DATABASE ENGINEERING

Unit 5

Question Bank — Transaction Processing and Concurrency Control

Section A — 40 short-answer questions (2 marks each)

Section B — 17 long-answer questions (7 marks each), with solved schedules

Merges the Unit V syllabus topics, the model question bank and every Unit V question from PYQ R25, B26 and R24

Compiled from the official five-unit syllabus, the previous-year question papers (R25, B26, R24), the class slide decks, the model question banks and the handwritten notes. Every question is answered in full; differences are shown in two-column tables and ER diagrams are drawn wherever a question needs one.

Section A — 2-Mark Questions

Q1. Define a transaction.

A **transaction** is a logical unit of database work made of one or more operations that must succeed or fail as a whole. Example: transferring ₹1,000 from A to B — Read(A), A = A − 1000, Write(A), Read(B), B = B + 1000, Write(B), COMMIT. Either the whole transfer happens or none of it does.

Q2. Define the read and write operations of a transaction.

Model question — Unit 5

Read(X) copies the database item X into a local variable of the transaction, written $R_i(X)$. **Write(X)** copies the local value back to the database item, written $W_i(X)$. A transaction is a sequence of these operations together with COMMIT or ABORT.

Q3. Write the ACID properties of a transaction.

Model question — Unit 5, PYQ R25 Q6(a)

Property	Meaning
Atomicity	All operations happen, or none happen
Consistency	The database moves from one valid state to another
Isolation	Concurrent transactions behave as if run alone
Durability	Committed changes survive any failure

Q4. Define atomicity.

Atomicity means a transaction is indivisible: either all of its operations are reflected in the database or none are. If it fails part-way, every change it made is undone. Example: a transfer that debits A but fails before crediting B must have the debit rolled back.

Q5. Define durability.

Durability means that once a transaction commits, its changes are permanent and survive any subsequent failure — power loss, system crash or disk error. It is ensured by writing the log to stable storage before commit and redoing committed work during recovery.

Q6. What is a schedule?

A **schedule** is the chronological order in which the operations of one or more transactions are executed. It preserves the internal order of operations within each transaction. Notation: $R_1(X)$ means T1 reads X, $W_2(Y)$ means T2 writes Y.

Q7. Distinguish serial and non-serial schedules.

PYQ B26 Q1(n)

Serial schedule	Non-serial schedule
One transaction completes before the next starts	Operations of several transactions are interleaved
No concurrency, so lower throughput	Better throughput and response time
Always correct	May be incorrect, so must be checked for serializability
Example: T1 → T2	Example: R1(X), R2(X), W1(X), W2(X)

Q8. What is the lost update problem?

Model question — Unit 5

Two transactions read the same old value and one update overwrites the other. If T1 and T2 both read X = 100, and T1 writes 120 while T2 writes 90, the final value is 90 and T1's update is lost. The correct value should have been 110.

Q9. What is a dirty read?

A **dirty read** occurs when a transaction reads data written by another transaction before that transaction commits. If the writer then rolls back, the reader has used a value that never officially existed. It is also called the uncommitted dependency or temporary update problem.

Q10. Define the unrepeatable read problem.

An **unrepeatable read** occurs when a transaction reads the same item twice and gets different values, because another transaction updated and committed it between the two reads. Within one transaction the same read has produced two different answers.

Q11. What is the incorrect summary problem?

An **incorrect summary** occurs when an aggregate is computed while another transaction is updating some of the rows being summarised. The total then mixes old and new values and is wrong, even though every individual read was of a committed value. Also called inconsistent analysis.

Q12. What is a phantom read?

A **phantom read** occurs when a transaction runs a condition-based query twice and the set of matching rows changes, because another transaction inserted or deleted a row that satisfies the condition. No existing row was updated, so it is not an unrepeatable read.

Q13. Define concurrency. Why is a concurrency control mechanism needed?

Model question — Unit 5

Concurrency is the interleaved execution of two or more transactions in overlapping time, so that CPU and I/O are not wasted waiting for one transaction to finish. Concurrency control is needed because uncontrolled interleaving produces lost updates, dirty reads, unrepeatable reads and incorrect summaries, which leave the database inconsistent.

Q14. Define serializability.

Serializability is the correctness criterion for concurrent schedules. A schedule is serializable if its effect is equivalent to that of some serial schedule of the same transactions. It allows concurrency while preserving the correctness that serial execution guarantees.

Q15. What is conflict serializability?

Model question — Unit 5

A schedule is **conflict serializable** if it is conflict equivalent to some serial schedule — that is, the order of every pair of conflicting operations is preserved. It is tested with a precedence graph: an acyclic graph means conflict serializable, a cycle means it is not.

