DATABASE ENGINEERING

Unit 5

Transaction Processing and Concurrency Control

Topic-wise study notes with importance ratings

34 topics — transactions and ACID, schedules and concurrency problems, serializability, protocols and recovery

Diagrams and solved schedules are given wherever a concept needs one

Topic-wise study notes. Every topic carries an importance rating from one to three stars: three stars means the topic is asked almost every year and must be known perfectly, two stars means it is regularly asked and should be prepared, one star means it is supporting material to read once. The notes explain the concept only; worked questions and answers are kept in a separate file.

Index

Topic, importance rating and page number. ★★★ must know, ★★ should know, ★ read once.

Topic	Importance	Page
5.1 Transaction — Concept	★★★	3
5.2 ACID Properties	★★★	3
5.3 Transaction States	★★★	4
5.4 Schedule — Definition and Notation	★★★	4
5.5 Serial Schedule	★★★	5
5.6 Non-Serial Schedule	★★★	5
5.7 Lost Update Problem	★★★	6
5.8 Dirty Read Problem	★★★	6
5.9 Unrepeatable Read Problem	★★★	7
5.10 Incorrect Summary Problem	★★	7
5.11 Phantom Read	★★	7
5.12 Serializability — Concept	★★★	8
5.13 Conflict Operations	★★★	8
5.14 Conflict Equivalence	★★	8
5.15 Conflict Serializability	★★★	9
5.16 Precedence Graph	★★★	9
5.17 Topological Order and Equivalent Serial Schedule	★★	10
5.18 View Serializability	★★★	10
5.19 View Equivalence — Worked Method	★★	10
5.20 Blind Write	★★★	10
5.21 Conflict versus View Serializability	★★★	11
5.22 Recoverable Schedule	★★★	11
5.23 Cascadeless Schedule	★★★	12
5.24 Strict Schedule	★★	12
5.25 Deadlock	★★★	12
5.26 Lock-Based Protocol	★★★	13
5.27 Two-Phase Locking (2PL)	★★★	13
5.28 Timestamp-Based Protocol	★★★	14
5.29 Types of Database Failure	★★★	15
5.30 Log-Based Recovery	★★★	15
5.31 UNDO and REDO	★★★	15
5.32 Checkpoint	★★★	16
5.33 Deferred and Immediate Update	★★	16
5.34 Shadow Paging	★★★	17

5.1 Transaction — Concept ★★★

A **transaction** is a logical unit of database work made up of one or more operations that must succeed or fail as a whole.

The standard 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.

Why it is a unit

If the system fails after Write(A) and before Write(B), ₹1,000 has vanished. Only by treating all six operations as one unit can the database be kept correct — either the whole transfer happens or none of it does.

The operations available

Operation	Meaning	Notation
Read(X)	Copy the database item X into a local variable	$R_i(X)$
Write(X)	Copy the local value back to the database item	$W_i(X)$
Commit	Make the transaction's changes permanent	C_i
Abort / Rollback	Cancel the transaction and restore the previous state	A_i

5.2 ACID Properties ★★★

Property	Meaning	Ensured by
Atomicity	All operations happen, or none happen	Rollback and undo of incomplete work
Consistency	The database moves from one valid state to another	Constraint checking
Isolation	Concurrent transactions behave as if run alone	Concurrency control and serializability
Durability	Committed changes survive any failure	Logging, checkpoints and redo

