UNIT-5

5. TRANSACTIONS

5.1. Introduction

- A ***transaction*** is a unit of program execution that accesses and possibly updates various data items.
- The transaction consists of all operations executed between the statements **begin** and **end** of the transaction
- **Transaction operations:** Access to the database is accomplished in a transaction by the following two operations:
  - ***read (X):*** *Performs the reading operation of data item X from the database*
  - ***write (X):*** *Performs the writing operation of data item X to the database*
- A transaction must see a consistent database
- During transaction execution the database may be inconsistent
- When the transaction is *committed*, the database must be consistent
- Two main issues to deal with:
  - *Failures, e.g. hardware failures and system crashes*
  - *Concurrency, for simultaneous execution of multiple transactions*

5.2 ACID Properties

- To preserve 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
- **Durability:** After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures
- **Example of Fund Transfer:** Let Ti be a transaction that transfers 50 from account A to B. This transaction can be illustrated as follows

☞ Transfer $50 from account *A* to *B*:

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

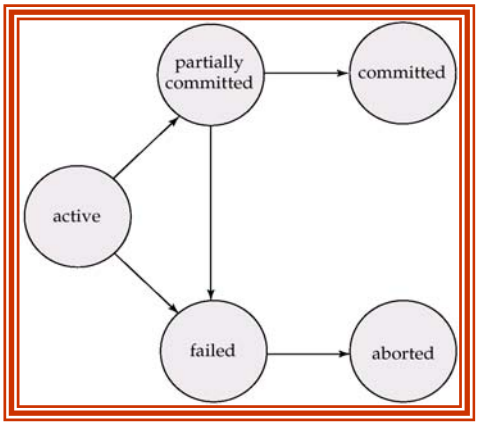
- ***Consistency:*** the sum of *A* and *B* is unchanged by the execution of the transaction.
- ***Atomicity:*** if the transaction fails after step 3 and before step 6, the system should ensure that its updates are not reflected in the database, else an inconsistency will result.
- ***Durability:*** once the user has been notified that the transaction has completed, the updates to the database by the transaction must persist despite failures.
- ***Isolation:*** between steps 3 and 6, no other transaction should access the partially updated database, or else it will see an inconsistent state (the sum *A* + *B* will be less than it should be).

5.3 Transaction and Schedules

- A transaction is seen by the DBMS as a series, or list of actions. We therefore establish a simple transaction model named as *transaction states*.

- **Transaction State:** *A transaction must be 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.
- **Committed,** after *successful completion*.
- **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.



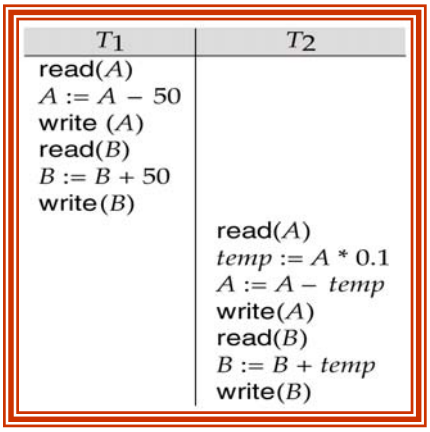
5.4 Concurrent Execution and Schedules

- **Concurrent execution:** executing transactions simultaneously has the following advantages:
  - *increased processor and disk utilization, leading to better throughput*
  - *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**: these are mechanisms to achieve isolation
  - *to control the interaction among the concurrent transactions in order to prevent them from destroying the consistency of the database*
- **Schedules**: sequences that indicate 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*

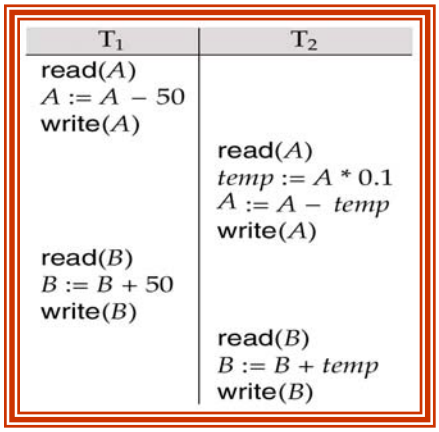
Example Schedules

- Let $T1$ transfer \$50 from A to B, and $T2$ transfer 10% of the balance from A to B. The following is a serial schedule (Schedule 1 in the text), in which $T1$ is followed by $T2$.



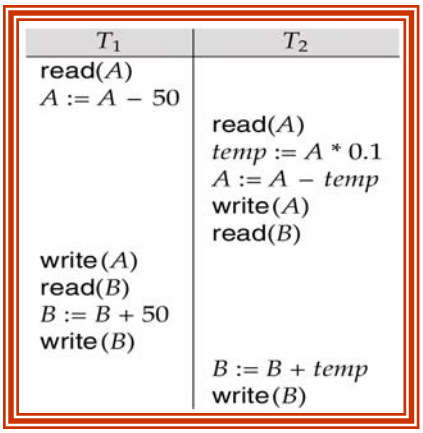
T_1	T_2
read(A)	
$A := A - 50$	
write (A)	
read(B)	
$B := B + 50$	
write(B)	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
	$B := B + temp$
	write(B)

- Let $T1$ and $T2$ be the transactions defined previously. The following schedule (Schedule 3 in the text) 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)

- The following concurrent schedule (Schedule 4 in the book) does not preserve the value of the the sum $A + B$



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)

■ Serializable Schedule

- A *serializable schedule* over a set S of committed transactions is a schedule whose effect on any consistent database is guaranteed to be identical to that of some complete serial schedule over S. i.e., even though the actions of transactions are interleaved, the result of executing transactions serially in different order may produce different results.
- **Example:** The schedule shown in the following figure is serializable.

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

Even though the actions of **T_1** and **T_2** are interleaved, the result of this schedule is equivalent to first running **T_1** entirely and then running and **T_2** entirely. Actually **T_1**'s read and write of B is not influenced by **T_2**'s actions on B, and the net effect is the same if these actions are the serial schedule First **T_1**, then **T_2**. This schedule is also serializable if first **T_2**, then **T_1**. Therefore if **T_1** and **T_2** are submitted concurrently to a DBMS, either of these two schedules could be chosen as first

- A DBMS might sometimes execute transactions which is not a serial execution i.e., not serializable.
- *This can be happen for two reasons:*
  - First the DBMS might use a concurrency control method that ensures the executed schedule itself.
  - Second, SQL gives programmers the authority to instruct the DBMS to choose non-serializable schedule.

■ Anomalies due to Interleaved execution

- There are three main situations when the actions of two transactions $\mathbf{T_1}$ and $\mathbf{T_2}$ conflict with each other in the interleaved execution on the same data object.
  - ***Write-Read (WR) Conflict:*** Reading Uncommitted data.
  - ***Read-Write (RW) Conflict:*** Unrepeatable Reads
  - ***Write-Write (WW) Conflict:*** Overwriting Uncommitted Data.

- **Reading Uncommitted Data (WR Conflicts)**
  - ***Dirty Read:*** The first source of anomalies is that a transaction $\mathbf{T_2}$ could read a database object A that has been just modified by another transaction $\mathbf{T_1}$, which has not yet committed, such a read is called a *dirty read.*
  - ***Example:*** Consider two transactions $\mathbf{T_1}$ and $\mathbf{T_2}$, where $\mathbf{T_1}$ stands for transferring $100 from A to B and $\mathbf{T_2}$ stands for incrementing both A and B by 6% of their accounts. Suppose that their actions are interleaved as follows:
    (i) $\mathbf{T_1}$ deducts $100 from account A, then immediately
    (ii) $\mathbf{T_2}$ reads accounts of A and B adds 6% interest to each, and then,
    (iii) $\mathbf{T_1}$ adds $100 to account B.
    This corresponding schedule is illustrated as follows:

$\mathbf{T_1}$	$\mathbf{T_2}$
R(A)	
A: = A -100	
W(A)	
	R(A)
	A: = A + 0.06 A
	W(A)
	R(B)
	B:= B+.06 B
	W(B)
R(B)	Commit
B: = B + 100	
W(B)	
Commit	

The problem here is $\mathbf{T_2}$ has added incorrect 6% interest to each A and B. Because before commitment that $100 is deducted from A, it has added 6% to account A before commitment that $100 is credited to B, it has added 6% to account B. thus, the result of this schedule is different from the result of the other schedule which is serializable: first T1 then $\mathbf{T_2}$.

