deadlock is an unwanted situation in which two or more transactions are waiting indefinitely for one another to give up locks.
- Suppose, Transaction T1 holds a lock on some rows in the Students table and **needs to update** some rows in the Grades table. Simultaneously, Transaction **T2 holds** locks on those very rows (Which T1 needs to update) in the Grades table **but needs** to update the rows in the Student table **held by Transaction T1**.
- Now, the main problem arises. Transaction T1 will wait for transaction T2 to give up lock, and similarly transaction T2 will wait for transaction T1 to give up lock.



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



- Initially the lock table is empty as no data item is locked.
- Whenever lock manager receives a lock request from a transaction T_i on a particular data item Q_i following cases may arise:
  - If Q_i is not already locked, a linked list will be created and lock will be granted to the requesting transaction T_i.
  - If the data item is already locked, a new node will be added at the end of its linked list containing the information about request made by T_i.
- If the lock mode requested by T_i is compatible with lock mode of transaction currently having the lock, T_i will acquire the lock too and status will be changed to 'granted'. Else, status of T_i's lock will be 'waiting'.
- If a transaction T_i wants to unlock the data item it is currently holding, it will send an unlock request to the lock manager. The lock manager will delete T_i's node from this linked list. Lock will be granted to the next transaction in the list.
- Sometimes transaction T_i may have to be aborted. In such a case all the waiting request made by T_i will be deleted from the linked lists present in lock table. Once abortion is complete, locks held by T_i will also be released.

Deadlock Handling

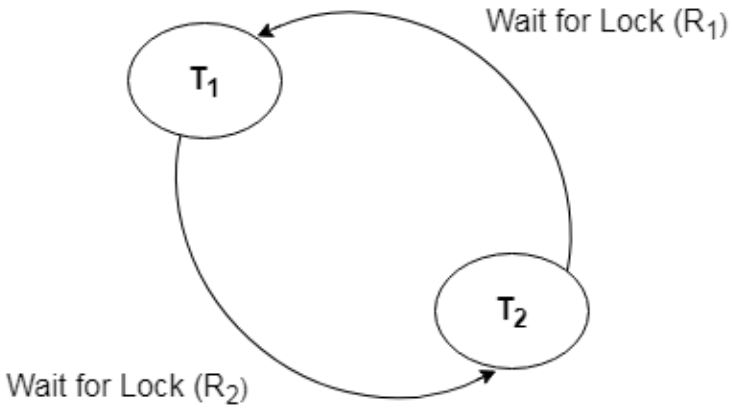
- System is deadlocked if there is a set of transactions such that every transaction in the set is waiting for another transaction in the set.
- ***Deadlock prevention*** protocols ensure that the system will *never* enter into a deadlock state. Some prevention strategies :
  - Require that each transaction 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 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 Detection

- In a database, when a transaction waits indefinitely to obtain a lock, then the DBMS should detect whether the transaction is involved in a deadlock or not.
- The lock manager maintains a **wait-for** the graph to detect the deadlock cycle in the database.
- 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 Recovery

- Deadlock recovery is generally used when deadlocks are rare and the cost of recovery is low.

Methods for Recovery

1. Termination of processes

- Some victim process is chosen for termination from the cycle of deadlocked processes.
- This process is terminated, requiring a later restart All the resources allocated to this processes are released, so that they may be reassigned to other deadlocked processes

2. Rolling back processes

- In order to rollback a victim process, there needs to have been some previous checkpoint at which time the state of the victim process was saved to stable storage
- There must also be an assurance that the rolled back process is not holding any resource needed by the other deadlocked processes at that point.

