UNIT-V

(Transaction Processing, Concurrency Control, and Recovery)

Concurrency Control

- How to design those schedules which ensure conflict serializability and other properties?
- **Timestamp Ordering Protocols**
- Basic idea of time stamping is to decide the order between the transaction before they enter in the system using a stamp (time stamp)
- With each transaction ti, in the system, we associate a unique fixed timestamp, denoted by TS(ti). If a new transaction tj , enters into the system with a timestamp TS(tj), then always TS(ti) <TS(tj).

- Let W-timestamp(Q) is the largest time-stamp of any transaction that executed write(Q) successfully.
- Let R-timestamp(Q) is the largest time-stamp of any transaction that executed read(Q) successfully.
- **Read Operation Request**
- If TS(Ti) < W-timestamp(Q), then Ti needs to read a value of Q that was already overwritten. Hence, the read operation is rejected, and Ti is rolled back.
- If TS(Ti)≥ W-timestamp(Q), then the read operation is executed, and R-timestamp(Q) is set to the maximum of R-timestamp(Q) and TS(Ti).

- **Write Operation Request**
- If TS(Ti) < R-timestamp(Q), This means that the value of Q is already read. Hence, the write operation is rejected, and Ti is rolled back.
- If TS(Ti) < W-timestamp(Q), then Ti is attempting to write an obsolete value of Q. Hence, this write operation is rejected, and Ti is rolled back.
- If TS(Ti) ≥ R-timestamp(Q), then the write operation is executed, and W-timestamp(Q) is set to max(W-timestamp(Q), TS(Ti)).
- If TS(Ti) ≥ W-timestamp(Q), then the write operation is executed, and W-timestamp(Q) is set to max(W-timestamp(Q), TS(Ti)).

- Problem: Dirty reads allowed; Order of commit not discussed; Hence Cascadelessness and recoverability are questionable.
- Advantage: Deadlock free, conflict and view serializability ensured
- Same for Thomas write rule except for conflict serializabilty.

- **THOMAS WRITE RULE**
- It is an improvement in time stamping protocol, which makes some modification and may generate those protocols that are even view serializable.
- It is a Modified version of the timestamp-ordering protocol in which Blind write operations may be ignored under certain circumstances.
- The protocol rules for read operations remain unchanged.

- Write Request Rules:
- if TS(Ti) < W-timestamp(Q), then Ti is attempting to write an obsolete value of {Q}. Rather than rolling back Ti as the timestamp ordering protocol would have done, this {write} operation can be ignored.
- This modification is valid because with TS(Ti) < W-timestamp(Q), the value written by this transaction will never be read by any other transaction performing Read(Q) ignoring such obsolete write operation is considerable.
- Thomas' Write Rule allows greater potential concurrency. Allows some view- serializable schedules that are not conflict serializable.

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

Lock Based Protocols

- To ensure isolation is required that data items be accessed in a mutually exclusive manner.
- Locking is the most fundamental approach to ensure this.
- Idea is first obtain a lock on the desired data item then if lock is granted then perform the operation and then unlock it.

- Two modes of locking
- **Shared mode**: If transaction Ti has obtained a shared-mode lock (denoted by S) on any data item Q, then Ti can read, but cannot write Q.
- Any other transaction can also acquire a shared mode lock on the same data item.
- **Exclusive mode**: If transaction Ti has obtained an exclusive-mode lock (denoted by X) on any data item Q, then Ti can both read and write Q,
- Any other transaction cannot acquire either a shared or exclusive mode lock on the same data item.

- Compatibility Matrix: Helps Concurrency Control Manager to grant the locks as per rules.

Requested Lock	Current State of lock of data items: Exclusive	Shared	Unlocked
Exclusive	N	N	Y
Shared	N	Y	Y
Unlock	Y	Y	-

- Lock based protocol **do not ensure serializability** as granting and releasing of lock do not follow any order and any transaction any time may go for lock and unlock.