- **Unrepeatable Reads (RW Conflicts)**
  - The second source of anomalies is that a transaction $\mathbf{T_2}$ could change the value of an object A that has been read by a transaction $\mathbf{T_1}$ and $\mathbf{T_1}$ is still in progress. This situation causes a problem that, if $\mathbf{T_1}$ tries to read the value of A again, it will get a different result, even though it has not modified A in the meantime. But, this situation could not arise in a serial execute of two transactions: this, it is called as ***unrepeatable read.***

- **Example:** Suppose that both T_1 and T_2 reads the same value of A, say 5. Then T_1 has incremented A value to 6 but before commitment as A value 6, T_2 has decremented A value from 5 to 4. Thus, instead of answer of A value as 5, i.e., from to 5 we got an answer 4 which is incorrect.

- **Overwriting Uncommitted Data (WW Conflicts)**

  - The third source of anomalies is that a transaction T_2 could overwrite the value of an object A, which has already been modified by a transaction T_1, while T_1 is still in progress.
  - **Example:** Suppose that A and B are two employees, and their salaries must be kept equal. Transaction T_1 sets their salaries to $1000 and transaction T_2 sets their salaries to $2000.

    The following interleaving of the actions T_1 and T_2 occurs:

    i) T_1 sets A's salary to $1000, at the same time, T_2 sets B's salary to $2000.

    ii) T_1 sets B's salary is set to to $2000, at the same time, T_2 sets A's salary to $2000.

    As a result A's salary is set to $2000 and B's salary is set to $1000, i.e., the result is not identical

  - **Blind-Write:** Neither transaction reads a value before writing it-such a write is called a ***blind-write.***

    The above example is the best example of blind write because T_1 and T_2 are concentrating only on writing but not on reading.

- **Schedules involving aborted Transactions**

- All transactions of aborted transactions are to be undone, and we can therefore imagine that they were never carried out to begin with.

- **Example:** Suppose that transaction T_1 deducts $100 from account A then immediately before committing A's new value the transaction T_2 reads the current values of accounts A and B and adds 6% interest to each, then commits, but incidentally T_1 is aborted. So, we get incorrect result of transaction T_2 because T_1 was aborted in the middle of the process and T_2 has taken incorrect value of A by T_1 and added 6%. We say that such a schedule is ***Unrecoverable Schedule.*** The corresponding schedule is shown as follows:

T_1	T_2
R(A)	
A: = A -100	
W(A)	
	R(A)
	A: = A + 0.06 A
	W(A)
	R(B)
	B:= B+.06 B
	W(B)
Abort	Commit

- Whereas, a recoverable schedule is one in which transactions read only the changes of committed transactions.

5.5 Lock-Based Concurrency Control

- Locking is a concurrent control technique used to ensure serializability.
- A lock disables occurs to data. There are two types of locks. A transaction needs to acquire a lock before performing a transaction.
- The read lock is known as *shared lock* and write lock is known as *exclusive lock*.
- A locking protocol is a set of rules that a transaction follows to attain serializability

■ Strict Two-Phase Locking (Strict 2PL) Protocol

- *The Strict 2PL has the following two rules:*
  - **Rule 1:** A transaction can read data only when it acquires a shared lock and can write data only when it acquires an exclusive lock on object.
  - **Rule 2:** A transaction should release the locks when it is completed.
- The entire request for the locks is maintain by DBMS without user intervention. A transaction is blocked until it gets a requested lock.
- If two transactions operate on two independent database objects then locking protocol allows such transactions. However if transactions operate on related data, locking protocol allows only the transaction which acquired lock.
- **Example:** Consider two transactions, $\mathbf{T_1}$ increments the values by 10 and $\mathbf{T_2}$ increments then by 20% of the values. If the initial values of database objects A and B are 10, then after serial execution they would have 24.

$\mathbf{T_1}$	$\mathbf{T_2}$
R(A) A: = A + 10 [A = 20] W(A)	
	R(A) A: = A + 0.20 A [A = 24] W(A) R(B) B: = B + 0.20 B [B = 12] W(B) Commit
R(B) B: = B + 10 [B = 14] W(B) Commit	

Such an interleaving would yield A = 24 and B = 14 as results.

Using Strict 2PL we can avoid such anomalies. When $\mathbf{T_1}$ wishes to operate on A, it has to first acquire the key on A. When $\mathbf{T_1}$ acquires the key no other transaction can interleave.

$\mathbf{T_1}$	$\mathbf{T_2}$
X(A) R(A) A: = A +10 W(A) X(B)	

R(B) B: = B + 10 W(B) Commit	
	X(A) R(A) A: = A + 0.20 A W(A) X(B) R(B) B:= B+.20 B W(B) Commit

- Using strict 2PL, the transaction first acquire a lock performs the action. However $\mathbf{T_2}$ cannot be interleaved and hence results in correct execution.

■ **Deadlocks**

- Deadlock is a situation where two or more transactions wait for locks held by other to be released.
- **Example:** $\mathbf{T_1}$ has lock on A and $\mathbf{T_2}$ has lock on B. If $\mathbf{T_1}$ requests for lock on B by holding lock on A. Similarly, $\mathbf{T_2}$ requests for lock on B. Either $\mathbf{T_1}$ nor $\mathbf{T_2}$ can continue with the execution. This is called deadlock.
- Deadlocks can be handled in three ways.
  i) Time-outs
  ii) Deadlock prevention
  iii) Deadlock detection and recovery

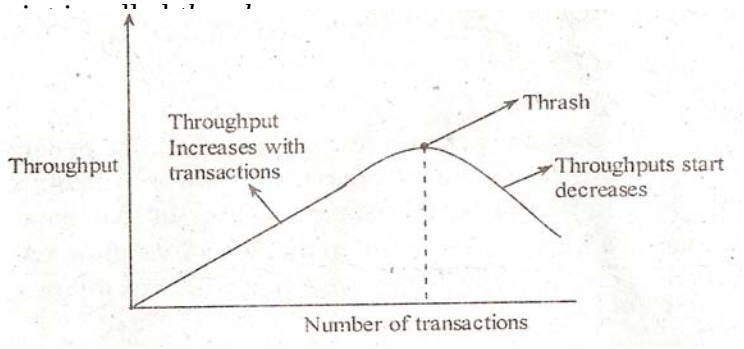
  - **Timeouts:** With this approach, the transaction waits for predefined amount of time before acquiring the lock. If the time-outs, DBMS assumes that so there could be a deadlock and aborts the transaction holding the object.
  - **Deadlock prevention:** DBMS looks ahead to determine a deadlock. If a deadlock is predicted, then it aborts the transaction and never allows a deadlock to occur. Two algorithms are used for the purpose.
    i) Wait-Dies
    ii) Wound-Wait.
  - **Deadlock Detection:** DBMS waits until a deadlock occurs and then takes measures to solve the deadlock problem. It constructs a wait-for graph (WFG) for the purpose.

5.6 Performance of Locking

- Long-based techniques use two methods to acquire serializabilty.
  i) Blocking
  ii) Aborting

- **Blocking:** A transaction is blocked until it gets a lock for operation. Deadlock is an extreme situation where a transaction blocks forever waiting for lock. This can be avoided by aborting the transaction.
- **Abort:** A transaction is forced to stop its execution and to restart.

- Practically, only 1% of transactions suffer from deadlocks and the transaction are aborted even less than 1%. Hence, there needs a wide consideration only on the delay introduced by blocking. These blocking delays in turn have an adverse effect on the throughput.
- Initially the throughput of the system increases with increasing number of transactions. This is because initially transactions are unlikely to conflict. As the transactions are increased, the throughput will not increase proportionally because of certain conflicts. As the transactions increase, there will reach a point when the system can no more handle the transactions and reduces the system throughput. This p



- If the system reaches the thrash point, the DBA takes effective measures to reduce the number of transactions.
- The following steps are taken to increase throughput:
  i) Reducing the situation where two objects request for same lock.
  ii) Each transaction should be allowed to hold the lock for a short period of time such that other transactions are not blocked for a long time.
  iii) Avoiding hot spots. A frequently accessed database object is known as hot spots. This hot spot reduces the system performance drastically.

5.7 Transaction Support in SQL

- We now consider what support SQL provides for users to specify transaction-level behavior.

■ Transaction Creation

- Transaction can be created by using certain SQL statements like:
  i) Select
  ii) Update
  iii) Create

- Once a transaction is created, it continues the execution of successive transactions until it finds a COMMIT and ROLLBACK statement.
- Two additional statements are provided to handle long-running transactions. They are:
  i) Save point
  ii) Chained Transaction

  - **Save points:** ROLLBACK undoes all the operation previously performed, but in long-running applications, it is always desired to roll back only till certain extent and save the other operations. This facility is provided by *SAVEPOINT*. This statement is generally used for decision making.

    Any number of save points can be defined in long-run application.
    The syntax for savepoint is given by:

    SAVEPOINT <savepoint name>

    There are two types of savepoints,

    i) Statement
    ii) Named

    The savepoints have two advantages:

    i) Initiating several transactions can do easily.
    ii) We can rollback to several savepoints

  - **Chained Transaction:** The statements make rollback and commit operations even simpler.

