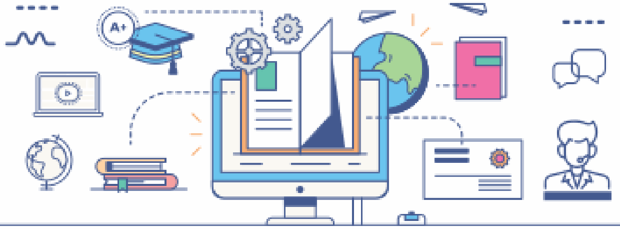
Unit 4
Transaction Processing

Prof. Vishwa Dabhi, Prof. Atul Kumar, Assistant Professor

Computer science and Engineering



CHAPTER-4

Transaction Processing

Transaction

- Collection of operations that form a single logical unit of work are called transactions.
- A transaction is a unit of program execution that accesses and possibly updates various data items.
- A transaction is a logical unit of work that contains one or more SQL statements.
- Usually, a transaction is initiated by a user program written in high-level data-manipulation language or programming language, where it is delimited by statements of the form **begin transaction** and **end transaction**.

ACID property

➢ To ensure integrity of the data, database system must maintain the following properties:

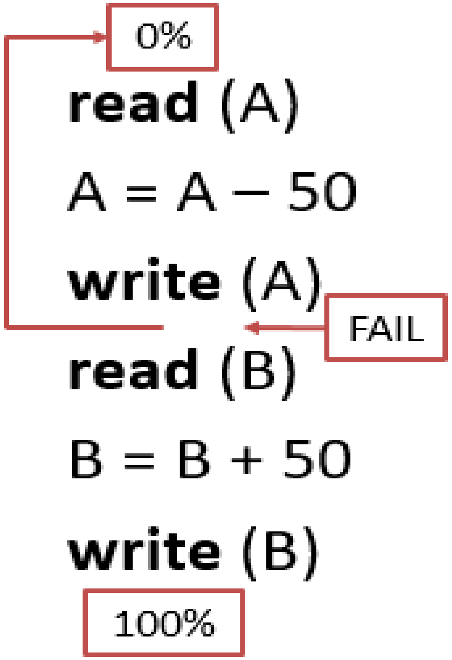
- Atomicity
- Consistency
- Isolation
- Durability

ACID property

- ➢ Atomicity: Either all operations of the transaction are reflected properly in the database, or none are.
- ➢ Consistency: Execution of a transaction in isolation i.ie., with no other transaction executing concurrently, preserves the consistency of the database.
- ➢ Isolation: Even though multiple transactions may execute concurrently, the system guarantees that, for every pair of transactions Ti and Tj, it appears to Ti that either Tj finished execution before Ti started or Tj started execution after Ti finished. Thus, each transaction is unaware of other transactions executing concurrently in the system.
- ➢ Durability: After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.

Atomicity

- ➢ This property states that a transaction must be treated as an atomic unit, i.e., either all of its operations are executed or none.
- ➢ For example, consider a transaction to transfer Rs. 50 from account A to account B.
- ➢ In this transaction, if Rs. 50 is deducted from account A then it must be added to account B.



0%

read (A)

A = A – 50

write (A)

FAIL

read (B)

B = B + 50

write (B)

100%

Consistency

- ➢ The database must remain in a consistent state after any transaction.
- ➢ If the database was in a consistent state before the execution of a transaction, it must remain consistent after the execution of the transaction as well.
- ➢In our example, total of A and B must remain same before and after the execution of transaction.

A=500, B=500

A+B=1000

read (A)

A = A – 50

write (A)

read (B)

B = B + 50

write (B)

A=450, B=550

A+B=1000

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.
- In the example once your transaction start from step one its result should not be accessed by any other transaction until last step (step 6) is completed.

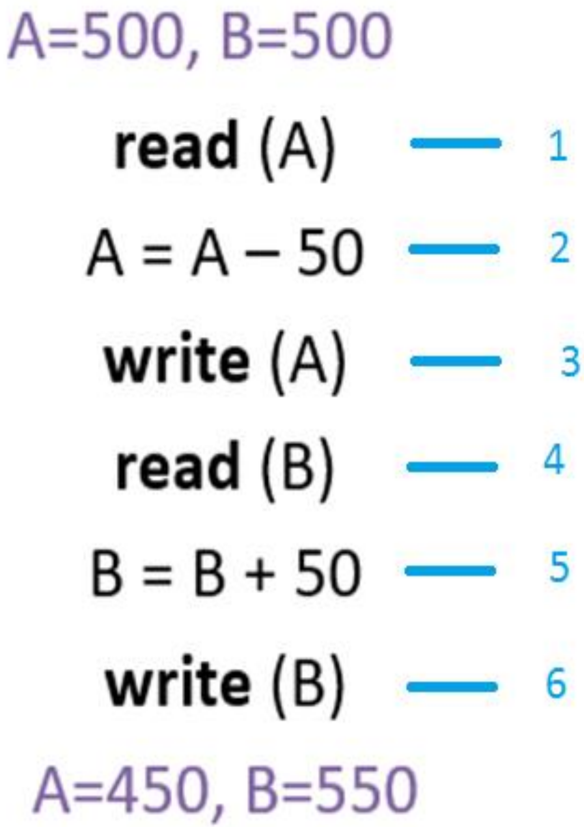
A=500, B=500

read (A)	1
A = A – 50	2
write (A)	3
read (B)	4
B = B + 50	5
write (B)	6

A=450, B=550

Durability

- ➢ After a transaction completes successfully, the changes it has made to the database persist(permanent), even if there are system failures.
- ➢Once our transaction is completed up to last step (step 6) its result must be stored permanently. It should not be removed if the system fails.



A=500, B=500

read (A)	1
A = A – 50	2
write (A)	3
read (B)	4
B = B + 50	5
write (B)	6

A=450, B=550

Transaction State

- A transaction may not always complete its execution successfully, such transaction is termed **aborted**.
- Once the changes caused by an aborted transaction have been undone, we say that the transaction has been **rolled back**.
- A transaction that completes its execution successfully is said to be **committed**.
- Once the transactions are committed, we cannot undo its effects by aborting it. The only way to undo the effects of a committed transaction is to execute a **compensating transaction**.