Why each one is needed

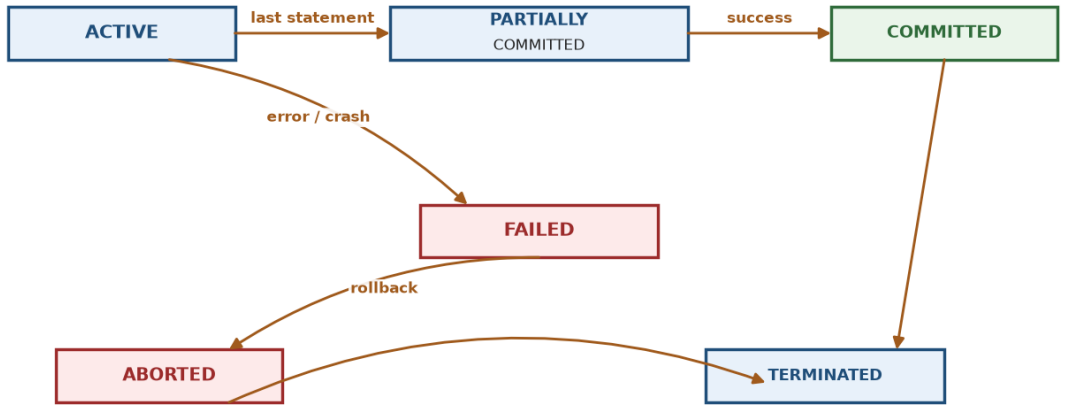
If the transfer fails after debiting A, **atomicity** undoes the debit, otherwise money disappears. **Consistency** guarantees the total of all balances is unchanged. **Isolation** stops another transaction from reading the half-finished state. **Durability** guarantees that once the customer is told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

The memory rule

Atomicity = all or nothing. Consistency = rules preserved. Isolation = safe concurrency. Durability = permanent after commit.

5.3 Transaction States ★★★

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 the 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.

Restart or terminate

An aborted transaction may be **restarted** if the failure was caused by something transient such as a deadlock or a system crash. If it failed because of a logic error or bad input, restarting would fail again, so it is terminated and the application is informed.

5.4 Schedule — Definition and Notation ★★★

A **schedule** is the chronological order in which the operations of one or more transactions are executed.

What a schedule preserves

The internal order of operations **within** each transaction. If T1 reads X before it writes X, that order must hold in every schedule containing T1.

Notation

$R_1(X)$ means transaction T1 reads X; $W_2(Y)$ means T2 writes Y; C_1 means T1 commits. A schedule is written as a comma-separated list: $R_1(X)$, $W_1(X)$, $R_2(X)$, $W_2(X)$.

The two families

Serial schedules run one transaction at a time; **non-serial** schedules interleave the operations of several transactions.

5.5 Serial Schedule ★★★

In a **serial schedule**, one transaction completes entirely before the next one starts.

Example

$R_1(X)$, $W_1(X)$, $R_1(Y)$, $W_1(Y)$, C_1, then $R_2(X)$, $W_2(X)$, C_2.

Its properties

Always **correct**, because no transaction can interfere with another. Simple to reason about. But it gives **no concurrency**: while T1 waits for disk I/O, T2 cannot use the CPU, so throughput and response time suffer badly.

Why it is still important

A serial schedule is the reference standard. A concurrent schedule is acceptable only if it is **equivalent** to some serial schedule — that is exactly what serializability means.

5.6 Non-Serial Schedule ★★★

In a **non-serial schedule**, the operations of several transactions are interleaved.

Why it is used

Better CPU and I/O utilisation, higher throughput, and shorter response time for short transactions that would otherwise wait behind a long one.

The danger

Some interleavings produce incorrect results — a lost update, a dirty read, an unrepeatable read or an incorrect summary. The interleaving must therefore be controlled.

The key point

Non-serial does not automatically mean incorrect. A non-serial schedule that is serializable is perfectly correct; the job of concurrency control is to allow exactly those.

5.7 Lost Update Problem ★★★

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.

A **lost update** occurs when two transactions read the same old value and one update overwrites the other.

The classic sequence

Both T1 and T2 read X = 100. T1 computes 120 and writes it. T2, still working from the old 100, computes 90 and writes it. The final value is 90, and T1's update has disappeared.

The expected result

After adding 20 and subtracting 10, the correct value is 110. The stored value is 90, so the database is wrong.

The cause and the cure

The cause is that T2 read a value that T1 had already changed but not yet committed over. The cure is isolation — either T2 must be prevented from reading X until T1 finishes, or T1 must be prevented from writing over T2's version.

5.8 Dirty Read Problem ★★★

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.

A **dirty read** occurs when a transaction reads data written by another transaction **before that transaction commits**.

The sequence

T1 writes X = 500 but has not committed. T2 reads X = 500 and acts on it. T1 then rolls back, so X returns to 100 — but T2 has already used a value that never officially existed.

Other names

The **uncommitted dependency** problem, or the **temporary update** problem.

The cure

A transaction must not read uncommitted data. In lock terms, T1 must hold an exclusive lock on X until it commits or aborts, and T2 must wait.