    Consider the following query:

    SELECT*
    FROM Students s
    WHERE s.category = “Sports”

    Suppose there are two transactions.

    i) Executes the above query.
    ii) The other transaction adds grace marks to students who are in sports category with greater than 60% of marks.

    It is difficult choice what should be locked. We can both lock the table for first transaction and provide an exclusive lock for second transaction. Though it provides for serializabilty, it provides poor concurrency.

    A better choice is to lock smaller objects. Instead of locking the complete table, it would be better to provide a shared lock only on the rows which satisfies the condition, “s.category = “Sports”. The other rows which do not have category as sports can run $\mathbf{T_1}$ and $\mathbf{T_2}$ concurrently.

It is a complicated task to choice the object that should be locked. DBMS obtains locks at various granularity'. Some transaction serve better if the shared lock is provided at row level and some few transactions serve better if lock is provided to complete table.

At times, the decision made for the previous 'students' example may also fail. Consider a third transaction $\mathbf{T_3}$, which adds a new row to students table with category as 'sports'. Since $\mathbf{T_1}$ is provided a shared lock, it does not stop when $\mathbf{T_2}$ will be executed. That is it may generate two different answers upon executing twice. This effect is called '*PHANTOM*" problem.

■ Transaction characteristics in SQL

- The three characteristics of a transaction in SQL are,

i) Access mode
ii) Diagnostics size
iii) Isolation level

▪ Access mode

This specifies the access a user can have on the data. If the access mode is READ ONLY, then it allows the user only to read the data. He cannot perform any data manipulation operations like insert, delete, update, create etc.

If the access mode is READ WRITE, then the user can read and perform various data manipulation operations. If access mode is READ ONLY, then exclusive locks are not required and hence increase concurrency.

▪ Diagnostics Size

It is the total number of errors.

▪ Isolation Level

Transaction isolation levels are,

i) Read Uncommitted
ii) Read committed
iii) Repeatable Read
iv) Serializable

♦ Read Uncommitted

A transaction can read the modifications made by an uncommitted transaction. Thus it becomes vulnerable to phantom problem.

An uncommitted transaction never obtains a shred lock before reading and it needs to have READ-ONLY access mode. As it cannot write, it does not need exclusive lock as well.

♦ Read Committed

A transaction reads the values only from committed transaction. It also does not allow other transactions to modify a value written by T.

However, the value T reads may be modified by some other transaction, hence prone to phantom problem.

Unlike READ UNCOMMITTED, a transaction obtains an exclusive lock before writing and a shared lock before reading objects.

♦ **Repeatable Read**

A transaction 'Y' reads only from a committed transaction. No other transaction is allowed to change a value, read or written by the transaction 'Y'. It sets locks same as a SERIALILZABLE transaction except index locking.

♦ **Serializable**

A Serializable transaction enjoys the highest degree of isolation. A Serializable transaction, 'T' reads only from a committed transaction and a value read or written by 'T' is not changed by other transactions until a T commits. T totally avoid phantom phenomenon.

It is the safest of all isolations.

5.8 Introduction to Crash Recovery

- A transaction may fail because of hardware or a software failure. It is the responsibility of the recovery manager to handle such failure and ensure 'atomicity' and 'durability'. It attains atomicity by undoing the uncommitted transactions. It also attains durability by retaining the committed transaction results even after system crashes.
- Under normal execution, transaction manger takes care of serializability by providing locks when requested. It writes the data to the disk o\in order to avoid loss of data after the system crash.

■ **Stealing Frames and forcing Pages**

- **Steal Approach:** The changes mode on an object '0' by a transaction is written onto the disk even before the transaction is committed. This is because another transaction wants a page to be loaded and buffer manger finds replacing frame with object '0' as optimal.
- **Force Approach:** All the objects in buffer pool are forced to disk after the transaction is committed.
- The simplistic implementation of recovery management is to use no-steal-force approach. With no steal, the data will not be written until a transaction is committed; hence there is no need of an undo operation. And force approach enables us to write data to the disk after committing; hence we need no perform redo operation.
- Through these approaches are simple, they have certain disadvantages.
    i) No steal approach requires a large buffer pool.
    ii) Force approach involves expensive I/O costs.

- If an object is frequently modified, then it needs to written onto the disk very frequently involve expensive I/O operation.
- Hence steal and no-force approach is implemented by recovery management. Using these techniques the page is not written onto disk when the modifying transaction is still active. And it does not force a page to be written onto disk, when transaction commits.

■ Recovery during Normal Execution

- The recovery manger, stores the modifications made onto a storage which does not react to system failures. Such storage is called stable storage.
- The modifications made to the data is called log. Recovery manager loads the log onto the stable storage before the new changes are made.
- If a transaction is aborted, then log enables recovery manager to undo the operations and redo the operations if it is committed.
- No force approach does not write data into the disk after the transaction is committed. If a transaction is committed just before a crash, then the modifications made by transaction will not be loaded onto disk. This modified data is reloaded from the stable storage.
- Steal approach enables to write data onto disk before committing. If a crash occurs before committing, then all the data modified onto the disk must be undone. This is done with help of log.

■ Overview of ARIES

- ARIES is an algorithm for recovering from crash, that uses no-force, steal approach.
- ARIES algorithm has three phases:
    - **Analyses Phase:** If analyses the buffer pool to identify the active transactions and dirty pages.
    - **Undo Phase:** If the modified data is loaded into disk before a transaction commits, then it must undo the modification in case of crash.
    - **Redo Phase:** It must restore the data which it was before the crash. This is done if the data modified by committed transaction is not loaded onto the disk.

■ Atomicity: Implementing Rollback

- It is important to recognize that the recovery subsystem is also responsible for executing the **ROLLBACK** command, which aborts a single transaction.
- Indeed, the logic (and code) involved in undoing a single transaction is identical to that used during the Undo phase in recovering form a system crash.

- All log records are stored in a linked list and to operate rollback, the linked list is accessed in reverse order.

---

6. CONCURRENCY CONTROL AND CRASH RECOVERY

6.1 Serializability

- Basic Assumption – Each transaction, on its own, preserves database consistency
  - *i.e. serial execution 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 *conflict serializability* and *view serializability*
- Simplifying assumptions:
  - *ignore operations other than read and write instructions*
  - *assume that transactions may perform arbitrary computations on data in local buffers between reads and writes*
  - *simplified schedules consist only of reads and writes*

■ Conflict Serializability

- Instructions *li* and *lj* of transactions *Ti* and *Tj* respectively, **conflict** if and only if there exists some item *Q* accessed by both *li* and *lj*, and at least one of these instructions wrote *Q*.
  1. *li* = **read**(*Q*), *lj* = **read**(*Q*). *li* and *lj* don't conflict.
  2. *li* = **read**(*Q*), *lj* = **write**(*Q*). They conflict.
  3. *li* = **write**(*Q*), *lj* = **read**(*Q*). They conflict
  4. *li* = **write**(*Q*), *lj* = **write**(*Q*). They conflict
- Intuitively, a conflict between *li* and *lj* forces a (logical) temporal order between them

- If *li* and *lj* are consecutive in a schedule and they do not conflict, their results would remain the same even if they had been interchanged in the ordering
- 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
- ***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 >.

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

- Schedule 3 below can be transformed into Schedule 1, a serial schedule where *T*2 follows *T*1, by series of swaps of non-conflicting instructions.
  Therefore Schedule 3 is conflict serializable.

■ **View Serializability**

- Let *S* and *S´* be two schedules with the same set of transactions. *S* and *S´* are *view equivalent* if the following three conditions are met, where *Q* is a data item and *Ti* is a transaction:
  1. If *Ti* reads the initial value of *Q* in schedule *S*, then *Ti* must, in schedule *S´*, also read the initial value of *Q*
  2. If *Ti* executes read(*Q*) in schedule *S*, and that value was produced by transaction *Tj* (if any), then transaction *Ti* must in schedule *S´* also read the value of *Q* that was produced by transaction *Tj*
  3. The transaction (if any) that performs the final write(*Q*) operation in schedule *S* (for any data item *Q*) must perform the final write(*Q*) operation in schedule *S´*

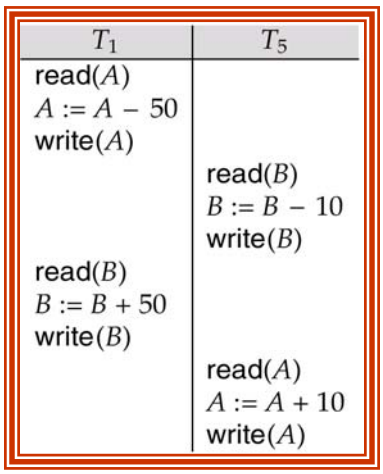
  NB: View equivalence is also based purely on **reads** and **writes**
- A schedule *S* is *view serializable* it is view equivalent to a serial schedule
- Every conflict serializable schedule is also view serializable
- Schedule 9 (from book) — a schedule which is view-serializable but *not* conflict 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*