Transaction State

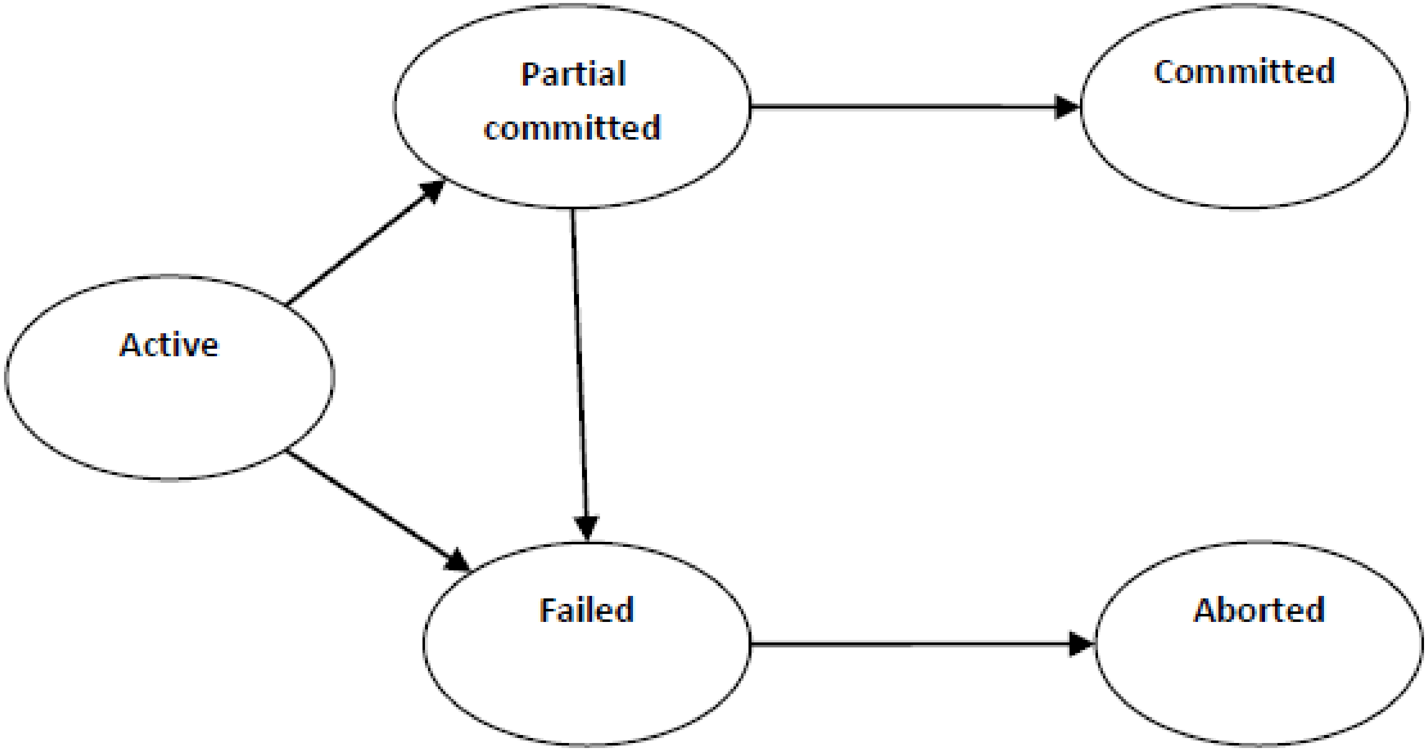


Fig. State Transition Diagram

Image source : Google

Transaction State

➢ A transaction must be in one of the following states:

- **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 has been restored to its state prior to the start of the transaction.
- **Committed**, after successful transaction.

The Transaction Log

- Keeps track of all transactions that update the database. It contains:
  - A record for the beginning of transaction
  - For each transaction component (SQL statement)
    - Type of operation being performed (update, delete, insert)
    - Names of objects affected by the transaction (the name of the table)
    - “Before” and “after” values for updated fields
    - Pointers to previous and next transaction log entries for the same transaction
  - The ending (COMMIT) of the transaction
- This increases processing overhead but the advantage is the **ability to restore a corrupted database**

Implementation of Atomicity and Durability

- The **recovery-management** component of a database system implements the support for atomicity and durability.
- E.g. the ***shadow-database*** scheme:
  - all updates are made on a *shadow copy* of the database
    - **db_pointer** is made to point to the updated shadow copy after
      - the transaction reaches partial commit and
      - all updated pages have been flushed to disk.