T_1	T_2
LOCK-X(A)	
READ(A)	
WRITE(A)	
UNLOCK(A)	
	LOCK-S(B)
	READ(B)
	UNLOCK(B)
LOCK-X(B)	
READ(B)	
WRITE(B)	
UNLOCK(B)	
	LOCK-S(A)
	READ(A)
	Unlock (A)

- Problem is that transactions release locks too soon, resulting in loss of total isolation and atomicity.
- To guarantee serializability, need an additional protocol concerning the positioning of lock and unlock operations in every transaction.
- Transaction follows 2PL protocol if all locking operations precede first unlock operation in the transaction
- Deadlock
- More duration of locked variable-> concurrency is compromised

- **Two phase locking protocol (Basic 2PL)**
- The protocol ensures that each transaction issue lock and unlock requests in two phases, note that each transaction will be 2 phased schedule.
- Growing phase- A transaction may obtain locks, but not release any locks.
- Shrinking phase- A transaction may release locks, but may not obtain any new locks.
- Initially a transaction is in growing phase and acquires lock as needed and in between can perform operation to reach lock point and once a transaction releases a lock, it can issue no more lock requests i.e. it enters the shrinking phase.
- The order of serializabilty is the order of Ti reaching at the lock point
- Problem:Deadlock; Non-recoverability, cascading rollback

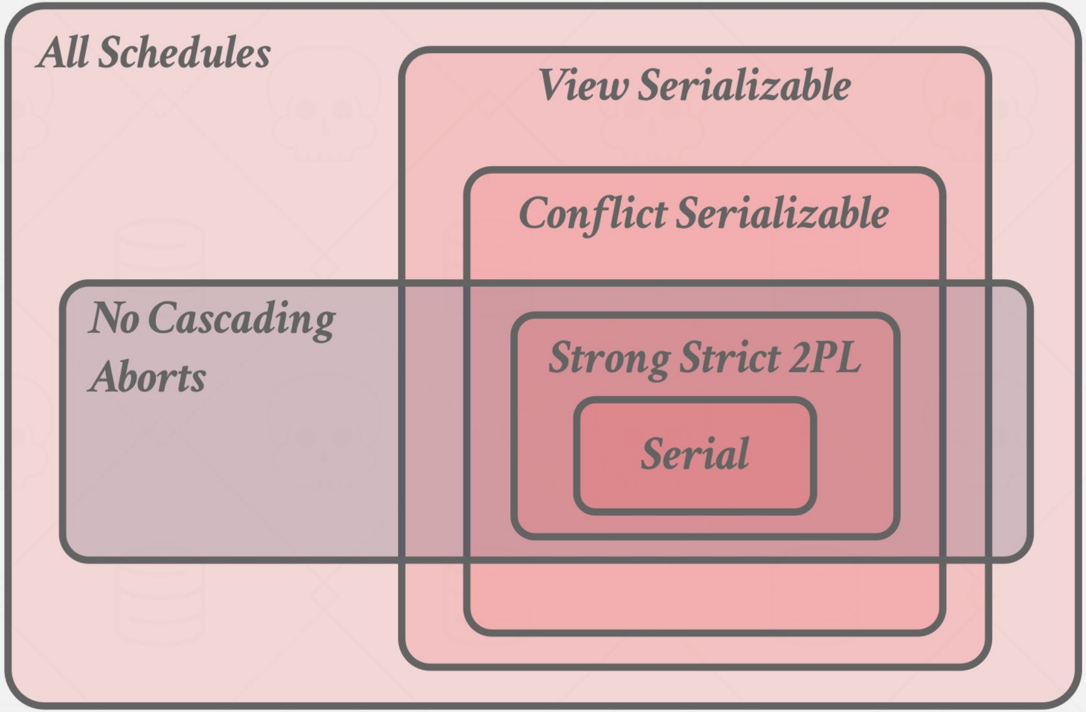
T_1	T_2
LOCK-X(A)	
READ(A)	
WRITE(A)	
	LOCK-S(B)
	READ(B)
LOCK-X(B)	
READ(B)	
WRITE(B)	
	LOCK-S(A)
	READ(A)
	UNLOCK(B)
UNLOCK(A)	
UNLOCK(B)	
	UNLOCK(A)

- **Conservative 2PL**
- The idea is there is no growing phase
- Transaction start directly from lock point,
- i.e. transaction must first acquire all the required locks then only it can start execution.
- If all the locks are not available then transaction must release the acquired locks and must wait.(Deadlock free)
- Shrinking phase will work as usual, and transaction can unlock any data item at anytime.
- Problem: we must have a knowledge in future to understand what is data required so that we can use it + recoverability and cascadelessness is compromised+ concurrency compromised because all locks held.