■ Other Notions of Serializability

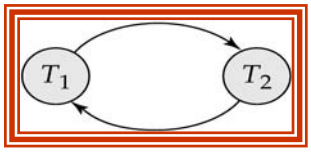
- This schedule produces the same outcome as the serial schedule < *T*1, *T*5 >
- However it is not conflict equivalent or view equivalent to it
- Determining such equivalence requires analysis of operations other than read and write



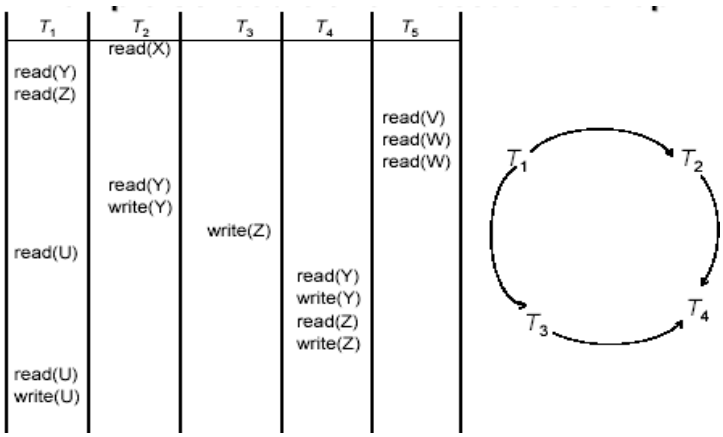
T_1	T_5
read(A)	
$A := A - 50$	
write(A)	
	read(B)
	$B := B - 10$
	write(B)
read(B)	
$B := B + 50$	
write(B)	
	read(A)
	$A := A + 10$
	write(A)

■ Testing for Serializability

- Consider some schedule of a set of transactions *T*1, *T*2, ..., *Tn*
- **Precedence graph***:* a directed graph where the vertices are transaction names
- We draw an arc from *Ti* to *Tj* if the two transaction conflict, and *Ti* accessed the data item before *Tj*
- We may label the arc by the item that was accessed
- **Example:**



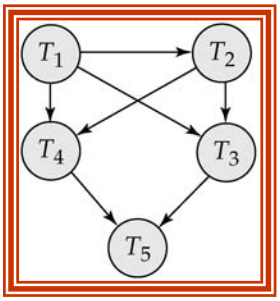
- **Example Schedule and Precedence Graph**



- A schedule is **conflict serializable** if and only if its precedence graph is acyclic

- Cycle-detection algorithms exist which take order $n2$ time, where n is the number of vertices in the graph
- 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 this graph is $T2 \rightarrow T1 \rightarrow T3 \rightarrow T4 \rightarrow T5$

- The precedence graph test for conflict serializability must be modified to apply to a test for **view serializability**
  - The problem of checking if a schedule is view serializable is *NP*-complete. Thus existence of an efficient algorithm is unlikely. However practical algorithms that just check some *sufficient conditions* for view serializability can still be used



Example of an acyclic precedence graph

Concurrency Control vs. Serializability Tests

- Goal – to develop concurrency control protocols that will ensure serializability
- These protocols will impose a discipline that avoids nonseralizable schedules
- A common concurrency control protocol uses *locks*
  - *while one transaction is accessing a data item, no other transaction can modify it*
  - *require a transaction to lock the item before accessing it*
  - *two standard lock modes are "shared" (read-only) and "exclusive" (read-write)*

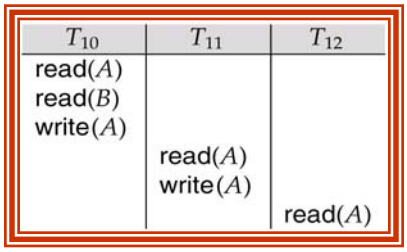
6.2 Recoverability

- Need to address the effect of transaction failures on concurrently running transactions.

- *Recoverable schedule:* if a transaction *Tj* reads a data item previously written by a transaction *Ti* , the commit operation of *Ti* appears before the commit operation of *Tj*
- The following schedule (Schedule 11) is not recoverable if *T9* commits immediately after the read

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

- 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
- *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)

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

6.3 Lock Management

- The data items must be accessed in a mutually exclusive manner in order to ensure serializability i.e., a data item can be accessed by only one transaction at a time. This can be accomplished by allowing the transaction to access a dataitem only if it is holding a lock o that data item.
- A lock manager is a component of the DBMS that keeps track of the locks issued to the transactions. It maintains a hash tables with the data object identifier as a key called Lock Table.
- It also maintains a unique entry for each transaction in a transaction table (TT) and each entry contains a pointer to a series of locks held by the transaction.

- **A Lock Table Entry:** While can either be a page or a record for an object contains the following information:
    i) The number of transactions which holds a lock on the object currently which can be more than one in shared mode.
    ii) The nature of the lock which can either be shared or exclusive and
    iii) A pointer to a queue holding lock requests.

- **Locking Modes**
    i) **Shared:** A transaction T-I can read the dataitem P but cannot write it, if it is holding a shared, mode lock. It is denoted by S.
    ii) **Exclusive:** A transaction T-I can both read and write the dataitem P if it is holding an exclusive-mode lock. It is denoted by X.

- **Implementing Lock and Unlock Requests**

- According to the strict 2PL, a transaction must obtain and hold a shred (S) or exclusive (X) lock on some object ‘O’ before it reads or writes an object ‘O’.
- A transaction T_1 can acquire the needed locks by sending a lock request to the lock manager in this manner.

    1. If a request is made for a shared lock and the request queue is empty and further the object is not locked currently in an exclusive mode then the lock manager accepts the lock requests and grant the needed lock and updates the entry for an object in the lock table.
    2. If a request is made for an exclusive lock(X) and none of the transactions is currently holding a lock on the object i.e., request queue is empty, then the lock manager grants the lock and updates the corresponding entry in the lock table.
    3. If the locks are not currently available then the request is added to the request – queue and the (requesting) corresponding transaction is suspended.

- A transaction releases all the acquired locks on its abortion or commitment. Once a lock is released a lock table entry for an object is updated by the lock manager and grants the lock to the requesting transaction present at the head of the queue.
- If more than one request is made for the shared lock then all the requests can be granted together. All the pending lock requests are queued.
- If transaction T_1 acquires a shared lock on object ‘A’ and if transaction T_2 requests for exclusive requests are placed in the queue, and a lock is granted when its predecessor releases the lock. Hence T2 is granted the lock when T1 unlocks.
- **Atomicity Assurance in Locking and Unlocking**

    - To ensure the atomicity of lock and unlock operations access to the lock-table can be achieved by using a semaphore which is an OS synchronization mechanism.
    - When a transaction issues an exclusive lock (X) request, the lock manager checks finds and grant the request if no other transaction is holding a lock on an object.

6.4 Lock Conversions

- If a transaction which is currently holding a shared lock on some object ‘O’ wants to obtain an exclusive lock on ‘O’ then this “upgrade lock” request is handled by granting the X on ‘O’ if none of the transaction holds ‘S’ on ‘O’ and no other request is present at the head of the queue. Otherwise the two transactions are deadlocked (if they request for the X on the same object).
- This deadlocks can be avoided by acquiring X locks the start-up and then downgrading them to S locks.

- **Advantage of lock-down Grading:** It improves the overall performance i.e, throughput by reducing deadlocks.
- **Disadvantage:** It reduces concurrency by acquiring write locks when they not actually needed. This drawback can be reduced by using update lock which is sent initially and prevents conflicts between the read operations else it is downgraded to a shared lock, if not needed. In case of object updates if is upgraded to X lock thereby preventing deadlocks.

6.5 Dealing with Deadlocks

- **Deadlock definition:** Deadlock is a situation where one transaction is waiting for another transaction to release locks before it can proceed.
- **Example:** Suppose a transaction T_1 holds an exclusive lock on some dataitem P and transaction T_2 holds an exclusive lock on data item Q. Now, T_1 requests an exclusive lock on Q and T_2 requests for an exclusive lock on P and are queued. So, T_1 is waiting for T-2 to release its lock and T_2 is waiting for T_1 to release its lock leading to a situation called deadlock where neither of the two transactions can proceed.
- The DBMS is responsible for the detection and prevention of such deadlocks.