5.9 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 in between.

The sequence

T1 reads X = 100. T2 updates X to 120 and commits. T1 reads X again and now sees 120. Within one transaction, the same read has produced two different answers.

Why it is a problem

Any computation that depends on reading a value twice — a consistency check, a running total — becomes unreliable. The transaction has not been given the isolation it was promised.

5.10 Incorrect Summary Problem ★★

An **incorrect summary** occurs when an aggregate is computed while another transaction is updating some of the rows being summarised.

The sequence

T1 sums the balances of accounts A and B. It reads A before T2 transfers money from A to B, and reads B after the transfer. The total it computes mixes an old value with a new one and is therefore wrong, even though every individual read was of a committed value.

Other name

Inconsistent analysis.

Why it is subtle

No single read is dirty and no update is lost, so the problem is easy to miss. It is a violation of isolation at the level of the whole computation rather than of one item.

5.11 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.

Example

T1 counts the employees in department D1 and finds 10. T2 inserts a new employee in D1 and commits. T1 repeats the count and finds 11. No existing row was updated, so an unrepeatable read has not occurred — a **new** row appeared.

The cure

Range locking, or a serializable isolation level that locks the predicate rather than only the rows found.

5.12 Serializability — Concept ★★★

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.

Why it is the right criterion

A serial schedule is correct by definition. So any concurrent schedule that produces the same effect is also correct, and it may still be much faster than running the transactions one after another.

The two kinds

Conflict serializability, tested with a precedence graph, and **view serializability**, tested with read-from and final-write conditions. Conflict serializability is the stricter of the two.

5.13 Conflict Operations ★★★

Two operations **conflict** when all three of these hold: they belong to **different transactions**, they access the **same data item**, and **at least one of them is a write**.

Pair	Conflict	Reason
$R_1(X)$, $R_2(X)$	No	Read and read
$R_1(X)$, $W_2(X)$	Yes	Read and write
$W_1(X)$, $R_2(X)$	Yes	Write and read
$W_1(X)$, $W_2(X)$	Yes	Write and write
$R_1(X)$, $W_2(Y)$	No	Different data items

Why conflicting operations matter

Conflicting operations cannot be swapped without possibly changing the result. Non-conflicting operations can be swapped freely. That single fact is the basis of both conflict equivalence and the precedence graph.

5.14 Conflict Equivalence ★★

Two schedules are **conflict equivalent** if they contain the same operations and preserve the order of every pair of **conflicting** operations.

Example

S = $R_1(X)$, $W_1(X)$, $R_2(X)$, $W_2(X)$. The conflicting pairs are $W_1(X)$ before $R_2(X)$ and $W_1(X)$ before $W_2(X)$. The serial order T1 → T2 preserves both, so S is conflict equivalent to that serial schedule.

The practical consequence

Because only conflicting pairs must be preserved, a schedule can be transformed into a serial one by repeatedly swapping adjacent **non-conflicting** operations. If that transformation succeeds, the

schedule is conflict serializable.

5.15 Conflict Serializability ★★★

A schedule is **conflict serializable** if it is conflict equivalent to some serial schedule.

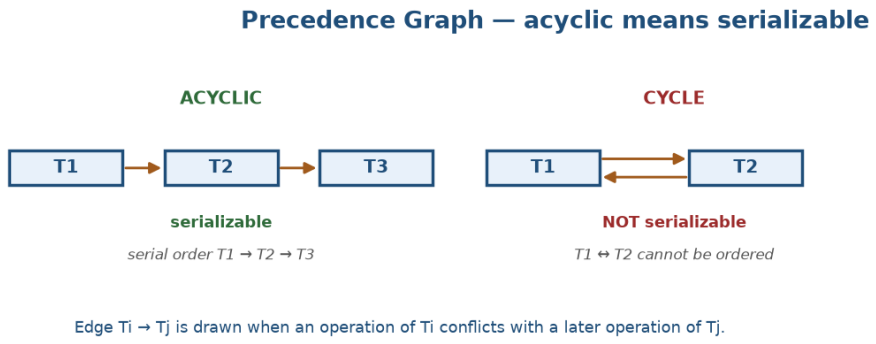
Two ways to test it

Method	How	When to use
Swapping	Repeatedly swap adjacent non-conflicting operations until a serial order is reached	Small schedules, or to demonstrate equivalence
Precedence graph	Draw edges for conflicts and look for a cycle	The standard method, and the one to use in an exam

The key theorem

A schedule is conflict serializable **if and only if** its precedence graph is acyclic. This is the single most examinable sentence in the unit.