- Starvation happens if same transaction is always chosen as victim. Include the number of rollbacks in the cost factor to avoid starvation

Failure Classification

- **Transaction failure** :
  - **Logical errors**: transaction cannot complete due to some internal error condition
  - **System errors**: the database system must terminate an active transaction due to an error condition (e.g., deadlock)
- **System crash**: a power failure or other hardware or software failure causes the system to crash.
- **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 and Atomicity

- To ensure atomicity despite failures, we first output information describing the modifications to stable storage without modifying the database itself.
- **log-based recovery mechanisms-**
- The log is a sequence of records. Log of each transaction is maintained in some stable storage so that if any failure occurs, then it can be recovered from there.
- If any operation is performed on the database, then it will be recorded in the log.

Log-Based Recovery

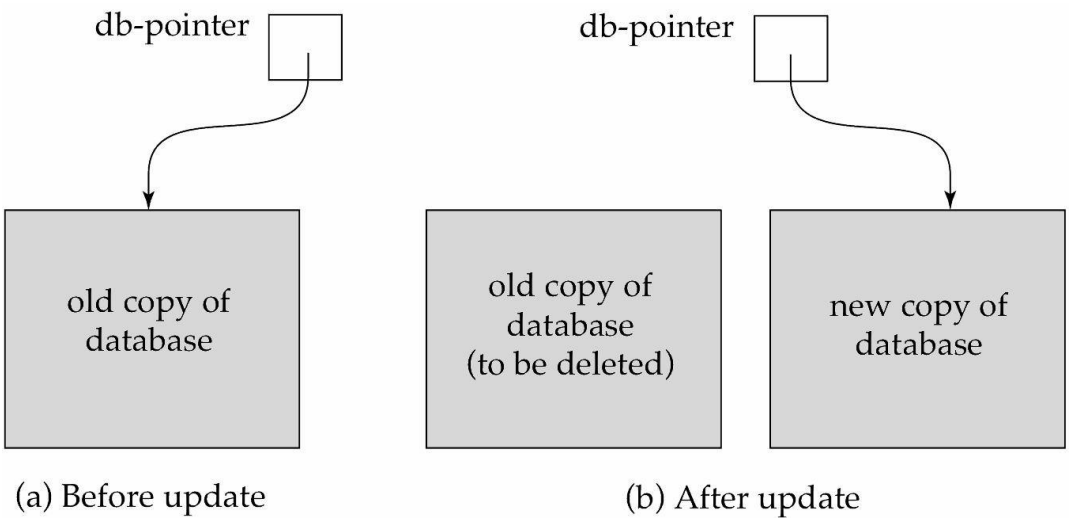
- A **log** is kept on stable storage.
  - The log is a sequence of **log records**, and maintains a record of update activities on the database.
- When transaction T_i starts, it registers itself by writing a

  $<T_i$ **start**>log record
- *Before* T_i executes **write**(X), a log record

  $<T_i, X, \quad V_1, \quad V_2>$

  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**).

- **Shadow-copy** and **Shadow-paging**
- This is the method where all the transactions are executed in the primary memory or the shadow copy of database.
- Once all the transactions completely executed, it will be updated to the database.
- if there is any failure in the middle of transaction, it will not be reflected in the database. Database will be updated after all the transaction is complete.



Checkpoints

- The checkpoint is a type of mechanism where all the previous logs are removed from the system and permanently stored in the storage disk.
- Checkpoint declares a point before which the DBMS was in consistent state, and all the transactions were committed.
- The recovery system reads the logs backwards from the end to the last checkpoint.
- It maintains two lists, an undo-list and a redo-list.
- If the recovery system sees a log with <tn, start=""> and <tn, commit=""> or just <tn, commit="">, it puts the transaction in the redo-list.
- If the recovery system sees a log with <tn, start=""> but no commit or abort log found, it puts the transaction in undo-list.
- All the transactions in the undo-list are then undone and their logs are removed.
- All the transactions in the redo-list and their previous logs are removed and then redone before saving their logs.

Checkpoints (Cont.)

- 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
     - Only transactions that are in L or started after the checkpoint need to be redone or undone
     - 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
  1. 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.