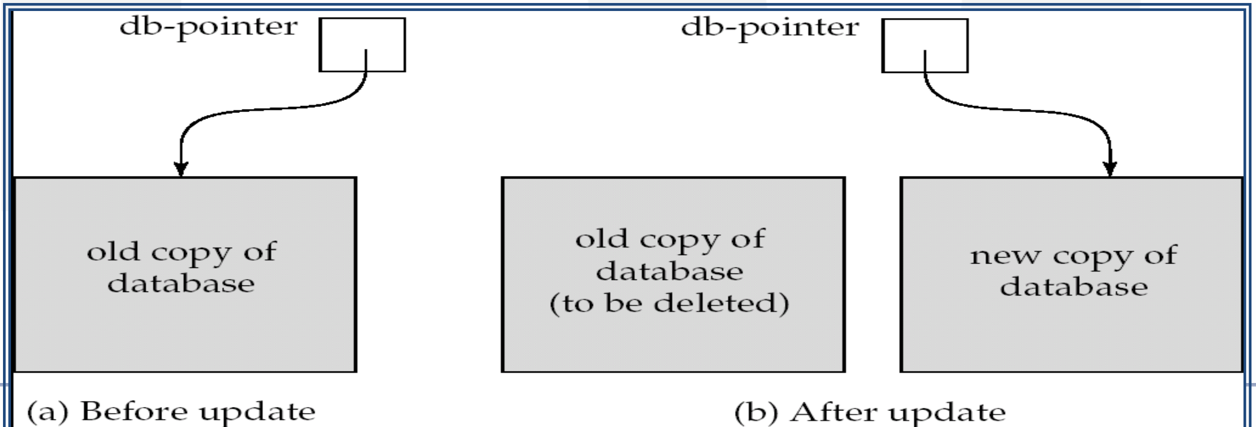


Image source : Google

Implementation of Atomicity and Durability

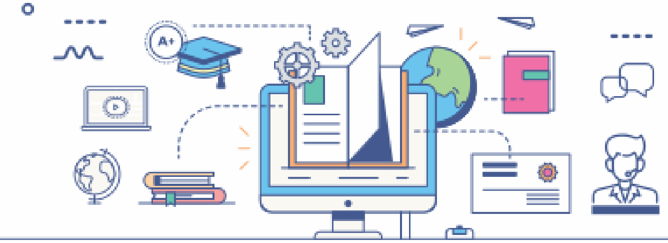
- db_pointer always points to the current consistent copy of the database.
  - In case transaction fails, old consistent copy pointed to by **db_pointer** can be used, and the shadow copy can be deleted.
- The shadow-database scheme:
  - Assumes that only one transaction is active at a time.
  - Assumes disks do not fail
  - Useful for text editors, but
    - extremely inefficient for large databases (why?)
      - Variant called shadow paging reduces copying of data, but is still not practical for large databases
  - Does not handle concurrent transactions

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

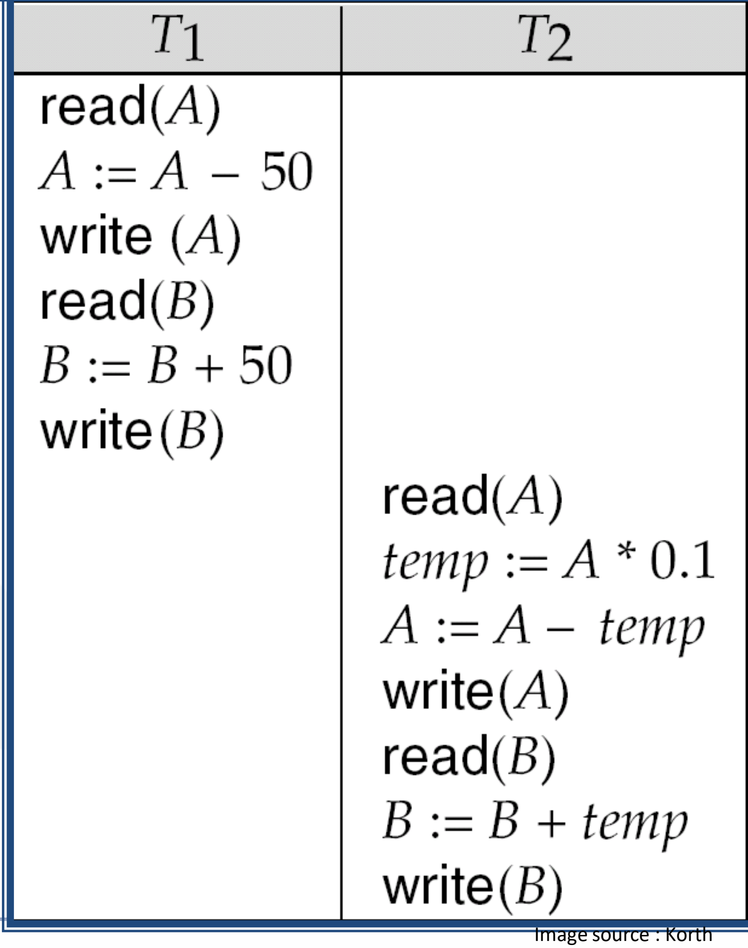
Schedules

- **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 :



$T1$	$T2$

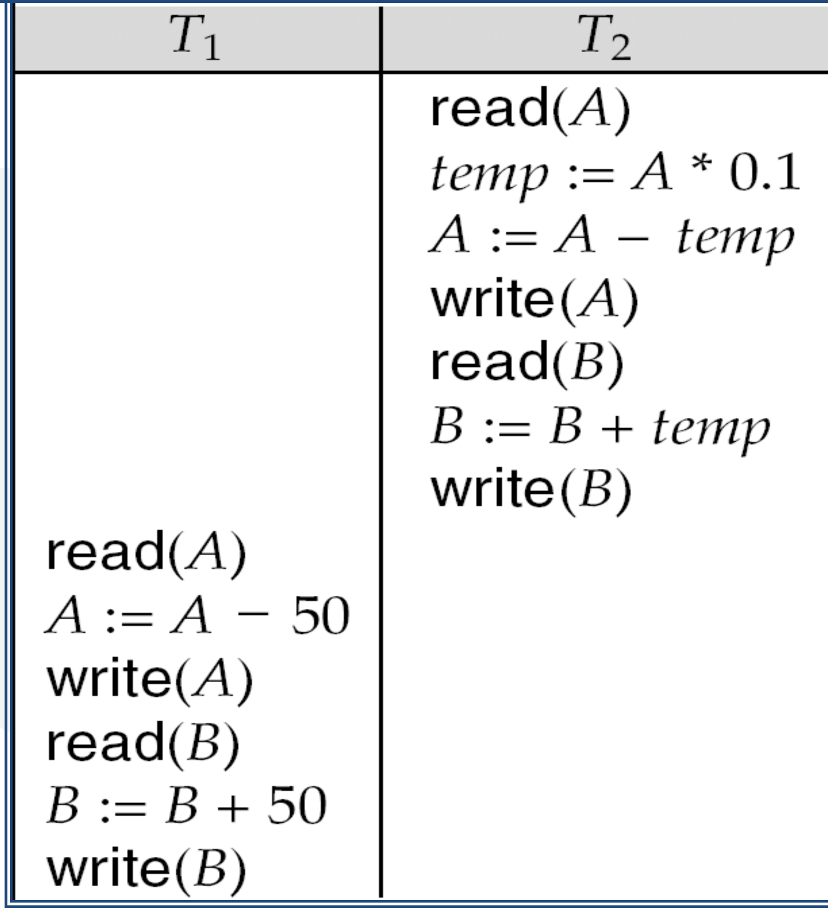
read(A)
$A := A - 50$
write (A)
read(B)
$B := B + 50$
write(B)	

	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
	$B := B + temp$
	write(B)