Q16. When do two operations conflict?

PYQ R25 Q1(n)

Two operations conflict when all three conditions hold: they belong to **different transactions**, they access the **same data item**, and **at least one of them is a write**. Read-read never conflicts; read-write, write-read and write-write always do.

Q17. What is a precedence graph?

Model question — Unit 5

A **precedence graph** (serialization graph) has one node per transaction and an edge $T_i \rightarrow T_j$ whenever an operation of T_i conflicts with a later operation of T_j. A schedule is conflict serializable if and only if this graph has no directed cycle.

Q18. What is conflict equivalence?

Two schedules are **conflict equivalent** if they contain the same operations and preserve the order of every pair of conflicting operations. Non-conflicting operations may be reordered freely; conflicting ones may not.

Q19. Define view serializability.

PYQ R24 Q1(m)

A schedule is **view serializable** if it is view equivalent to some serial schedule — that is, the initial readers, the read-from relationships and the final writers of every data item are the same as in that serial schedule. It is a broader class than conflict serializability.

Q20. State the three conditions for view equivalence.

1. **Initial read** — if T_i reads the initial value of X in one schedule, it must read the initial value in the other. 2. **Read-from** — if T_i reads X written by T_j in one schedule, the same must hold in the other. 3. **Final write** — the transaction performing the final write of X must be the same in both.

Q21. What is a blind write?

A **blind write** is a write performed by a transaction without first reading that data item. Blind writes matter because a schedule containing them can be view serializable even though it is not conflict serializable — the only situation in which the two notions differ.

Q22. Differentiate conflict and view serializability.

PYQ R25 Q1(m), B26 Q1(m)

Conflict serializability	View serializability
Based on the order of conflicting operations	Based on read-from relations and final writes
Tested with a precedence graph	Tested by the three view conditions
Rejects some valid blind-write schedules	Accepts certain blind-write schedules
A subset of view-serializable schedules	A superset of conflict-serializable schedules

Q23. What is a recoverable schedule?

A schedule is **recoverable** if, whenever T_j reads a value written by T_i, the commit of T_i occurs before the commit of T_j. Without this, a reader could commit a value whose writer later aborts, and no correct recovery would be possible.

Q24. What is a cascadeless schedule?

A **cascadeless** schedule is one in which every transaction reads only items written by **committed** transactions. It prevents cascading rollback, where the abort of one transaction forces the rollback of every transaction that read from it.

Q25. What is a strict schedule?

A **strict** schedule is one in which a transaction neither reads nor writes an item until the last transaction that wrote it has committed or aborted. Strictness makes recovery simple, because an aborted transaction's changes can be undone without disturbing anyone else.

Q26. What is deadlock?

A **deadlock** occurs when two or more transactions are each waiting for a lock held by another, so none can proceed. Example: T1 holds X and waits for Y while T2 holds Y and waits for X.

Q27. What is a wait-for graph?

A **wait-for graph** has one node per transaction and an edge $T_i \rightarrow T_j$ when T_i is waiting for a lock held by T_j. A cycle in this graph indicates a deadlock, and the DBMS resolves it by aborting one transaction, called the victim.

Q28. What are shared and exclusive locks?

A **shared lock (S)** allows a transaction to read an item and is compatible with other shared locks. An **exclusive lock (X)** allows reading and writing and is compatible with nothing. Two transactions may hold S on the same item, but X excludes everyone.

Q29. What is two-phase locking?

Two-phase locking (2PL) requires a transaction to acquire all its locks in a growing phase and release them in a shrinking phase, with no lock acquired after the first release. If every transaction follows 2PL, every resulting schedule is conflict serializable.

Q30. What is the lock point?

The **lock point** of a transaction under 2PL is the moment at which it holds all the locks it will ever need — the boundary between the growing and shrinking phases. The lock points of the transactions define their serialization order.

Q31. What is a timestamp in a timestamp protocol?

A **timestamp** $TS(T_i)$ is a unique number given to a transaction when it starts, usually from a counter or the system clock. The protocol keeps the schedule equivalent to the serial order of these timestamps, so no locks are needed and no deadlock can occur.

Q32. What is the Thomas write rule?

Under timestamp ordering, if T_i tries to write X and $TS(T_i) < write_TS(X)$, the write is **ignored** rather than causing a rollback, because a later write will overwrite it anyway. The rule allows some schedules that the strict rule would reject.

Q33. List the types of database failure.

PYQ R25 Q1(o), R24 Q1(o)

Type	Example	Recovery need
Transaction failure	Logical error, constraint violation, deadlock victim	Undo the transaction

Type	Example	Recovery need
System failure	Power loss, OS or DBMS crash	Redo committed, undo incomplete
Media failure	Disk crash, corrupted file	Restore from backup and apply logs
Communication failure	Network break in a distributed system	Resolve incomplete distributed work

Q34. What is a transaction log? What does it contain?