5.16 Precedence Graph ★★★



A **precedence graph** (serialization graph) is a directed graph G = (V, E) where V is the set of transactions and E is the set of conflict edges.

The construction

1. Create one node for each transaction. 2. Find every pair of conflicting operations. 3. If an operation of T_i occurs **before** the conflicting operation of T_j, draw the edge $T_i \rightarrow T_j$. 4. Inspect the graph for a directed cycle.

The conclusion

Acyclic = conflict serializable. Cycle = not conflict serializable. For an acyclic graph, any topological order gives an equivalent serial schedule.

Common mistakes

Treating two reads as a conflict; drawing an edge for operations on different data items; ignoring write-write conflicts; drawing the edge in the wrong direction; and looking at only one conflict and missing the reverse edge that creates the cycle.

5.17 Topological Order and Equivalent Serial Schedule ★★

When the precedence graph is acyclic, a **topological ordering** of its nodes gives an equivalent serial schedule.

Example

Edges T1 → T3 and T2 → T3. Both T1 and T2 must precede T3, but there is no edge between them, so both orders are valid: T1 → T2 → T3 and T2 → T1 → T3.

Why more than one order is possible

An acyclic graph need not be a single chain. The number of valid serial orders equals the number of topological orderings, which is why an answer that gives only one of them is still correct but an answer that claims it is the only one is not.

5.18 View Serializability ★★★

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

Why it was introduced

Because conflict serializability is **stricter than necessary**. Some schedules that do not preserve the order of conflicting operations still produce exactly the same final state, and rejecting them loses useful concurrency.

The three conditions

Two schedules are view equivalent when: 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.

5.19 View Equivalence — Worked Method ★★

The practical procedure

1. Take the transaction that performs any operation **first** in the schedule, and write all of its operations together. 2. Take the transaction that appears next, and write its operations below. 3. Repeat for all transactions, keeping the order. 4. Compare the resulting serial schedule with the given schedule on the three conditions.

Worked example

S = $R_1(A)$, $W_2(A)$, $R_3(A)$, $W_1(A)$, $W_3(A)$. The first operation belongs to T1, so the serial order starts with T1, then T2, then T3. In S, the initial read of A is by T1 and the final write is by T3 — the same holds in the serial order T1 → T2 → T3, and the read-from relations match. So S is view serializable with serial order T1 → T2 → T3.

A schedule that fails

If the final writer in the schedule differs from the final writer in every candidate serial order, the schedule is not view serializable and must be rejected.

5.20 Blind Write ★★★

A **blind write** is a write performed by a transaction **without first reading** that data item.

Why it matters

A blind write can make a schedule view serializable even though it is not conflict serializable. This is the only situation in which the two notions differ.

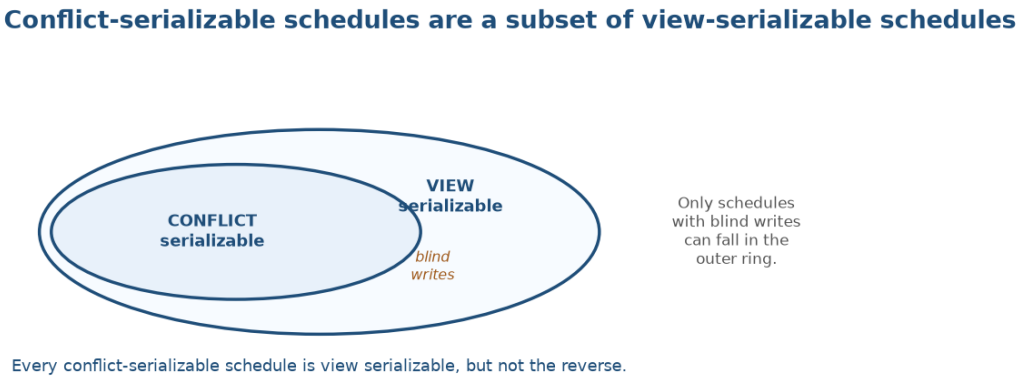
The rule to apply in an exam

Situation	Conclusion
Schedule is conflict serializable	It is definitely view serializable — stop
Not conflict serializable, and no blind write	Not view serializable either — stop
Not conflict serializable, and blind writes present	Check the three view conditions

The standard example

S = $W_1(X)$, $W_2(X)$, $W_1(X)$. On X, $W_1(X)$ before $W_2(X)$ gives T1 → T2, and $W_2(X)$ before $W_1(X)$ 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.