Image source : Korth

Schedule - 2

- ➢ A serial schedule where T_2 is followed by T_1

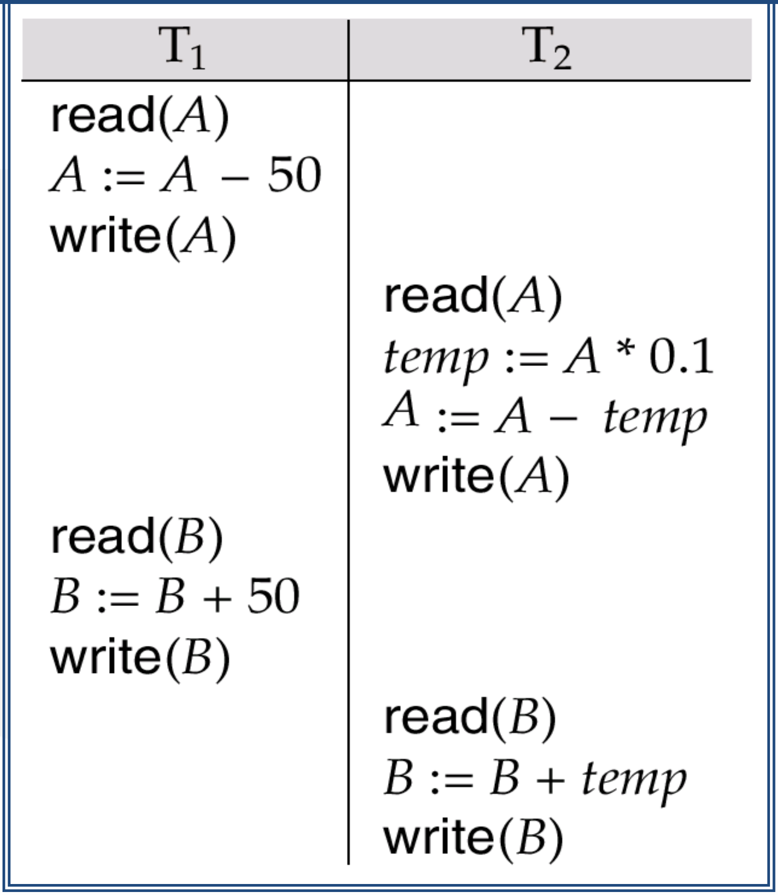


T_1	T_2
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
	$B := B + temp$
	write(B)
read(A)	
$A := A - 50$	
write(A)	
read(B)	
$B := B + 50$	
write(B)	

Image source : Korth

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)

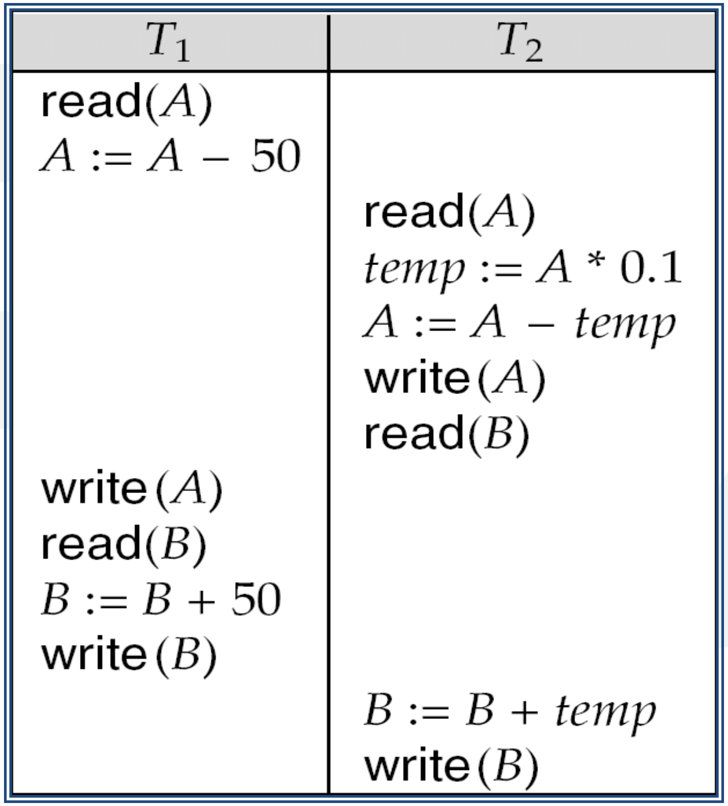
read(B)
$B := B + 50$
write(B)	

	read(B)
	$B := B + temp$
	write(B)

Image source : Korth

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)	
	$B := B + temp$
	write(B)

Image source : Korth

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**

Serializability

- Simplified view of transactions
  - We ignore operations other than read and write instructions
  - We assume that transactions may perform arbitrary computations on data in local buffers in between reads and writes.
  - Our simplified schedules consist of only read and write instructions.

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

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

- 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

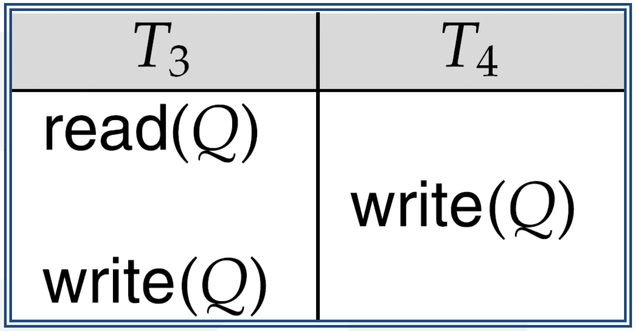
T_1	T_2
read(A)	
write(A)	
read(B)	
write(B)	
	read(A)
	write(A)
	read(B)
	write(B)

Schedule 6

Image source : Korth

Conflict Serializability

- Example of a schedule that is not conflict serializable:



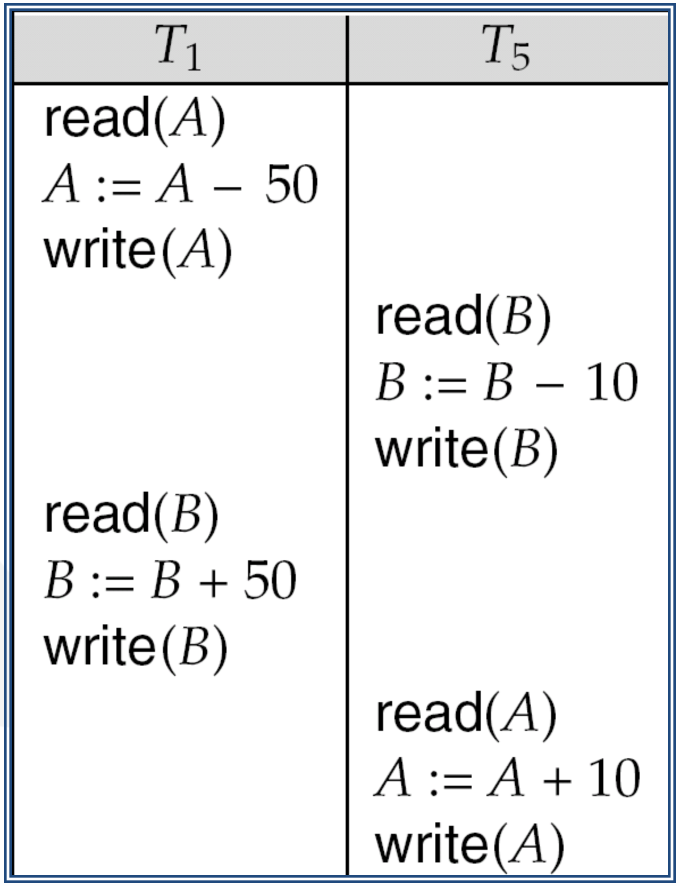
T_3	T_4
read(Q)	
	write(Q)
write(Q)	

- 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 >.

Image source : Korth

Other Notions of Serializability

- The schedule below produces same outcome as the serial schedule < T_1, T_5 >, yet is not conflict equivalent or view equivalent to it.
- Determining such equivalence requires analysis of operations other than read and write.



T_1	T_5
read(A)	
$A := A - 50$	
write(A)	
	read(B)
	$B := B - 10$
	write(B)
read(B)	
$B := B + 50$	
write(B)	
	read(A)
	$A := A + 10$
	write(A)

Image source : Korth

Testing for Serializability

- Consider some schedule of a set of transactions T_1, T_2, ..., T_n
- **Precedence graph** — a direct graph where the vertices are the transactions (names).
- We draw an arc from T_i to T_j if the two transaction conflict, and T_i accessed the data item on which the conflict arose earlier.
- We may label the arc by the item that was accessed.
- **Example 1**

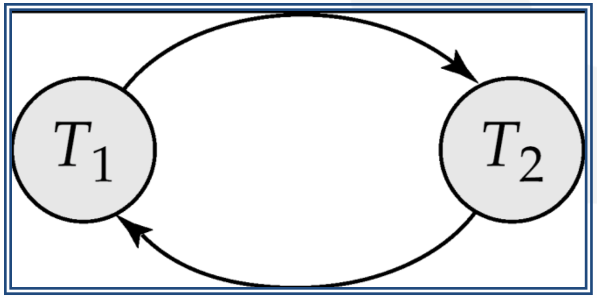


Image source : Google

Example Schedule (Schedule A) + Precedence Graph

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

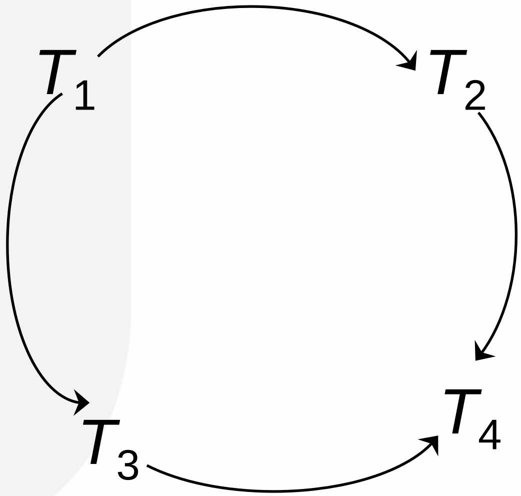


Image source : Google

Test for Conflict Serializability

- A schedule is conflict serializable if and only if its precedence graph is acyclic.
- Cycle-detection algorithms exist which take order n^2 time, where n is the number of vertices in the graph.
  - (Better algorithms take order $n + e$ where e is the number of edges.)
- If precedence graph is acyclic, the Serializability order can be obtained by a *topological sorting* of the graph.
  - This is a linear order consistent with the partial order of the graph.
  - For example, a Serializability order for Schedule A would be

    $T_5 \rightarrow T_1 \rightarrow T_3 \rightarrow T_2 \rightarrow T_4$

    - Are there others?