■ Deadlock Prevention

- As the saying goes, prevention is better than cure, it is always better to prevent a deadlock rather than waiting for it to occur and then taking measures to avoid deadlock. It can be prevented by prioritizing the transactions.
- If a transaction T_i requests a lock which held by some other transaction T_j then the lock manager can use one of these policies.
- **Wait-Die:** Wait- Die is a non preemptive scheme. As the name specifies, the transaction either waits for the lock or dies. The decision on whether to wait or die is made based on the time stamp. A transaction T_2 time stamp is greater than the time stamp value of a transaction T_1 currently using the lock, then T_2 cannot wait and is rolled back.
    - **Example:** T_3 with the time stamp values as 10, 20, 30 respectively. If a transaction T_1 requests a data item which is held by T_2, then it is allowed to wait. If T_3 requests a dataitem which is held by T_2, then T_3 will be rolled back (dies).
- **Wound – Wait:** It is a preemptive scheme wherein if a transaction T_i requests a dataitem which is currently under the control of transaction T_j, it is allowed to wait if its time stamp value is greater than that of T_j, else T_j is rolled back.
    - **Example:** Consider three transactions T_1, T_2 and T_3 with the time stamp values as, respectively. If a transaction T_1 requests a dataitem which is held by T_2 then the dataitem will be preempted from T_2 and T_2 is rolled back. Also, if T_3 requests a dataitem which is held by T_2 then T_3 is allowed to wait since its time stamp is greater than that of transaction Tj.

- **Advantages of wait-die Scheme**

  i) No occurrence of deadlock, because lower priority transactions need not have to wait for higher priority transactions.

  ii) It avoids starvation (i.e, no transaction is allowed to progress and rolled back repeatedly).

- **Disadvantage:** Unnecessary rollbacks may occur

- **Advantages of Wound-wait scheme**

  1. Deadlock never occurs because higher priority transactions needn't have to wait for lower priority transaction.
  2. It also avoids starvation.

- **Disadvantage:** Unnecessary rollbacks may occur.

- **Note:** When a transaction is aborted and restarted again it should get the same time stamp as before abortion, otherwise reassignment of time stamps causes each transaction to become the oldest transaction.

- **Conservative 2PL:** It is a variant of 2PL that can prevent deadlock between the transactions by acquiring all the needed locks at the time of their beginning or blocking, while waiting for these locks to be available. This scheme ensures that no deadlocks can occur because a transaction acquires all the locks needed for its execution.

- **Deadlock Detection:** In order to detect and recover from the deadlocks a system must perform the following operations.

  1. It should maintain the information about the allocation of the data items to different transactions and the outstanding requests.
  2. It should provide an algorithm that determines the occurrence of deadlock.
  3. Whenever a deadlock has been detected find out the ways to recover from it. For deadlock detection the lock manager maintains a structure called a waits for graph in which nodes represents the active transactions and the are from T_i to T_j ($T_i \rightarrow T_j$) represents that T_k is waiting for T_j to release a lock. These edges are added to the graph by the lock manager whenever a transaction requests a lock and are removed when it grants lock requests.

- **Example:** Consider the wait – for graph as shown in figure (i)

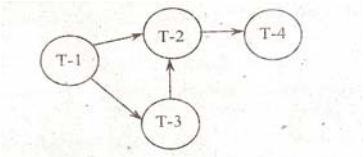


Figure: Wait for Graph with no Cycle

The following points must be noted;

i) Transaction T_1 is waiting for the transactions T_2 and T_3.

ii) Transaction T_3 is waiting for the transaction T_2.

iii) Transaction T_2 is waiting for the transaction T_4 Deadlock cannot occur as there are no cycles is the graph.

Further if a transaction T_4 is requesting for an item held by T_3 then the edge $T_4 \rightarrow T_3$ is added to the wait-for graph resulting in a new state with a cycle.

$T_2 \rightarrow T_4 \rightarrow T_3 \rightarrow T_2$ as shown below

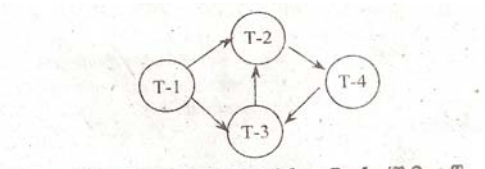


Figure: (ii) Wait-for Graph with a Cycle (T-2 → T-4 → T-3 → T-2)

The wait-for graph is checked periodically for the presence of cycles which represents deadlock. When a cycle is found, the deadlock is resolved by aborting corresponding transaction on a cycle thereby releasing its locks.

6.6 Specialized Locking Techniques

- The database objects are not always constant in the real world.
- For example, the number of customers of a bank will not be constant. Some may withdraw their accounts and some may create new accounts. Thus the number of database objects may grow of shrink. This varying number of database objects reads to a problem called Phantom problem.
- The performance of database can be enhanced by using the protocol which clearly explains the relationship between the objects. Two such cases include
  - Tree structured index locking
  - Collection of objects and containment relationship locking

■ The Phantom Problem and the Dynamic Database Phantom Problem

- Suppose a transaction say T1 retrieve the rows of a record that satisfy certain condition. Another transaction say T2 which is running concurrently inserts another records satisfying the same condition. If T1 retrieves the rows again, it will have a row which wasn't present previously. This differing result of same query is called phantom problem.
- Consider a student database. The principal can retrieve the records of the students at any time. The administrator creates a new record for each new student's admission say transaction T1 returns the details of students and transaction, T1 adds a new record to the database.
- Suppose the principal is willing to see the records of students who have scored the highest marks. Transaction T1 acquires shared lock runs to find the student with highest score. Suppose, the highest score was found to be 500. However the administrator provides admission to a new student who secures 520 marks. And hence T2 acquires exclusive locks and updates the database.
- Now if T2 runs before T1, principal would view 520 marks. On the other hand if T1 runs before T2, principal views 500 marks.

■ Concurrency Control in B+ Trees

- **B+ Tree:** B+ tree is a balanced tree which has an equi-length for all paths from root to the leaf.
- Indices provide high level of abstraction and hence concurrency control in B+ trees ignores the index structure.
- The concurrent control in B+ trees is based on locking. The highest rend node is locked and the complete tree is traversed. Locking overhead is negligible when efficient locking protocols are used.
- Searches acquire a shared lock on the root node and proceeds further. The rood node unlocks when its child takes up the lock.
- We can also obtain an exclusive lock on all the nodes of the tree. However for insertions. Exclusive lock is required only in cases when the child node is full hence this technique is not implemented.
- The efficient technique assigns a shared lock to the root node and proceeds further by assigning shared lock to the child. If the child is not full, the lock on its parent is released. However if the child is full, the lock on the parent is not released. This is called "crabbing" or "lock-coupling"

■ Multiple Granularity Locking

- It is a technique used for locking complex objects (object within an object)
- **Example:** A University contains several colleges, each college has many courses and each course has several students. A student can select a preferred course in a particular college.
- Similarly a database contains several files and each file contains many pages which in turn is a gap of records. This is called containment hierarchy which can be represented as a tree of objects. A transaction can obtain a lock on a selected item just as a student chooses a course of his choice locking a node in a tree involves locking all its children.
- Apart from S and X locks multiple granularity locking uses:
  (i) Intension shared (IS) and
  (ii) Intension exclusive (IX).
- Conflicts between the locks can be explained using the flowing table

	S	X
IS	Doesn't conflicts	Conflicts
IX	Conflicts	Conflicts

- Thus, if a transaction acquires on X or S lock on some node 'i', it must first lock its parents either in IS or IX.

- A transaction must obtain S and IX clock in order to read a file followed by the modification of some of its records or it can also acquire SIX lock.

SIX = S+IX

- Locks acquisition is a top-down approach whereas lock releasing is a bottom-up approach (leaf-root). Multiple granularity locking is used in conjunction with 2PL, this ensures serializability as 2PL predicts when to release the locks.

- "Granularity" of the locking is an important issue. Hence fine granularity locks are acquired in the beginning and after making some requests, the next higher level granularity locks can be obtained. This phenomenon is called lock escalation.

6.7 Concurrency control without locking.

- In DBMS the concurrency can be controlled without locking also, by using the following techniques:
    - Optimistic Concurrency Control
    - Timestamp-Based Concurrency Control
    - Multiversion Concurrency Control

■ **Optimistic Concurrency Control**

- In this it is assumed that the conflicts between the transactions are occasional hence there is no need for locking and time stamping.

- In this technique when a transaction reaches its COMMIT step, it is checked for the presence of conflicts. On occurrence of it the transaction must be rolled back and restarted. This happens rarely as there are very few conflicts.

- The transactions proceed in optimistic concurrency control in three phases as follows:

    - **Read:** The transaction executes, reading values from the database and writing to private workspace.

    - **Validation:** If the transaction decides that it wants to commit, the DBMS checks whether the transaction could have conflicts with any other currently executing transaction. If there is possibility to conflict, the transaction is aborted, and its private workspace is cleared and it is restarted.