A **log** is a sequential file on stable storage recording every change made by every transaction. It contains START, update records holding the data item with its old and new values, COMMIT and ABORT records, and CHECKPOINT records. The old value supports UNDO and the new value supports REDO.

Q35. What is write-ahead logging?

Write-ahead logging requires that the log record for a change reaches stable storage **before** the changed data page does. Without it, a crash could leave a change in the database with no log record to undo or redo it.

Q36. Differentiate UNDO and REDO.

UNDO	REDO
Reverses the effect of a change, restoring the old value	Reapplies a change, setting the new value
Applied to transactions that did not commit	Applied to transactions that committed
Uses the old value in the log record	Uses the new value in the log record
Example: T1 wrote 9000 but aborted, restore 10000	Example: T2 committed 12000, reapply it after a crash

Q37. What is a checkpoint?

A **checkpoint** is a recovery marker at which the DBMS forces all modified pages and log records to disk and writes a CHECKPOINT record. After a crash, recovery starts from the last checkpoint rather than the beginning of the log, which keeps restart time bounded.

Q38. What is shadow paging?

Shadow paging is a recovery technique needing neither a log nor UNDO/REDO. Modified pages are written to new disk blocks, and the current page table points to them while the shadow page table still points to the old blocks. Commit is a single atomic switch of the table; a failure before it leaves the old state intact.

Q39. Differentiate deferred update from immediate update.

Deferred update	Immediate update
Changes are written to the database only at commit	Changes may be written before commit
Recovery needs only REDO	Recovery needs both UNDO and REDO
Nothing uncommitted ever reaches the database	Uncommitted data may reach the database
All changes must be held until commit	Practical for large transactions

Q40. Explain the requirements of recovery.

PYQ B26 Q1(o)

Recovery must guarantee **atomicity** — the effects of uncommitted transactions are undone — and **durability** — the effects of committed transactions are redone. It must restore the database to the most recent consistent state after a transaction, system or media failure, using information that survives the failure: the log, checkpoints and backups.

Section B — 7-Mark Questions

Q1. Explain transactions and the ACID properties with an example.

PYQ R25 Q6(a)

The transaction

A transaction is a logical unit of work made of operations that must succeed or fail as a whole. Example: transferring ₹1,000 from account A to account B. Read(A); A = A − 1000; Write(A); Read(B); B = B + 1000; Write(B); COMMIT.

ACID

Property	Meaning	How it is ensured
Atomicity	All operations happen or none do	Rollback and undo of incomplete work
Consistency	One valid state moves to another	Constraint checking
Isolation	Concurrent transactions behave as if alone	Concurrency control and serializability
Durability	Committed changes survive failure	Logging, checkpoints and redo

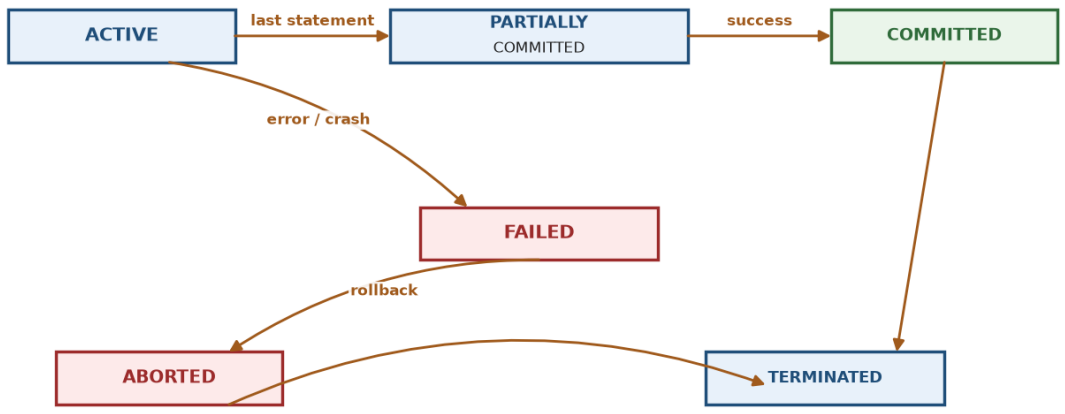
Why each matters

If the transfer fails after debiting A, **atomicity** undoes the debit, otherwise money disappears. **Consistency** guarantees the total of all balances is unchanged. **Isolation** stops another transaction from reading the half-finished state. **Durability** guarantees that once the customer is told the transfer succeeded, a power cut cannot undo it.

Q2. Explain the transaction states with a state diagram.