- **<u>Rigorous 2PL</u>**
- Requires that all locks be held until the transaction commits.
- This protocol requires that locking be two phase and also all the locks taken be held by transaction until that transaction commit.
- Hence there is no shrinking phase in the system.
- Recoverable and cascadeless schedule created (Hint: Dirty read avoided)
- Problem: Deadlock because growing phase exists + No unlock hence concurrency is compromised

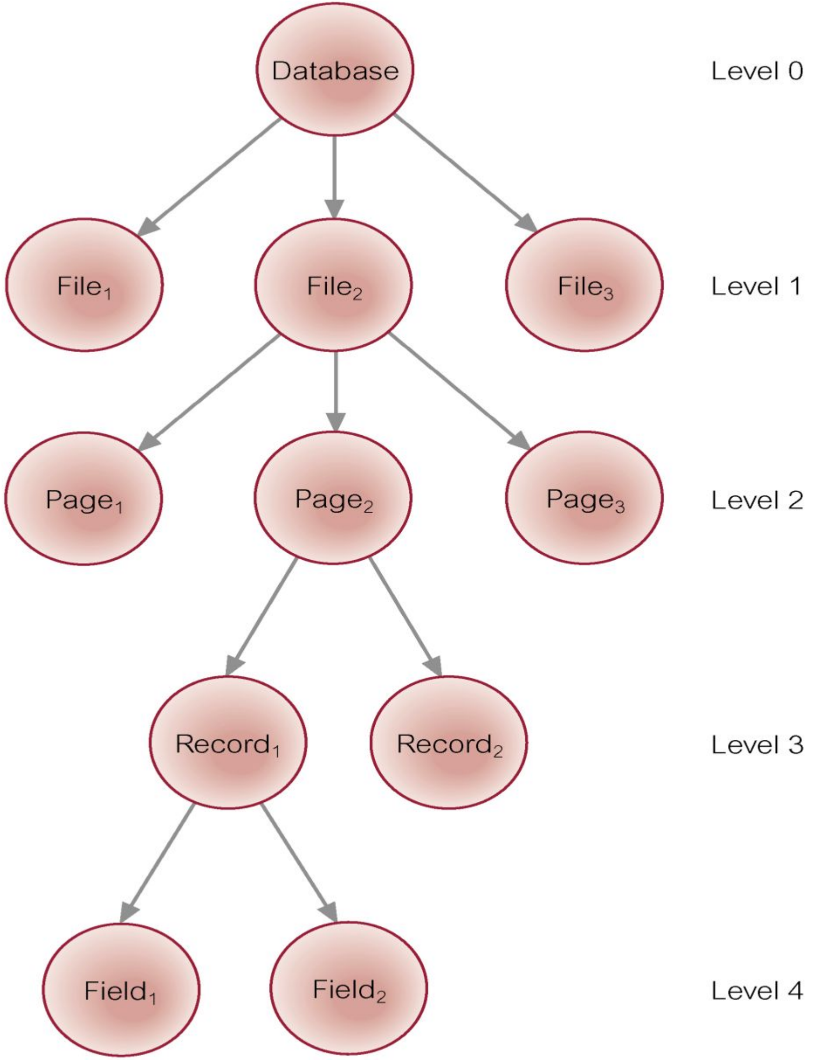
- **<u>Strict 2PL</u>**
- All exclusive-mode locks needed by transaction are held until that transaction commits.
- This requirement ensures that any data written by an uncommitted transaction are locked in exclusive mode.
- This prevents any other transaction from reading the data.
- Shared locks are released in partial shrinking phase
- So it is simplified form of rigorous 2PL
- Properties similar to Rigorous 2PL
- Problem: Deadlock because growing phase exists