    - **Write:** If validation determines that there are no possible conflicts, the changes to data objects made by the transaction in its private workspace are copied into the database.
- Remember, if there are few conflicts, then validation can be done efficiently, this approach should lead to better performance than locking. But, if there are many conflicts, the cost of repeatedly restarting transaction hurts performance.

- Thus, each transaction T_i is assigned a time stamp TS (T_i) at the beginning of its validation phase, and the validation criterion checks whether the timestamp ordering of transactions is an equivalent serial order transaction.
- For every pair of transactions T_i and T_j such that TS(T_i)< TS(T_j), one of the following validation conditions must hold:
    1. Ti completes (all three phases) before T_j begins.
    2. Ti completes before T_j starts its write phase, and Ti does not write any database objects read by T_j.
    3. Ti completes its Read phase before T_j completes its Read phase, and Ti does not write any database object, that is either read or written by T_j.
- To validate T_j, we must check to see that one of these conditions holds with respect to committed transaction T_i such that TS(T_i) < TS(T_j). Moreover, each condition ensures that T_j's modifications are not visible to T_i.
- The first condition allows T_j to see some of T_i's changes, but they execute completely in serial order with respect to each other.
- The second condition allows T_j to read objects while Ti is still modifying objects, but there is objects written by T_i, all of T_i's writes precede all of T_j's writes.
- The third condition allows T_i and T_j to write objects at the same time and thus, have even more overlap in time than the second condition, but the sets of objects written by the two transactions cannot overlap.
- Thus, no RW, WR, or WW conflicts are possible if any of these three conditions is met and the concurrency is controlled without locking through an optimistic concurrency control approach.

■ Time Stamp –Based Concurrency control Time Stamp

- In optimistic concurrency control, a timestamp ordering is imposed on transactions and validation checks that all conflicting actions occurred in the same order.

- So, each transaction can be assigned a timestamp at startup, and we can ensure, at execution time, that if an action a_i of transaction T_i conflicts with action a_j of transaction T_j, a_i occurs before a_j if TS(T_i)< TS(T_j). If an action violates this ordering, the transaction is aborted and restarted.

- The timestamp concurrency control is implemented by giving every database object O a read timestamp RTS(O) and a write timestamp WTS(O). If transaction T wants to read object O, and TS(T)<WTS(O), the order of the read with respect to the most recent write on O would violate the timestamp order between this transaction and the writer. Therefore, T is aborted and restarted with anew, larger timestamp, if TS(T)> WTS(O), T reads O, and RTS(O) is set to the larger of RTS(O) and TS(T).

- Now consider what happens when transaction T wants to write object O:

1. If TS(T)<RTS(O), the write actions conflicts with the most recent read action of O, and T is therefore aborted and restarted.
2. If TS(T)<WTS(O), a simple approach would be to abort T because it writes action conflicts with the most recent write of O and is out of timestamp order. However, we can safely ignore such writes and continue. Ignoring outdated writes is called the Thomas Write Rule.
3. Otherwise, T writes O and WTS(O) is set to TS(T).

- **Thomas's Write Rule:** As roll back restart doesn't occur in the time stamp method, Thomas's write rule ahs been used.

    1. When transaction i wants to write the value of some data item 'D' which is already being read by some younger transaction then it is not possible for transaction i to write its value hence, it must be aborted, rolled back and restarted with a new time stamp value.

    2. When a transaction i wants to write a new value to some data item 'D' on which the write operation has already been applied by some younger transaction then the write operation requested by the transaction i is neglected and is allowed to proceed with its normal execution.

    3. Whereas in other operations a transaction; is permitted to continue with its execution and its write time stamp is changed along with the change in transactional time stamp.

- Thus, by using Thomas's write rule it would be possible to obtain both serializable and recoverable schedules.

- The time stamp protocol just presented above, permits schedules that are not recoverable, as illustrated in the following figure:

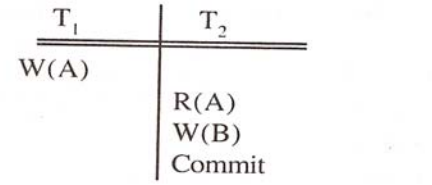


Figure : An Unrecoverable Schedule

- If TS(T_1) = and TS(T_2)=2, this schedule is permitted by the time stamp protocol (with or without the Thomas write Rule). This timestamp protocol can be modified to disallow such schedules by buffering all write actions until the transaction commits.

- In the above example, when T_1 wants to write A, WTS(A) is updated to reflect this action, but the change to A is not carried out immediately; instead, it is recorded in a private workspace, or buffer. When T2 wants to read A, then its timestamp is compared with WTS (A), and the read is seen to be permissible. However, T_2 is blocked until T_1 completes. If T_1 commits, its change to A is

copies from the buffer; otherwise, the changes in the buffer are discarded. T_2 is then allowed to read A.

- This blocking of T_2 is similar to the effect of T_1 obtaining an exclusive lock on A. With this modification, the timestamp protocol permits some schedules which are not permitted by 2PL at all.

- As recoverability is essential, such a modification must be used for the timestamp protocol.

■ **Multiversion concurrency control**

- Multiversion concurrency control protocol represents another way of using timestamps, assigned at startup time, to achieve serializability. The goal of this protocol is to ensure that a transaction never has to wait to read a database object and the idea to maintain several versions of each database object with a write timestamp and let transaction T_i read the most recent version whose timestamp precedes TS (T_i).

- If transaction Ti wants to write an object, we must ensure that the object has not already been read by some other transaction T_j such that $TS(T_i)<TS(T_j)$. If we allow T_i to write such an object, its change should be seen by T_j for serializability, but T_j which read the object at sometime in the past, will not see T_i's change.

- This condition can be checked, byu associating read timestamp with every object and whenever a transaction reads the object, the read timestamp is set to the maximum of the current read is aborted and restarted with a new, larger timestamp. Otherwise, Ti creates a new version of O and sets the read and write timestamps of the new version to TS(Ti).

- The drawbacks of this scheme are same, as timestamp concurrency control and in addition, there is the cost of maintaining versions. On the other hand, reads are never blocked, which can be important for workloads dominated by transaction that only read values from the database.

6.8 Introduction to crash recovery

- A computer system, like any other mechanical or electrical device, is subject to failure. There are many causes of such failure, such as disk crash, power failure, software error, etc. In each of these cases, information may be lost. Therefore, the database system maintains an integral part known as recovery manager, which is responsible for the restorage of the database to a consistent state that existed prior to the occurrence of the failures.

- The recovery manager of a DBMS is responsible for ensuring transaction atomicity and durability. It ensures atomicity by undoing the action of transactions, that do not commit, and durability by making sure that all actions of

committed transactions survive system crashes (example: the central part of the system is dumped by an error) and media failure (example: a disk is corrupted).

- When a DBMS is restarted after crashes, the recovery manager is given control and must bring the database to a consistent state. The recovery manager is also responsible for undoing the actions of aborted transactions.

■ **System/transaction failures**

- There are two types of errors that may cause a transaction failure:

1. **Logical Error:** The transaction can no longer continue with its normal execution with some internal conditions such as bad input, data not found, overflow or resource limits exceeded.
2. **System Error:** The system has entered an undesirable state (example: deadlock), as a result of which a transaction cannot continue with its normal execution. This transaction can be re-executed at a later time.
3. **System Crash:** there is a hardware failure or an error in the database software or the operating system, the causes the loss of the content of temporary storage and brings transaction processing to a halt. The content of permanent storage remains same and is not corrupted.
4. **Disk Failure:** A disk block loses its content as result of either a head crash or failure brings a data transfer operating. Copies of the data on other disks, or backups on tapes, are used to recover from the failure.

■ **Arises recovery algorithm**

- ARIES stands for “Algorithm for recovery and isolation exploiting semantics”. ARIES is a recovery algorithm designed to work with a steal, no-force approach.

- If a steal policy is in effect, the change made to an object in the buffer pool by a transaction can be written to disk before the transaction commits. This might be because some other transaction might “steal” the buffer page presently occupied by an uncommitted transaction.

- If no-force policy is in effect, when a transaction commits, we need not ensure that all the changes it has made to objects in the buffer pool are immediately forced to disk.

- ARIES has been implemented (“to varying degrees”) in several commercial and experimental systems including in particular DB2.

- When the recovery manager is invoked after a crash, restart proceeds in three phases:
  1. Analysis: the analysis phase identified dirty pages (i.e., pages that contain changes that have not been written to disk) in the buffer pool and active transactions at the time of the crash.
  2. Redo: The redo phase repeats all actions, starting from an appropriate point in the log (log is a history of actions executed by the DBMS) and restores the database state to what it was at the time of crash.

3. Undo: the undo phase undoes the actions of transactions that did not commit, so that the database does only the actions of committed transactions.