PYQ B26 Q6(a), model question

Transaction State Diagram



An aborted transaction may be restarted, or terminated permanently.

State	Meaning
Active	Executing read, write and computation operations
Partially committed	The final statement has executed but changes are not yet secured
Committed	Completed successfully; changes are permanent
Failed	Cannot proceed because of an error, deadlock or crash
Aborted	Rolled back; incomplete effects removed
Terminated	Processing finished after commit or abort

The two paths

Active → Partially committed → Committed, or Active → Failed → Aborted. Both end at Terminated. An aborted transaction may be restarted if the failure was transient, such as a deadlock or a crash; if it failed because of a logic error, restarting would fail again, so it is terminated and the application is informed.

Q3. Distinguish serial and non-serial schedules and explain the concurrency problems with examples.

Model question — Unit 5

Serial and non-serial

A **serial** schedule completes one transaction before starting the next; it is always correct but allows no concurrency. A **non-serial** schedule interleaves operations, giving better throughput but risking incorrect results.

Lost update

Lost Update — one write overwrites the other

Step	T1	T2	Value of X
1	Read X = 100		100
2		Read X = 100	100
3	X = X + 20		100
4		X = X - 10	100
5	Write X = 120		120
6		Write X = 90	90

Expected after both updates: 110. Stored: 90. T1's update is lost.

Both transactions read X = 100; T1 writes 120 and T2 writes 90. The stored value is 90, though the correct value is 110. T1's update is lost.

Dirty read

Dirty Read — reading a value that is later rolled back

Step	T1	T2	Comment
1	Write X = 500		uncommitted
2		Read X = 500	reads dirty value
3	ROLLBACK		X returns to 100
4		uses 500	value never existed

Also called the uncommitted dependency or temporary update problem.

T1 writes X = 500 without committing; T2 reads 500; T1 rolls back. T2 has used a value that never existed.

Unrepeatable read

T1 reads X = 100; T2 updates X to 120 and commits; T1 reads X again and sees 120. The same read gave two answers within one transaction.

Incorrect summary

T1 sums A and B while T2 transfers money from A to B. T1 reads A before the transfer and B after it, so the total mixes an old value with a new one.

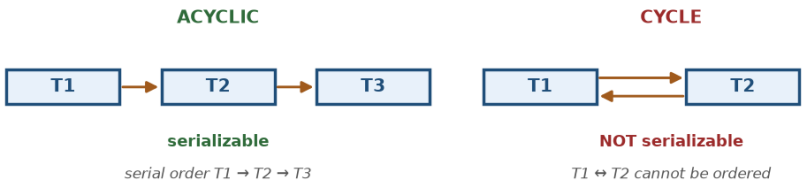
Phantom read

T1 counts the rows matching a condition; T2 inserts a matching row and commits; T1 repeats the count and gets a different number.

Q4. Explain conflict serializability and the precedence graph method.

Model question — Unit 5

Precedence Graph — acyclic means serializable



Edge Ti → Tj is drawn when an operation of Ti conflicts with a later operation of Tj.

Conflicting operations

Two operations conflict when they belong to different transactions, access the same item, and at least one is a write. Read-read never conflicts; read-write, write-read and write-write always do.

The method

1. Create one node per transaction. 2. Find every conflicting pair. 3. If an operation of T_i precedes the conflicting operation of T_j, draw $T_i \rightarrow T_j$. 4. Check for a directed cycle.

The conclusion

Acyclic = conflict serializable, and any topological order gives an equivalent serial schedule. **Cycle = not conflict serializable.**

Worked example 1

S = R1(X), W1(X), R2(X), W2(X). Conflicts: W1(X) before R2(X) gives T1 → T2; W1(X) before W2(X) gives T1 → T2. Acyclic, so serializable with order T1 → T2.

Worked example 2

S = R1(X), R2(Y), W1(Y), W2(X). Conflicts: R1(X) before W2(X) gives T1 → T2; R2(Y) before W1(Y) gives T2 → T1. Cycle T1 ↔ T2, so not conflict serializable.

Q5. Check the conflict serializability of the three schedules from the paper.

PYQ R25 Q6(b)

S1: r1(x); r3(x); w3(x); w1(x); r2(x)

Conflicts on x: r1(x) before w3(x) gives T1 → T3; w3(x) before w1(x) gives T3 → T1. The graph has the cycle T1 ↔ T3, so **S1 is not conflict serializable**.

S2: r3(x); r2(x); r1(x); w3(x); w1(x)

Conflicts: r2(x) before w3(x) gives T2 → T3; r1(x) before w3(x) gives T1 → T3; r2(x) before w1(x) gives T2 → T1. Edges T2 → T3, T1 → T3, T2 → T1 form the acyclic chain T2 → T1 → T3, so **S2 is conflict serializable with the serial order T2, T1, T3**.