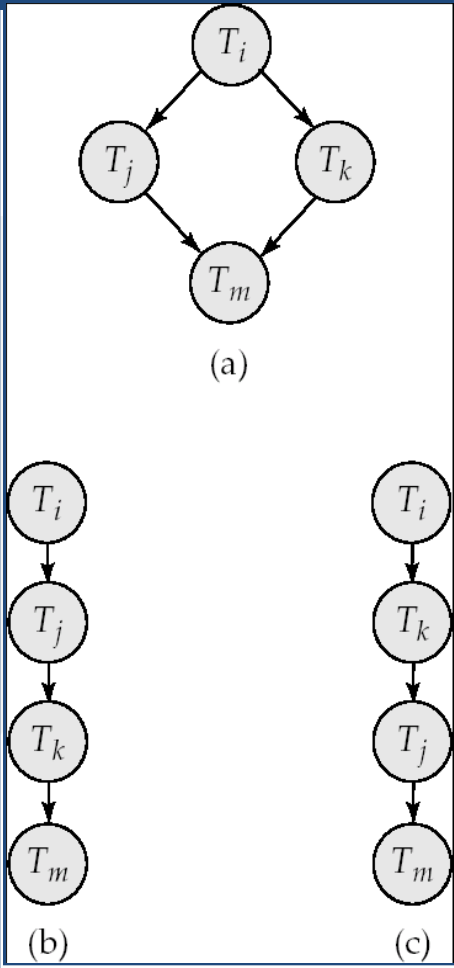


Image source : Google

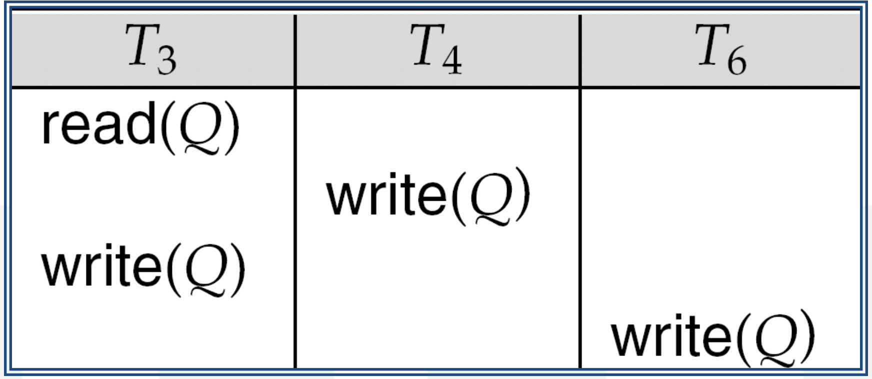
View Serializability

- 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



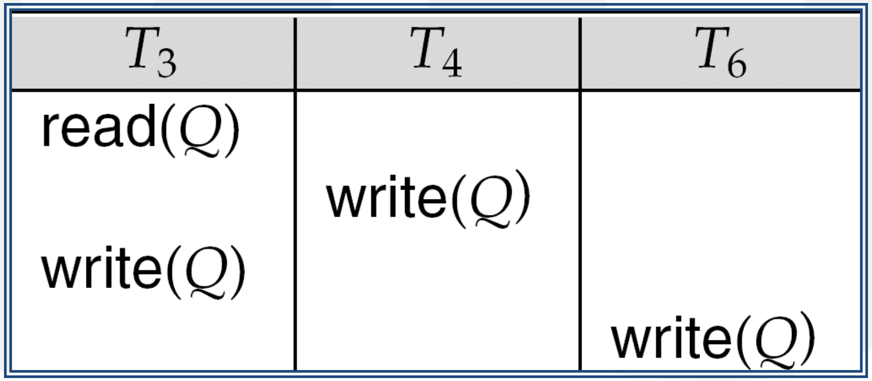
T_3	T_4	T_6
read(Q)		
	write(Q)	
write(Q)		
		write(Q)

- ➢Without read , if read perform is ‘Blind Write’
- ➢If there is no blind writes and if schedules is not CS than schedule is not serializable
- ➢If there is blind writes and if schedules is not CS than check for view serializable (VS).
- ➢Every CS is VS but vise versa is not true.
- ➢In view we have to check the sequence of only read.

Image source : Korth

View Serializability

- 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_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.**

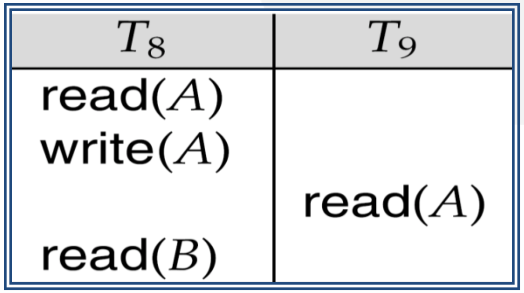
Image source : Korth

Test for View Serializability

- The precedence graph test for conflict Serializability cannot be used directly to test for view Serializability.
  - Extension to test for view Serializability has cost exponential in the size of the precedence graph.
- The problem of checking if a schedule is view serializable falls in the class of *NP*-complete problems.
  - Thus existence of an efficient algorithm is *extremely* unlikely.

Recoverable Schedules

- Need to address the effect of transaction failures on concurrently running transactions.
- **Recoverable schedule** — if a transaction T_j reads a data item previously written by a transaction T_i, then the commit operation of T_i appears before the commit operation of T_j.
- The following schedule (Schedule 11) is not recoverable if T_9 commits immediately after the read.
- If T_8 should abort, T_9 would have read (and possibly shown to the user) an inconsistent database state. Hence, database must ensure that schedules are recoverable.



T_8	T_9
read(A)	
write(A)	
	read(A)
read(B)	

Image source : Korth

Cascading Rollbacks

- **Cascading rollback** – a single transaction failure leads to a series of transaction rollbacks. Consider the following schedule where none of the transactions has yet committed (so the schedule is recoverable). If T_{10} fails, T_{11} and T_{12} must also be rolled back.
- Can lead to the undoing of a significant amount of work.

T_{10}	T_{11}	T_{12}
read(A)		
read(B)		
write(A)		
	read(A)	
	write(A)	
		read(A)

Image source : Korth

Cascadeless Schedules

- **Cascadeless schedules** — cascading rollbacks cannot occur; for each pair of transactions T_i and T_j such that T_j reads a data item previously written by T_i, the commit operation of T_i appears before the read operation of T_j.
- Every cascadeless schedule is also recoverable
- It is desirable to restrict the schedules to those that are cascadeless

Concurrency Control

- To ensure serializability practically use concurrency control protocol.
- 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
- A policy in which only one transaction can execute at a time generates serial schedules, but provides a poor degree of concurrency
  - Are serial schedules recoverable/cascadeless?
- Testing a schedule for serializability *after* it has executed is a little too late!
- **Goal** – to develop concurrency control protocols that will assure serializability.

Locking and Time Stamp Based Schedulers

- Whenever Multiple transactions are executed concurrently , It is exceptionally important to maintain concurrency between all of the concurrent transactions.
- Concurrency Protocols can be used to maintain ACID properties of all transactions and they can be divided into two categories:-

  (i) Lock Based Protocol

  (ii) Time Stamp based Protocol

Lock Based Protocols

- Lock based Protocol provides a mechanism which
- Locks are of two types:-

  (i) Lock Based Protocol

  (ii) Time Stamp based Protocol

Lock Based Protocols (Continued….)

- Lock based Protocols provide a mechanism in which any transaction cannot read or write data without suitable lock.
- Locks are of two types:-

(i) Binary Locks:- A lock on data items can be in two states only, it will be either locked or unlocked.

(ii) Shared/exclusive:- This type of locking mechanism differentiates the locks based on their uses. If a lock is acquired on a data item to perform a write operation, it is an exclusive lock.

Lock Based Protocols (Continued….)

- **There are four types of lock protocols available –**

  (i) Simplistic Lock Protocol

  (ii) Pre-claiming Lock Protocol

  (iii) Two-Phase Locking 2PL

  (iv) Strict Two-Phase Locking

Lock Based Protocols (Continued….)