5.21 Conflict versus View Serializability ★★★



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 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

Always 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.

5.22 Recoverable Schedule ★★★

A schedule is **recoverable** if, whenever a transaction T_j reads a value written by T_i, the commit of T_i occurs **before** the commit of T_j.

Why it is required

If T_j were allowed to commit first and T_i then aborted, T_j would have committed a value that no longer exists, and it could not be undone without violating durability. A non-recoverable schedule therefore cannot be rolled back safely.

Example of a non-recoverable schedule

T_1 writes X, T_2 reads X, T_2 commits, and then T_1 aborts. T_2 has committed a dirty value, and there is no correct recovery action.

The hierarchy

Recoverable ⊇ cascadeless ⊇ strict. Each is stronger than the one before.

5.23 Cascadeless Schedule ★★★

A schedule is **cascadeless** (avoids cascading rollback) if every transaction reads only items written by **committed** transactions.

The problem it prevents

In a recoverable but cascadable schedule, an abort of T_1 forces the rollback of every transaction that read from it, and those rollbacks may force further rollbacks — a cascade that can unwind much of the system.

The trade-off

Cascadeless schedules are safer but restrict concurrency more, because a reader must wait for the writer to commit rather than merely to finish writing.

5.24 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.

Why strictness helps

It makes recovery simple: an aborted transaction's changes can be undone by restoring the before-image of each item, because no one else has touched those items in the meantime.

The practical result

Most commercial systems implement strict two-phase locking, which produces strict schedules, and therefore need only UNDO — never a cascading rollback.

5.25 Deadlock ★★★

A **deadlock** occurs when two or more transactions are each waiting for a lock held by another, so none of them can proceed.

Example

T_1 holds a lock on X and waits for Y; T_2 holds a lock on Y and waits for X. Neither will ever be released.

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 in this graph is a deadlock. The DBMS then chooses a **victim** — usually the transaction with the least work done — and rolls it back.

Prevention

Technique	Idea
Wait-die	Older transaction waits, younger one dies and restarts
Wound-wait	Older transaction wounds (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

5.26 Lock-Based Protocol ★★★

In a lock-based protocol, a transaction must acquire a lock on a data item before accessing it.

Lock mode	Allows	Compatible with
Shared (S)	Reading	Other shared locks
Exclusive (X)	Reading and writing	Nothing else

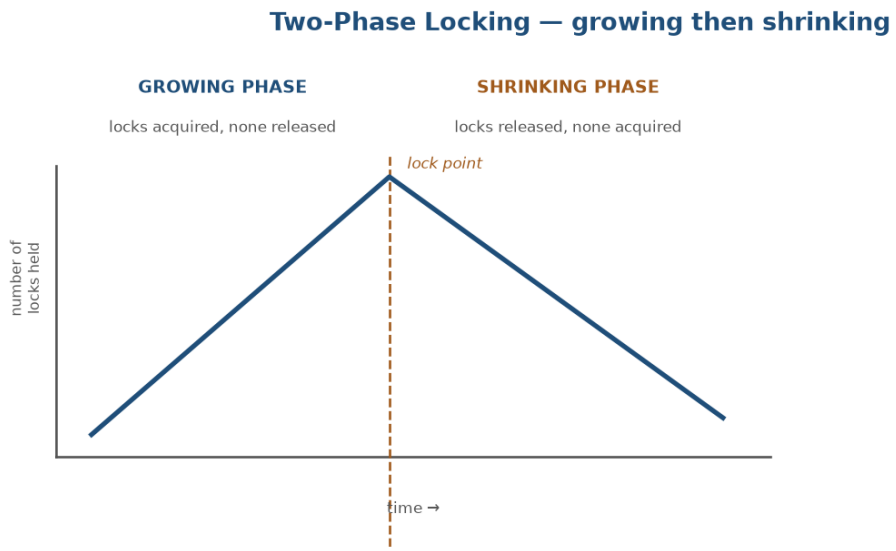
The compatibility matrix

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

What locks alone do not guarantee

Acquiring a lock before each access is not enough — it 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, and the standard one is two-phase locking.