S3: r1(x); w2(x); r3(x); w1(x); r2(x); w3(x)

Conflicts: r1(x) before w2(x) gives T1 → T2; w2(x) before w1(x) gives T2 → T1; w2(x) before r3(x) gives T2 → T3; r1(x) before w3(x) gives T1 → T3. The cycle T1 ↔ T2 means **S3 is not conflict serializable**.

Q6. Check the conflict serializability of the two schedules from the backlog paper.

PYQ B26 Q6(b)

S1: r1(x); r1(y); w1(x); w2(x); r2(y)

Conflicts: r1(x) before w2(x) gives T1 → T2; w1(x) before w2(x) gives T1 → T2; r1(y) and r2(y) are two reads and do not conflict. The graph has only T1 → T2 and is acyclic, so **S1 is conflict serializable with the serial order T1, T2**.

S2: r2(x); r1(y); w1(x); w2(y)

Conflicts: r2(x) before w1(x) gives T2 → T1; r1(y) before w2(y) gives T1 → T2. The cycle T1 ↔ T2 means **S2 is not conflict serializable**.

Q7. Analyse the schedule R1(A), R2(B), W1(B), W2(A) and check view serializability.

PYQ R24 Q1(n), model question

Conflict serializability

Conflicts: R2(B) before W1(B) gives T2 → T1; R1(A) before W2(A) gives T1 → T2. The precedence graph contains the cycle T1 ↔ T2, so the schedule is **not conflict serializable**.

Is it view serializable?

Check for blind writes. T1 reads A before writing A, and T2 reads B before writing B, so **there is no blind write**. When a schedule is not conflict serializable and contains no blind write, it cannot be

view serializable either. Therefore the schedule is **neither conflict nor view serializable**, and it is inconsistent.

Q8. Analyse the schedule R1(A), R2(A), R1(B), R2(B), R3(B), W1(A), W2(B) from the model paper.

Model question — Unit 5

The schedule

R1(A), R2(A), R1(B), R2(B), R3(B), W1(A), W2(B)

Conflicts on A

R2(A) before W1(A) gives T2 → T1. R1(A) and R2(A) are two reads and do not conflict.

Conflicts on B

R1(B) before W2(B) gives T1 → T2. R3(B) before W2(B) gives T3 → T2. The reads R1(B), R2(B) and R3(B) do not conflict with each other.

The graph

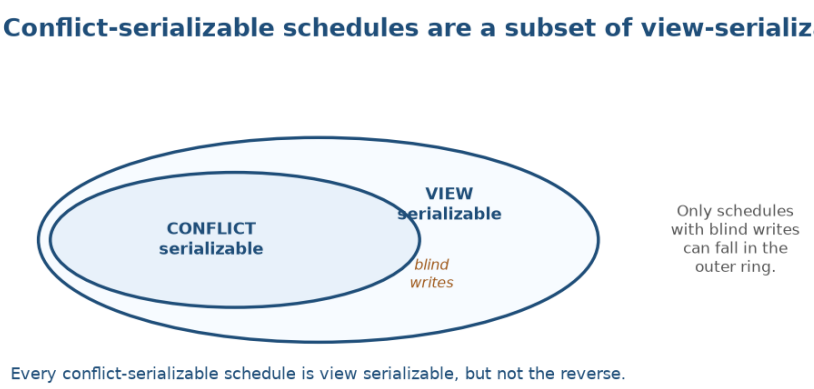
Edges are T2 → T1, T1 → T2 and T3 → T2. The pair T1 ↔ T2 forms a cycle, so the schedule is **not conflict serializable**.

View serializability

No blind write exists, because T1 read A before writing A and T2 read B before writing B. With no blind write and no conflict serializability, the schedule is **not view serializable** either.

Q9. Explain view serializability with its conditions and worked examples.

PYQ R24 Q1(m)



The definition

A schedule is view serializable if it is view equivalent to some serial schedule.

The three conditions

1. **Initial read** — the same transaction reads the initial value of each item in both schedules. 2. **Read-from** — if T_i reads X written by T_j in one schedule, the same holds in the other. 3. **Final write** — the same transaction performs the final write of each item in both.

The practical method

Take the transaction that performs any operation first, write all its operations together; take the next transaction that appears, write its operations below; repeat. That gives the candidate serial order. Then check the three conditions against it.

Worked example — view serializable

S = R1(A), W2(A), R3(A), W1(A), W3(A). The first operation belongs to T1, so the candidate order is T1 → T2 → T3. The initial read of A is by T1 and the final write is by T3 in both S and the serial order, and the read-from relations match. So S is **view serializable**.