UNIVERSE OF SCHEDULES



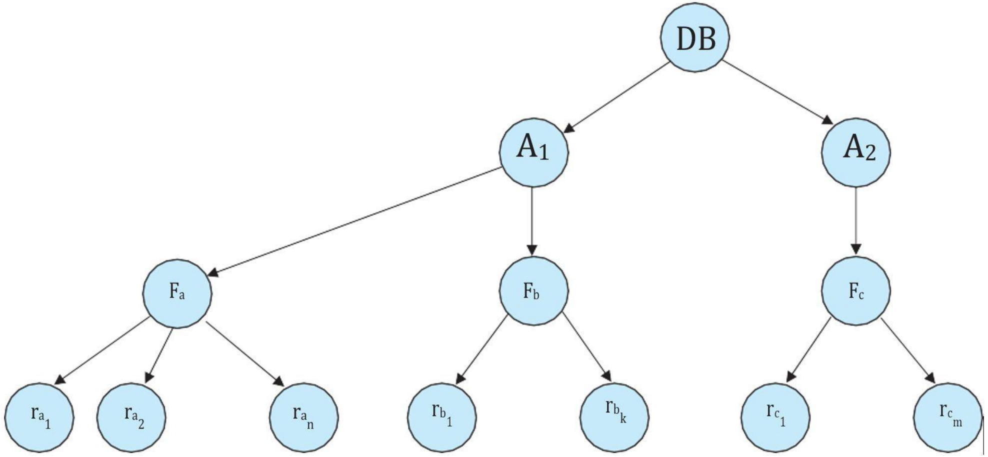
- **Granularity of Data Items (Multiple)**
- Size of data items chosen as unit of protection by concurrency control protocol.
- Ranging from coarse to fine:
  - The entire database.
  - A file.
  - A page (or area or database spaced).
  - A record.
  - A field value of a record.

- Tradeoff:
  - coarser, the lower the degree of concurrency;
  - finer, more locking information that is needed to be stored.
  - Best item size depends on the types of transactions.
- **Hierarchy of Granularity**
- Could represent granularity of locks in a hierarchical structure.
- Root node represents entire database, level 1s represent files, etc.
- When node is locked, all its descendants are also locked.
- DBMS should check hierarchical path before granting lock.


Database
Level 0
File$_1$
File$_2$
File$_3$
Level 1
Page$_1$
Page$_2$
Page$_3$
Level 2
Record$_1$
Record$_2$
Level 3
Field$_1$
Field$_2$
Level 4

•

	IS	IX	S	SIX	X
IS	True	True	True	True	False
IX	True	True	False	False	False
S	True	False	True	False	False
SIX	True	False	False	False	False
X	False	False	False	False	False



- **<u>Deadlock</u>**
- An impasse that may result when two (or more) transactions are each waiting for locks held by the other to be released.
- .

Time	T_{17}	T_{18}
t_1	begin_transaction	
t_2	write_lock(bal_x)	begin_transaction
t_3	read(bal_x)	write_lock(bal_y)
t_4	$bal_x = bal_x - 10$	read(bal_y)
t_5	write(bal_x)	$bal_y = bal_y + 100$
t_6	write_lock(bal_y)	write(bal_y)
t_7	WAIT	write_lock(bal_x)
t_8	WAIT	WAIT
t_9	WAIT	WAIT
t_{10}	⋮	WAIT
t_{11}	⋮	⋮

- Three general techniques for handling deadlock:
  - Deadlock prevention.
  - Timeouts.
  - Deadlock detection and recovery.
- **Deadlock Prevention**
- DBMS looks ahead to see if transaction would cause deadlock and never allows deadlock to occur.
- Could order transactions using transaction timestamps:
- Wait-Die - older transaction waits for younger one; older transaction is aborted if necessary. Allows recent jobs (short ones) to complete
- Wound-Wait - younger transaction waits for an older one. If older transaction requests lock held by younger one, younger one is aborted (wounded). Allows long jobs to complete.

- **Timeouts**
- Transaction that requests lock will only wait for a system-defined period of time.
- If lock has not been granted within this period, lock request timeout.
- Problem: DBMS assumes transaction may be deadlocked, even though it may not be, and it aborts and automatically restarts the transaction.

- **Deadlock Detection and Recovery**
- DBMS allows deadlock to occur but recognizes it and breaks it.
- Usually handled by construction of wait-for graph (WFG) showing transaction dependencies:
- Create a node for each transaction.
- Create edge Ti -> Tj, if Ti waiting to lock item locked by Tj.
- Deadlock exists if and only if WFG contains cycle.
- WFG is created at regular intervals.