- *Three main principles lie behind the ARIES recovery algorithm as follows*:

a) Write-Ahead Logging: Any change to a database object is first recode din the log; the record in the log must be written to stable storage before the change to the database object is written to disk.

b) Repeating History during read: On restart following a crash, ARIES retraces all actions of the DBMS before the crash and brings the system back to the exact state that it was in at the time of the crash. Then, it undoes the actions of transactions still active at the time of the crash.

c) Logging changes during Undo: changes made to the database while undoing a transaction are logged to ensure such an action is not repeated in the event of repeated (failures causing) restarts.

- The second principle distinguished ARIES from other recovery algorithms and is the basis for much of its simplicity. Therefore, ARIES can support concurrency control protocols that involve locks of finer granularity locks (example; record-level locks) than a page.

- The second and third points are also important in dealing with operations where redoing and undoing the operation are not exact inverses of each other.

6.9 Log recovery

- The log is a history of actions executed by the DBMS, sometimes also called as the trail or journal. Physically, the log is a file of records stored in stable storage, which can tolerate crashed; this can be achieved by maintaining two or more copies of the log in different locations, so that chance of losing all copies of the log is negligibly small.

- The log tail is the most recent portion of the log, which is kept in main memory and is time to time force to store safely. This way, log records and data records are written to disk on the same pages.

- Each and every log record is given a unique id called the log sequence number (LSN). As with any record id, we can fetch a log record with one disk access given the LSN. Moreover, LSN's should be assigned in increasing order; this property is required for the ARIES recovery algorithm. If the log is a sequential file growing indefinitely, the LSN can simply be the address of the first byte of the log record.

- Thus for recovery purposes, every page in the database contains the LSN of the most receipt log record that describes a change to this page. This LSN is called the page LSN.

- A log record is written for each of the following actions:

(i) **Updating a page:** After modifying the page, an update type record is appended to the log tail. The page LSN of the page is then set to the LSN of the update log record. The page must be pinned in the buffer pool while these actions are carried out.

(ii) **Commit:** When a transaction decided to commit, it force-writes a commit type log record containing the transaction id i.e, the log record is appended to the log, and the log tail is written to stable storage including the commit record. The transaction is considered to have committed at the instant, that is commit log record is written to stable storage.

(iii) **Abort:** When a transaction is aborted, an abort type log record containing the transaction id is appended to the log.

(iv) **End:** when a transaction is aborted or committed, some additional actions must be taken beyond writing the abort or commit log record. After all these additional steps are completed, an end type log record containing the transaction id is appended to the log.

(v) **Undoing an update:** when a transaction is rolled back (because the transaction is aborted during crash), its updates are undone. When the actions described by an update log record is undone, a compensation log record, or CLR, is written.

- Every log record has certain fields: prevLSN, transID, and type. The set of all log records for a given transaction is maintained as a linked list going back in time, using the prevLSN field; this list must be updated whenever a log record is added. The transID field is the id of the transaction generating the log record, and the type filed indicates the type of the log record.

■ **Update Log Records:**

- :

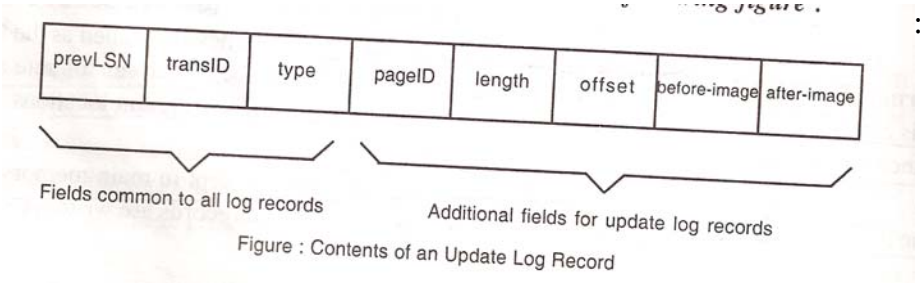


Figure : Contents of an Update Log Record

- Additional fields for update log record that prevLSN, transID and type are the page id length, offset, before-image and after –image.

- Where the page ID field is the page id of the modified page; the length in bytes and the offset of the change are also included. The before-image is the value of the changed bytes before the change; the after-image is the value after the change. An update log record that contains the both before-image and after-image can be sued to redo the change and undo the change. A read-only update log record

contains just the after – image; similarly an undo-only update log record contains just the before – image.

■ **Compensation Log Records (CLR)**

- A compensation log record (CLR) is written just before the change recorded in an update log record U is undone. Such an undo can happen during normal execution is aborted. A compensation log record C describes the action taken to undo the actions recorded in the corresponding update log record and is appended to the log tail just like any other log record. The compensation log record C also contains a field called undo/next/LSN, which is the LSN of the next log record that is to be undone for the transaction that wrote update record U; this field in c is set to the value of prevLSN in U.

- A CLR describes an action that will never be undone, i.e., we never undo undo actions. The reason is simple. An update log record describes a change may by a transaction during normal execution and the transaction may be aborted, whereas a CLR describes an action taken to rollback, a transaction for which the decision to abort has already been made. Therefore, the transaction must be rolled back, and the undo action described by the CLR is definitely required. This observation is very useful because it bounds the amount of space needed for the log during restart from a crash: The number of CLRs that can be written during Undo is no more than the m\number of update log records for active transaction at the time of the crash.

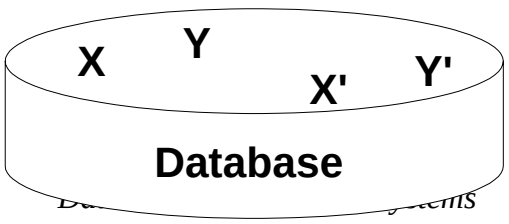
- A CLR may be written to stable storage but the undo action it describes may be written to disk. When the system crashes again. In this case, the undo action described in the CLR is reapplied during the Redo phase, just like the actions described in Update log records. For these reasons, a CLR contains the information needed to reapply, or redo, the change described but not to reverse it.

■ **Shadow paging**

- An alternative to log-based crash-recovery techniques is shadow paging. Shadow paging may require fewer disk accesses when compared to the log-based recovery method.

- The database is partitioned into some number of fixed-length blocks, which are referred as pages. The term page is borrowed from operating systems, since we are using a paging scheme for memory management.

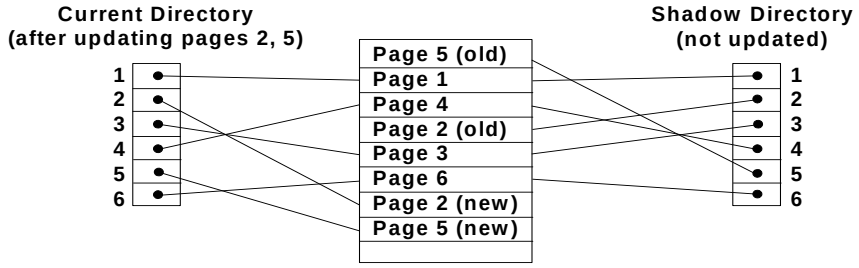
- Assume that there are n pages, numbered from 1 to n. (n may be in hundreds or thousands). Theses pages are not stored in any particular order on disk. But, there is a way to find the i^{th} page of the database for any given i, by using a page table, as illustrated in the following figure.



X and Y: Shadow copies of data items
X` and Y`: Current copies of data items

- The page table has n entries-one for each database page. Each entry contains a pointer to page on disk. The first entry contains a pointer to the first page of the database, the second entry may contain a pointer to the second page, and so on.

- The key idea behind the shadow-paging technique is to maintain two page tables during the life of a transaction: the current page table and the shadow page table, as illustrated in the following figure:



- When a transaction starts, both pages are identical. The shadow page table is never changed over the duration of the transaction. The current page table may be changed when a transaction performs a write operation. Thus, all input and output operations use the current page table to locate the database pages on disk.

☞ **Drawbacks of shadow –paging**

1. **Commit overhead:** the commit of a single transaction using shadow paging requires multiple blocks to the output-the actual data blocks, the current page table, and the disk address of the current page table.
2. **Data fragmentation:** we consider strategies to ensure locality-that is, to keep related database pages close physically on the disk. Because, locality allows for faster data transfer. But, shadow paging causes database pages to change location when they are updated.
3. **Garbage collection:** Each time that a transaction commits, the database pages containing the old version of data changed by the transaction become inaccessible. Such pages are considered garbage, since they are not a part of free space and do not contain usable information. Garbage may be created also as a side effect of crashed. Periodically, it is necessary to find all the garbage pages and add them to the list of free pages. This process is called garbage collection.

- In addition to the drawbacks of shadow paging, even shadow paging is more difficult than logging to adapt to systems that allow several transactions to execute concurrently.

6.10 Transaction table recovery

- This table contains one entry for each active transaction. The entry contains the transaction id, the status, and a field called last LSN, which is the LSN of the most recent log record for this transaction. The status of a transaction can be that which is in progress or committed or aborted.