Worked example — not view serializable

S = W1(X), W2(X), W1(X) is **view serializable** but not conflict serializable, because T1 performs the final write in both the schedule and the serial order T1 → T2. Change the order to W1(X), W1(X), W2(X) and it remains view serializable; but if the final writer in the schedule differs from the final writer of every candidate serial order, the schedule fails condition 3 and is not view serializable.

Q10. Compare conflict serializability with view serializability.

PYQ R25 Q1(m), B26 Q1(m)

Aspect	Conflict serializability	View serializability
Basis	Order of conflicting operations	Read-from relations and final writes
Testing	Precedence graph, simple	Three conditions, harder
Blind writes	Rejects some valid blind-write schedules	Accepts certain blind-write schedules
Relationship	A subset of view-serializable schedules	A superset of conflict-serializable schedules
Exam technique	Draw the graph and check for a cycle	Check the three conditions

The order of attack

Test conflict serializability first, because the graph is quick. Only if it fails **and** blind writes are present is it worth testing view serializability. That saves time and avoids mistakes.

The standard example that separates them

S = W1(X), W2(X), W1(X). On X, W1(X) before W2(X) gives T1 → T2, and W2(X) before W1(X) gives T2 → T1, so the graph has a cycle and the schedule is not conflict serializable. But no transaction reads X, so the read-from conditions are vacuous and T1 performs the final write in every equivalent order — so it **is** view serializable.

Q11. Explain recoverable, cascadeless and strict schedules.

Recoverable schedule

A schedule is recoverable if, whenever T_j reads a value written by T_i, the commit of T_i occurs before the commit of T_j. Otherwise T_j could commit a value whose writer later aborts, and no correct recovery action would exist.

Cascadeless schedule

A schedule is cascadeless if every transaction reads only items written by committed transactions. This prevents cascading rollback, where one abort forces the rollback of every reader and those rollbacks force further ones.

Strict schedule

A schedule is strict if a transaction neither reads nor writes an item until the last transaction that wrote it has committed or aborted. Recovery then needs only the before-image of each item, because no one else has touched it.

The hierarchy

Schedule type	Reads allowed from	Recovery consequence
Recoverable	Committed or uncommitted, but commit order respected	Cascading rollback may be needed

Schedule type	Reads allowed from	Recovery consequence
Cascadeless	Committed only	No cascading rollback
Strict	Committed only, and writes also blocked	Simple UNDO, no cascading rollback

Recoverable ⊇ cascadeless ⊇ strict. Most commercial systems implement strict two-phase locking, which produces strict schedules.

Q12. Explain deadlock — its causes, detection and prevention.

The cause

A deadlock occurs when two or more transactions are each waiting for a lock held by another. T1 holds X and waits for Y; T2 holds Y and waits for X; neither can proceed.

Detection

Build a **wait-for graph** with an edge $T_i \rightarrow T_j$ when T_i waits for a lock held by T_j. A cycle is a deadlock. The DBMS chooses a **victim** — usually the transaction with the least work done, or the one that has touched the fewest items — rolls it back, and releases its locks.

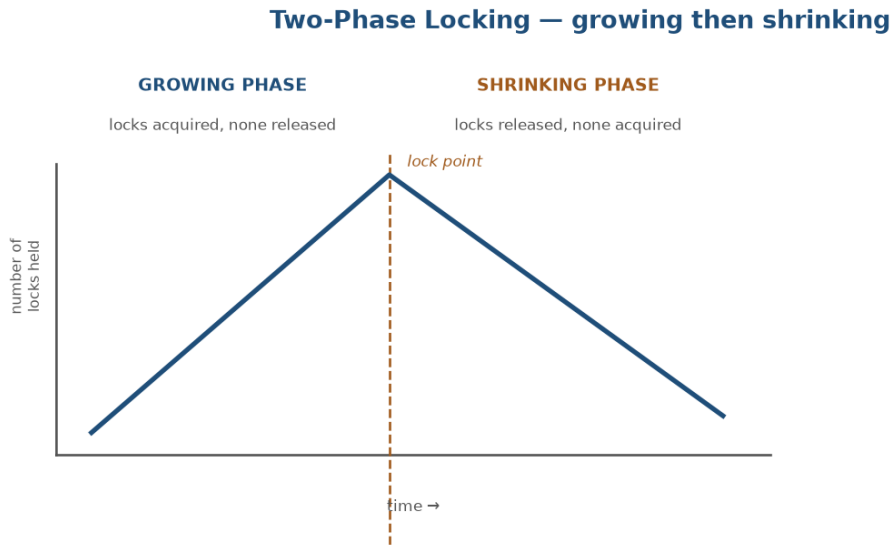
Prevention

Technique	Idea
Wait-die	Older transaction waits, younger one dies and restarts
Wound-wait	Older transaction aborts the younger one, younger waits
Timestamp ordering	Conflicts resolved by timestamp, so no circular wait arises
Timeout	A transaction waiting too long is aborted
Lock ordering	All transactions acquire locks in the same global order

The cost of each approach

Prevention avoids the deadlock but rolls back transactions that might have completed. Detection allows more concurrency but wastes work when a victim is chosen. Most systems combine a timeout with wait-for graph detection.