- **Recovery from Deadlock Detection**
- Several issues:
- choice of deadlock victim;
- how far to roll a transaction back;
- avoiding starvation
- Where one transaction (say the longest one) is not explicitly prevented from progressing, but actually never does progress
- Wait-Die: long transactions repeatedly aborted to give priority to shorter transactions
- Wound-Wait: short transactions never started because longer transactions are always in process

- **Optimistic Techniques**
- Locking and Timestamping techniques focus on preventing conflicts
- In some systems, conflict is rare and more efficient to let transactions proceed without delays, so no guarantee of serializability.
- At commit, check is made to determine whether or not a conflict has occurred.
- If there is a conflict, transaction must be rolled back and restarted.
- Potentially allows greater concurrency than traditional protocols.
- Best for systems having more read requests rather than write requests

- **Database Recovery**
- Process of restoring database to a correct state in the event of a failure.
- Need for Recovery Control
- Two types of storage: volatile (main memory) and nonvolatile.
- Volatile storage does not survive system crashes.
- Stable storage represents information that has been replicated in several nonvolatile storage media with independent failure modes.

- **Transactions and Recovery**
- Transactions represent basic unit of recovery.
- Recovery manager responsible for atomicity and durability.
- If failure occurs between commit and database buffers being flushed to secondary storage then, to ensure durability, recovery manager has to redo (rollforward) transaction’s updates.
- If transaction had not committed at failure time, recovery manager has to undo (rollback) any effects of that transaction for atomicity.
- Partial undo - only one transaction has to be undone.
- Global undo - all transactions have to be undone.

- **DBMS should provide following facilities to assist with recovery:**
- Backup mechanism, which makes periodic backup copies of database.
- Logging facilities, which keep track of current state of transactions and database changes.
- Checkpoint facility, which enables updates to database in progress to be made permanent.
- Recovery manager, which allows DBMS to restore database to consistent state following a failure.

- **Log File**
- Contains information about all updates to database:
- Transaction records.
- Checkpoint records.
- Often used for other purposes (for example, auditing).
- Log file may be duplexed or triplexed.
- Log file sometimes split into two separate random-access files.
- Potential bottleneck; critical in determining overall performance.
- **Checkpoint**
- Point of synchronization between database and log file. All buffers are force-written to secondary storage.
- Checkpoint record is created containing identifiers of all active transactions.
- When failure occurs, redo all transactions that committed since the checkpoint and undo all transactions active at time of crash.

- The system log, or transaction log, records all the changes or activities happening in the database, ensuring that transactions are durable and can be recovered in the event of a failure.
- Different types of log records represent various stages or actions in a transaction.
- <Ti start>: This log entry indicates that the transaction Ti has started its execution.
- < Ti, Xj, V1, V2>: This log entry documents a write operation. It states that transaction Ti has changed the value of data item Xj from V1 to V2.
- < Ti commit>: This log entry marks the successful completion of transaction Ti.
- < Ti abort>: This log entry denotes that the transaction Ti has been aborted, either due to an error or a rollback operation.
- Before any write operation modifies the database, a log record of that operation needs to be created. This is to ensure that in case of a failure, the system can restore the database to a consistent state using the log records.

- **Three Main Recovery Techniques**
- Deferred Update
- Immediate Update
- Shadow Paging

- **Deferred Update**
- Updates are not written to the database until after a transaction has reached its commit point. (Delayed Commit)
- If transaction fails before commit, it will not have modified database and so no undoing of changes required. (Less complex recovery)
- May be necessary to redo updates of committed transactions as their effect may not have reached database.
- Reduced I/O operations, as changes are batched and written at once during the commit, saving on intermediate I/O operations.

- **Immediate Update**
- Updates are applied to database as they occur (don’t wait for commit).
- As before, to redo updates of committed transactions following a failure (may not have reached DB).
- May need to undo effects of transactions that had not committed at time of failure.
- Essential that log records are written before write to database. Write-ahead log protocol.
- Increased I/O operations, as each change triggers immediate write operations, leading to more frequent I/O operations.

- **Shadow Paging**
- Maintain two page tables during life of a transaction: current page and shadow page table.
- When transaction starts, two pages are the same.
- Shadow page table is never changed thereafter and is used to restore database in event of failure.
- During transaction, current page table records all updates to database.
- When transaction completes, current page table becomes shadow page table.
- Four phases: Initialization->modification->commit->recovery