(i) Simplistic Lock Protocol

(ii) Pre-claiming Lock Protocol

(iii) Two-Phase Locking 2PL

(iv) Strict Two-Phase Locking

Lock Based Protocols (Continued….)

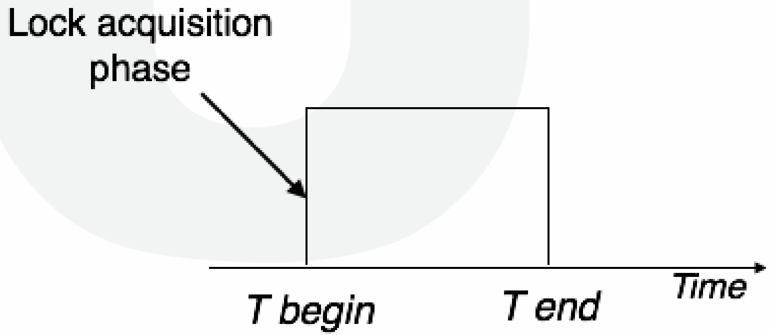
(i) Simplistic Lock Protocol :-

It means lock must be obtained before every “write” operation and after completion of “write” operation transaction must release lock

Lock Based Protocols (Continued....)

(ii) Pre-claiming Lock Protocol :-

- This Protocol will create a list of database operation and after evaluating them ,Locks will be granted.
- If all locks are granted , After completing all the database operations Transaction will release all the locks.



Lock Based Protocols (Continued….)

(iii) Two-Phase Locking 2PL:-

•According to this protocol Execution phase will get divided into three parts as follows:

(a) Transaction will need permission to get a lock

(b) Transaction will acquire all the locks

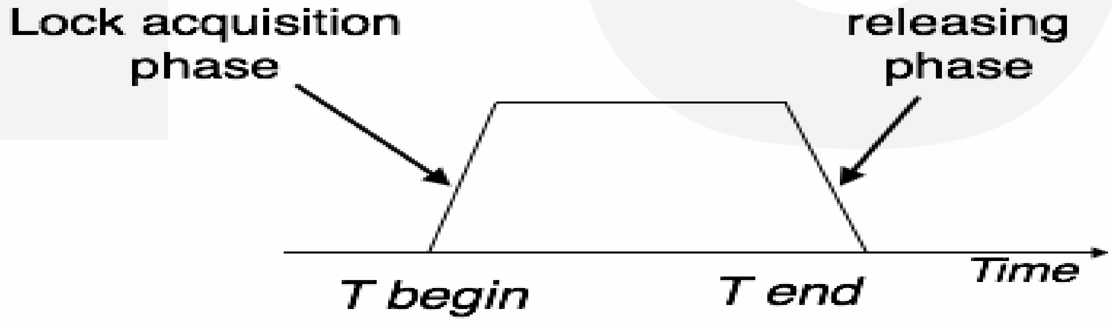
After completion when transaction will release its 1st lock then 3rd phase will start

(c) Now Transaction cannot get a new lock it will release the acquired lock.

Lock Based Protocols (Continued….)

(iii) Two-Phase Locking 2PL (Continued…..)

- This Process of acquiring and releasing lock is called as Growing and Shrinking Phase respectively.
- In **Growing Phase,** Locks will get acquired.
- In **Shrinking Phase,** Locks will get released.



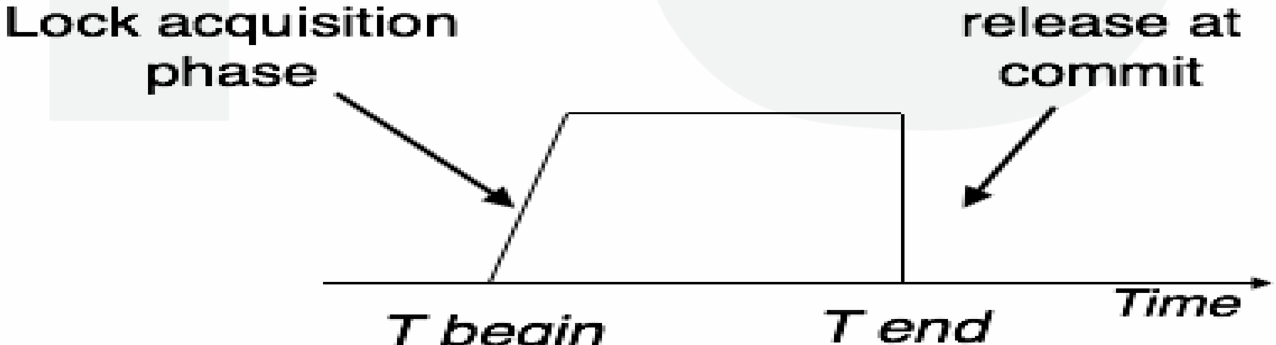
Lock Based Protocols (Continued….)

(iv) Strict Two-Phase Locking :-

•According to this protocol Execution phase will get divided into three parts as follows:

(a) Transaction will need permission to get a lock and will acquire all the locks

(b) Now Transaction is not allowed to release any lock until commit (successful transaction) transaction.



Time Stamp Based Protocols (Continued….)

- The timestamp-ordering protocol assures serializability between transactions in their conflict read and write operations. This is the liability of the protocol system that the conflict pair of tasks should be completed according to the timestamp values of the transactions.
- The timestamp of transaction T_i is denoted as $TS(T_i)$.
- Read time-stamp of data-item X is denoted by R-timestamp(X).
- Write time-stamp of data-item X is denoted by W-timestamp(X).

Time Stamp Based Protocols (Continued....)

- Timestamp ordering protocol works as follows –
- **If a transaction Ti issues a read(X) operation –**
  - If TS(Ti) < W-timestamp(X)
    - Operation rejected.
  - If TS(Ti) >= W-timestamp(X)
    - Operation executed.
  - All data-item timestamps updated.
- **If a transaction Ti issues a write(X) operation –**
  - If TS(Ti) < R-timestamp(X)
    - Operation rejected.
  - If TS(Ti) < W-timestamp(X)
    - Operation rejected and Ti rolled back.
  - Otherwise, operation executed.