Q13. Explain lock-based protocols and two-phase locking.



Lock modes

A **shared lock (S)** allows reading and is compatible with other shared locks. An **exclusive lock (X)** allows reading and writing and is compatible with nothing.

Held / Requested	Shared	Exclusive
Shared	Compatible	Must wait
Exclusive	Must wait	Must wait

Why locks alone are not enough

Acquiring a lock before each access does not prevent the lost-update or unrepeatable-read problems, because a transaction can release a lock too early. A protocol is needed on top of the locks.

Two-phase locking

Every transaction acquires all its locks in a **growing phase** and releases them in a **shrinking phase**, with no lock acquired after the first release. The moment it holds all its locks is the **lock point**, and the lock points define the serialization order. If every transaction follows 2PL, every resulting schedule is conflict serializable.

The variants

Variant	Extra condition	Guarantee
Basic 2PL	Growing then shrinking	Conflict serializable
Conservative 2PL	Acquire all locks before starting	Deadlock free, low concurrency
Strict 2PL	Hold all exclusive locks until commit or abort	Serializable and strict
Rigorous 2PL	Hold all locks until commit or abort	Stronger still

The limitation

2PL guarantees serializability but does **not** prevent deadlock, so a deadlock detection or prevention mechanism is still needed alongside it.

Q14. Explain the timestamp-based concurrency control protocol.

Timestamp Ordering — the rules

Each transaction Ti carries a timestamp TS(Ti); every data item X keeps read_TS(X) and write_TS(X).

Operation of Ti	Condition	Action
Read(X)	TS(Ti) < write_TS(X)	Reject and roll back Ti
Read(X)	TS(Ti) ≥ write_TS(X)	Allow; set read_TS(X) = max(TS(Ti), read_TS(X))
Write(X)	TS(Ti) < read_TS(X)	Reject and roll back Ti
Write(X)	TS(Ti) < write_TS(X)	Ignore the write (Thomas rule)
Write(X)	otherwise	Allow; set write_TS(X) = TS(Ti)

The schedule is kept equivalent to the serial order of the timestamps, so no locks are needed.

The idea

Every transaction receives a unique timestamp TS(T_i) when it starts. The schedule is kept equivalent to the serial order of these timestamps, so **no locks are needed** and no deadlock can occur.

The data maintained

For each item X the DBMS stores read_TS(X), the largest timestamp of any transaction that read X, and write_TS(X), the largest timestamp of any transaction that wrote X.

The rules

Operation	Condition	Action
Read(X)	$TS(T_i) < write_TS(X)$	Reject and roll back T_i
Read(X)	$TS(T_i) \geq write_TS(X)$	Allow; $read_TS(X) = \max(TS(T_i), read_TS(X))$
Write(X)	$TS(T_i) < read_TS(X)$	Reject and roll back T_i
Write(X)	$TS(T_i) < write_TS(X)$	Ignore the write (Thomas rule)
Write(X)	otherwise	Allow; $write_TS(X) = TS(T_i)$

The trade-off

Timestamp ordering avoids deadlock entirely because no transaction ever waits, but transactions may be rolled back and restarted more often when conflicts are frequent. It suits systems where deadlock is expensive and conflicts are rare.

Q15. Explain the types of database failure and the requirements of recovery.

PYQ R25 Q1(o), R24 Q1(o), B26 Q1(o)

Failure	Examples	Recovery need
Transaction failure	Logical error, constraint violation, deadlock victim	Undo the incomplete transaction
System failure	Power loss, operating system crash, DBMS crash	Redo committed work, undo incomplete work
Media failure	Disk crash, corrupted database file	Restore from backup and apply archived logs
Communication failure	Network break in a distributed system	Resolve incomplete distributed work