- During normal operation, this table is maintained by the transaction manger and during restart after a crash; this table is reconstructed in the Analysis phase of restart.

6.11 Dirty page table recovery

- This table contains one entry for each dirty page in the buffer pool, i.e., each page with changes not yet reflected on disk. The entry contains a field rec LSN, which is the LSN of the first log record that caused the page to become dirty. Thus, this LSN identifies the earliest log record that might have to be redone for this page during restart from a crash.

- During normal execution, this table is maintained by the buffer manager and during restart after a crash; this table is reconstructed in the Analysis phase of restart.

6.12 The write-ahead log(WAL) Protocol

- Transaction can be interrupted before running to completion for a variety of reasons. A DBMS must ensure that the changes made by such incomplete transaction are removed form the database. To do son, the DBMS maintains a log of all units to the database. A crucial property of the log is each write action change must be recorded first in the database and then the change is recorded in the log. But, if the system crashed between this order of changes, i.e., if the system crashes just after making the change in the database but before making the change in the log, then the DBMS would be unable to detect this new change and undo this new change. This property is called as Write-Ahead Log or WAL.

6.13 CHECKPOINTING

- A checkpoint is like a small idea of the DBMS state and by taking checkpoints periodically, the DBMS can reduce the amount of work to be done during restart in the event of a crash.

- **Checkpointing in ARIES has three steps:**

    1. First, a begin-checkpoint record is written to indicate when the checkpoint starts.
    2. Second, an end-checkpoint record is constructed, including in it the current contents of the transaction table and the dirty page table, and appended to the log.
    3. Third step is carried out after the end-checkpoint record is written to stable storage: A special master record containing the LSN of the begin-checkpoint log record is written to a known place on stable storage.

- While the end-checkpoint record is being constructed, the DBMS continues executing transactions and writing other log records; the only guarantee we have,

is that the transaction table and dirty page table are accurate as the time of the begin-checkpoint record. This kind of checkpoint, is called as fuzzy checkpoint, which is inexpensive because it does not require writing out pages in the buffer pool. The effectiveness of this checkpointing technique is limited by the earliest rec, LSN of pages in the dirty pages table, because during restart we must redo changes starting from the log record whose LSN is equal to this rec LSN. Having a background process that periodically writes dirty pages to disk, helps to limit this problem'

- Thus, when the system comes back after a crash, the restart process begins the normal execution by taking a checkpoint, in which the transaction table and dirty page table both are empty.

6.14 Recovering from a System Crash

- When the system is restarted after a crash, the recovery manager proceeds in three phases, as illustrated in the following figure:

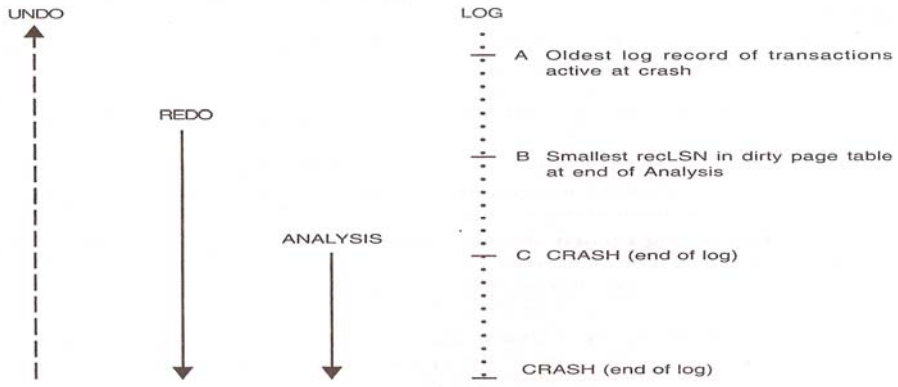


Fig : Three Phases of Restart in ARIES

- ıeckpoint ard in the log until the last log record.

- The Redo phase follows Analysis and redoes all changes to any page that might have been dirty at the time of the crash; this set of pages and the starting point for Redo (the smallest rec LSN of any dirty page) are determined during Analysis.

- The Undo phase follows Redo and undoes the changes of all transactions that were active at the time of the crash; again, this set of transactions is identified during the Analysis phase.

- Remember that Redo reapplies changes in the order in which they were originally carried out; whereas Undo reverses changes in the opposite order, reversing the most recent change first.

■ Analysis phase

- The Analysis phase performs three tasks:
  1. It determines the pointy in the log at which it starts the Redo phase.
  2. It determines pages in the buffer pool that were dirty at the time of the crash.
  3. It identifies transactions that were active at the time of the crash and must be undone.

- Analysis phase begins by examining the most recent begin checkpoint log record and initializing the dirty page table and transaction table to the copies of those structures in the next end-checkpoint record. Thus, these tables are initialized to the set of dirty pages and active transactions at the time of the checkpoint.

- Analysis phase then scans the log in the forward direction until it reaches the end of the log as follows:

  (a) If an end log record for a transaction T is encountered, T is removed form the transaction table because it is no longer active.
  (b) If a log record other than an end record for a transaction T is encountered, an entry for T is added to the transaction table if it is not already there. Further, the entry for T is modified:

    1. The last LSN field is set to the LSN of this log record.
    2. If the log record is a commit record, the status is set to C, otherwise it is set to u (indicating that it is to be undone).

  (c) If a redoable log record affecting page 'P' is encountered, and 'P' is not in the dirty page table, an entry is inserted into this table with page id P and rec LSN equal to the LSN of this redoable log record. This LSN identifies the oldest change affecting page P that may not have been written to disk.

- Thus, at the end of the Analysis Phase, the transaction table contains an accurate list of all transaction that were active at the time of the crash, this is the set of transactions with status U. the dirty page table includes all pages that were dirty at the time of the crash, but may also contain some pages that were written to disk. If an end-write log records were written at the completion of each write operation, the dirty page table constructed during Analysis could be made more accurate, but in ARIES, the additional cost of writing end-write log records is not considered to be worth the gain.

■ **Redo Phase**

- During the redo Phase, ARIES reapplies the updates of all transactions, committed or uncommitted. Further, if a transaction was aborted before the crash, and its updated were undone, as indicated by CLRs, the actions describes in the CLRs are also reapplied. This repeating history paradigm distinguishes ARIES from other proposed WAL-based recovery algorithms and causes the database to be brought to the same state that it was in at the time of the crash.

- The Redo Phase begins with the log record that has the smallest rec LSN of all pages in the dirty page table constructed by the Analysis Phase, because this log record identifies the oldest update that may not have been written to disk prior to the crash. Starting from this log record. Redo scans forward until the end of the log. For each readable log record (update or CLR) encountered, Redo checks whether the logged actions must be redone.

- The actions must be redone, unless one of the following conditions holds:

- The affected page is not in the dirty page table. (OR)
- the affected page is in the dirty page table, but the rec LSN for the entry is greater than the LSN of the log record being checked. (OR)
- the page LSN is greater than or equal to the LSN of the log record being checked.

    - The first condition obviously means that all changes to this page have been written to
    - disk, because the rec LSN is the first update to this page that may not have been written to disk.
    - The second condition means that the update being checked was propagated to disk.
    - The third condition, which is checked last be cause it requires us to retrieve the page, also ensures that the update being checked was written to disk, because either this update or a later update to the page was written.

    - If the logged action must be redone:
      (i) The logged action is reapplied.
      (ii) The page LSN on the page is set to the LSN of the redone log record. No additional log record is written at this time.

■ Undo Phase

- The undo phase scans backward from the end of the log. The goal of this phase is to undo the actions of all transactions that were active at the time of the crash, i.e., effectively abort them. This set of transactions is identified in the transaction table constructed by the Analysis Phases.

▪ The Undo Algorithm

- Undo begins with the transaction table constructed by the analysis Phase, which identifies all transactions that were active at the time of crash, and includes the LSN of the most recent log record (the last LSN field) for each such transaction. Such transactions are called Loser Transactions.
- Thus all actions of losers must be undone, and further, these actions must be undone in the reverse order in which they appear in the log.
- Consider the set of last LSN values for all loser transactions. Let us call this set To Undo. Undo repeatedly chooses the largest (i.e., most recent) LSN value in this set and processes it, until ToUndo is empty.
- **To proceed a log record:**

1. If it is a CLR, and the undo Next LSN value is not null, the undo Next LSN is added to the set ToUndo; if the undoNextLSN is null, an end record is written for the transaction because it is completely undone, and the CLR is discarded.

2. If it is an update record, a CLR is written and the corresponding action is undone and the prev LSN value in the update log record is added to the set ToUndo.

- When the set ToUndo is empty the Undo Phase is complete. Restart is now complete, and the system can proceed with normal operations.