Multi Version Concurrency Control(MVCC)

- MVCC will maintain more than one copy of same data item in order to avoid inconsistency.
- Whenever An user will need to update a piece of data in transaction then it will overwrite a copy of data not original data.
- MVCC allows POINT IN TIME CONSISTENT view. Read transactions under MVCC typically use a timestamp or Trans ID to decide which state of the database has to read, and read the particular versions of the data that is provided by MVCC.
- Due to above approach Read and Write both will get executed at different data items (copies of maintained data).

Multi Version Concurrency Control(MVCC)

- MVCC uses timestamp (**TS**), and incrementing transaction IDs, to attain consistency between transactions. MVCC assures a transaction (**T**) will never wait for *Reading* a database object (**D**) by maintaining several versions of the object. Each version of object **D** has both a *Read Timestamp* (**RTS**) and a *Write Timestamp* (**WTS**) which lets a specific transaction $\mathbf{T_i}$ read the most recent version of the object which precedes the transaction's *Read Timestamp* **RTS**($\mathbf{T_i}$).
- f transaction $\mathbf{T_i}$ wants to *Write* to object **D**, and there is also another transaction $\mathbf{T_k}$ happening to the same object, the Read Timestamp **RTS**(T_i) must precede the Read Timestamp **RTS**($\mathbf{T_k}$), i.e., **RTS**($\mathbf{T_i}$) < **RTS**($\mathbf{T_k}$).

Optimistic Concurrency Control(OCC)

- **Optimistic concurrency control** (**OCC**) is a concurrency control method applied to transactional systems such as relational database management systems and software transactional memory.
- OCC provides a concept in which more than one transaction can frequently execute without interfering with each other.
- During Execution, transactions utilize data resources without getting locks on those resources. Before committing, each transaction will verify that no other transaction has updated the data it has read. If the check confirms conflicting modifications, the committing transaction rolls back and will get restarted.

Optimistic Concurrency Control(OCC)

- Phases of OCC:-
- **Begin**: Save a timestamp by noting the transaction's beginning TS.
- **Modify**: Read database values, and tentatively write changes.
- **Validate**: Check whether other transactions have updated data that currently executed transaction has used (read or written). This includes transactions that completed after this transaction's start time, and optionally, transactions that are still active at validation time.
- **Commit/Rollback**: If there is no conflict, make all changes take effect. If there is a conflict, resolve it.

Database Recovery

- DBMS is a extremely complex system where thousands of transactions get executed each second. The resilience and robustness of a DBMS depends on its composite architecture and its underlying hardware and system software. If it fails or crashes during transactions, it is expected that the system would follow some sort of algorithm or techniques to recover lost data.
- There are two types of errors can which leads to transaction failure:-

    (a) Logical Error:- Code Error or any Query structure error

    (b) System Error:- Error occurred due to system failure e.g. Deadlock
- System Crash
- Disk Failure

Database Recovery (Continued….)

- **Recovery and Atomicity**
- When a system crashes, there ma be a possibility that some of the transactions are getting executed and various files are opened for them to update the data items. Transactions are made of different operations, which must be atomic in nature. But according to ACID property of DBMS, atomicity must be maintained throughout the transaction.
- Below points must be maintained in case of recovery from failure:-
- It should verify the current state of all the transactions, which were in execution
- A transaction may be in intermediate of some operation; the DBMS must assure the atomicity of the transaction in this case.
- It should verify whether the transaction can be completed now or it needs to be rolled back.

Database Recovery (Continued....)

- **Recovery and Atomicity**
- There are two types of techniques, which can help a DBMS in recovering as well as maintaining the atomicity of a transaction –
- By Maintaining the logs of each transaction, and writing them onto in some storage and then modify the actual data.
- Maintaining shadow paging.

Database Recovery (Continued….)

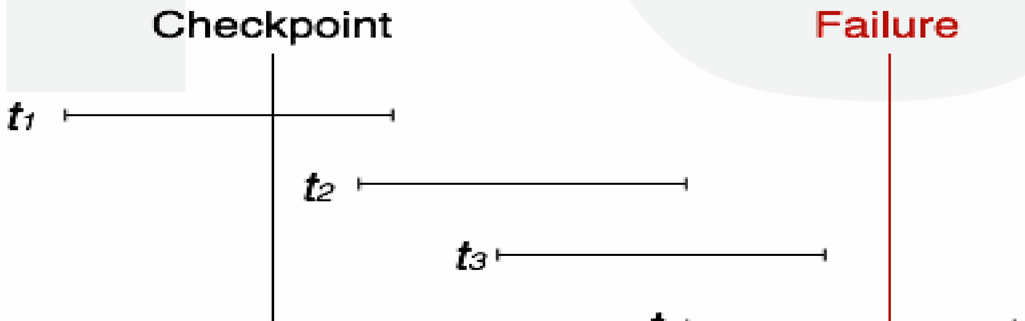
- **Log-based Recovery**
- Log is a list of records which are logically related, which stores the records of activities carried out by a transaction. It is important that the logs are written before actual modification and stored on a stable storage media, which must be safe.
- Log-based recovery works as follows –
- The log file is kept on a stable storage media.
- When a transaction enters the system and starts execution, it writes a log about it.
- <T_n, Start> When the transaction modifies an item X, it write logs as follows –
- <T_n, X, V_1, V_2> It reads T_n has changed the value of X, from V_1 to V_2.
- When the transaction finishes, it logs –
- <T_n, commit>

Database Recovery (Continued….)

- **Recovery with Concurrent Transactions**
- When some transaction are getting executed in parallel, the logs are interleaved. At the moment of recovery, it will become tough to recover system to restore all logs, and then start recover. To ease this situation, most modern DBMS use the concept of **'checkpoints'.**
- **Checkpoint**
- Checkpoint is a technique where all the previous logs are removed from the system and stored permanently in a storage disk. Checkpoint declares a point before which the DBMS was in consistent state, and all the transactions were committed.

Database Recovery (Continued….)

- **Recovery with Concurrent Transactions**
- Transaction behave in following manner when it crashes and then recover:-
- The 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 <T_n, Start> and <T_n, Commit> or just <T_n, Commit>, it puts the transaction in the redo-list.
- If the recovery system sees a log with <T_n, Start> but no commit or abort log found, it puts the transaction in undo-list.



DIGITAL LEARNING CONTENT

Parul® University

www.paruluniversity.ac.in