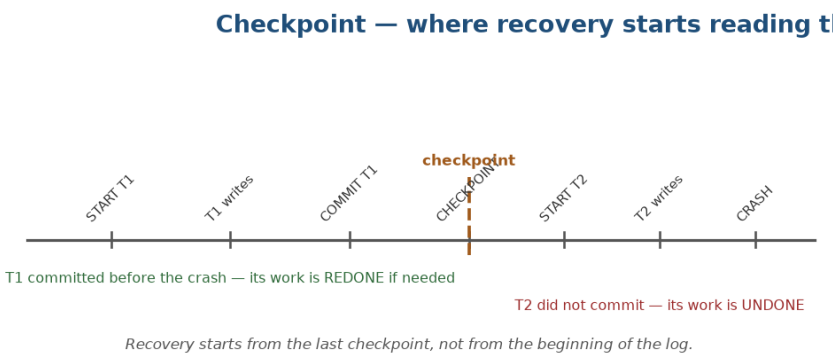
The requirements of recovery

Recovery must guarantee **atomicity**, so the effects of uncommitted transactions are undone, and **durability**, so the effects of committed transactions are redone. It must restore the database to the most recent consistent state, and it must use information that survives the failure — the log, checkpoints and backups.

Why the classification matters

Each type needs a different mechanism. A transaction failure needs only a rollback; a system failure needs the log; a media failure needs a backup plus archived logs. An answer that treats them alike misses the point of the question.

Q16. Explain log-based recovery with UNDO, REDO and checkpoints.



The log

A log is a sequential file on stable storage recording every change: START, update records holding the data item with its old and new values, COMMIT, ABORT and CHECKPOINT.

Write-ahead logging

The log record for a change must reach stable storage **before** the changed data page does. Otherwise a crash could leave a change in the database with no record to undo or redo it.

UNDO and REDO

Action	Meaning	Applied to
UNDO	Restore the old value	Transactions that did not commit
REDO	Reapply the new value	Transactions that committed but whose pages may not have reached disk

The decision rule is simple: a transaction with a COMMIT record in the log is redone; a transaction that started without a COMMIT record is undone.

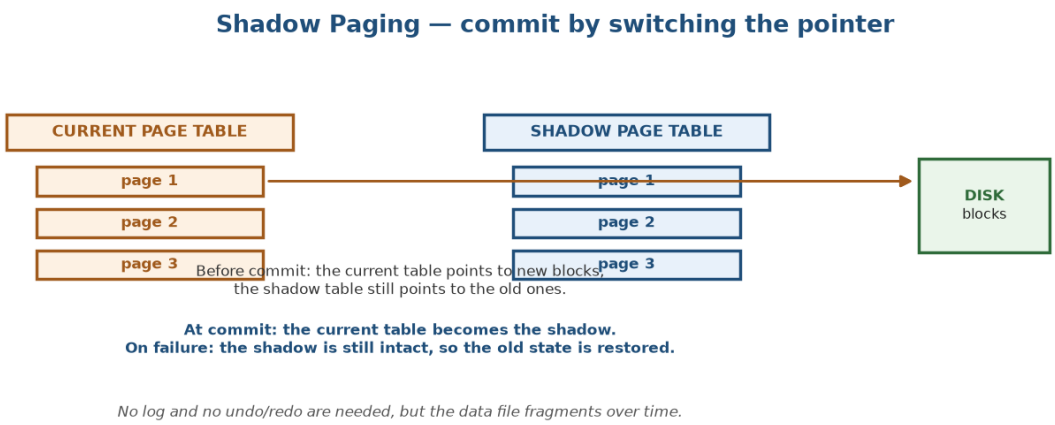
Checkpoints

At a checkpoint the DBMS forces all modified pages and log records to disk and writes a CHECKPOINT record. After a crash, recovery starts from the last checkpoint instead of the beginning of the log, which keeps restart time bounded as the log grows.

Worked scenario

Suppose the log holds START T1, T1 updates X, COMMIT T1, START T2, T2 updates Y, and then a crash. T1 committed, so its effect is redone if needed; T2 did not commit, so its effect is undone.

Q17. Explain shadow paging as a recovery technique.



The mechanism

The DBMS keeps two page tables. The **current page table** points to the pages being modified; the **shadow page table** points to the pages as they were at the last commit. Modified pages are written to **new** disk blocks, so the old blocks are never overwritten.

Commit and failure

At commit the current page table is written to disk and becomes the new shadow — a single atomic pointer switch makes all the changes permanent. If the system fails before that switch, the shadow table still points to the old pages, so the old state is restored by discarding the current table.

Advantages and disadvantages

Advantages	Disadvantages
No log records to write	Data fragments because pages are never overwritten in place
Recovery is very fast	Garbage collection of old pages is needed
Commit is a single atomic act	Difficult to combine with concurrent transactions

Comparison with log-based recovery

Log-based recovery handles concurrency naturally and is what commercial DBMSs use. Shadow paging is simpler and faster to recover but hard to extend to many concurrent transactions, so it remains mainly a textbook technique.