5.27 Two-Phase Locking (2PL) ★★★



Two-phase locking requires every 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.

Phase	Rule
Growing	Locks may be acquired, none may be released

Phase	Rule
Lock point	The moment the transaction holds all its locks
Shrinking	Locks may be released, none may be acquired

What 2PL guarantees

If every transaction follows 2PL, every resulting schedule is **conflict serializable**. The lock point of each transaction defines the serialization order.

The variants

Variant	Extra condition	Guarantee
Basic 2PL	Growing then shrinking	Conflict serializable
Conservative (static) 2PL	Acquire all locks before starting	Deadlock free, low concurrency
Strict 2PL	Hold all exclusive locks until commit or abort	Serializable and strict, avoids cascading rollback
Rigorous 2PL	Hold all locks until commit or abort	Stronger still, easiest to reason about

The limitation

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

5.28 Timestamp-Based 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.

In a timestamp protocol, every transaction is given a timestamp $TS(T_i)$ when it starts, and the schedule is kept equivalent to the serial order of those timestamps. No locks are used.

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

A read or write is allowed only if it does not violate the timestamp order; otherwise the offending transaction is rolled back and restarted with a new timestamp.

The Thomas write rule

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. This allows some schedules that the strict rule would reject.

The trade-off

Timestamp ordering avoids deadlock entirely, because no transaction ever waits. The cost is that transactions may be rolled back and restarted more often when conflicts are frequent.

5.29 Types of Database Failure ★★★

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 logs
Communication failure	Network break in a distributed system	Resolve incomplete distributed work

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 lumps them together misses the point of the question.

5.30 Log-Based Recovery ★★★

A **log** is a sequential file on stable storage recording every change made by every transaction. It is the basis of recovery after a system failure.

Log record	Meaning
<T_i START>	The transaction has begun
<T_i, X, old, new>	T_i changed X from old to new
<T_i COMMIT>	The transaction has committed
<T_i ABORT>	The transaction has been rolled back
<CHECKPOINT>	A recovery marker

The two rules

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

Why both old and new values are stored

The **old value** supports UNDO; the **new value** supports REDO. Storing both lets a single log serve both purposes.

5.31 UNDO and REDO ★★★

Action	Meaning	Applied to
UNDO	Reverse the effect of a change, restoring the old value	Transactions that did not commit
REDO	Reapply a change, setting the new value	Transactions that committed but whose pages may not have reached disk

The decision rule after a crash

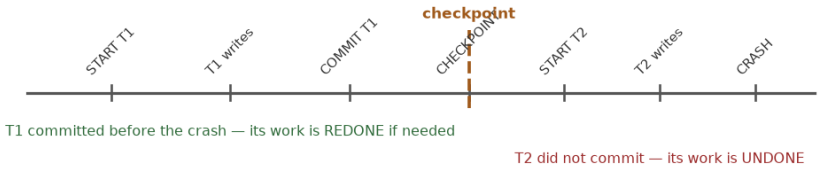
A transaction that has a COMMIT record in the log is **redone**. A transaction that started but has no COMMIT record is **undone**.

Example

T_1 wrote X from 10000 to 9000 and did not commit: recovery restores 10000. T_2 committed X = 12000 but the page was not written before the crash: recovery reapplies 12000.

5.32 Checkpoint ★★★

Checkpoint — where recovery starts reading the log



Recovery starts from the last checkpoint, not from the beginning of the log.

A **checkpoint** is a recovery marker that limits how much of the log must be examined after a failure.

What happens at a checkpoint

The DBMS forces all modified pages and all log records to disk, then writes a CHECKPOINT record listing the transactions that are still active.

The effect on recovery

After a crash, recovery starts from the last checkpoint rather than from the beginning of the log, so it does not have to redo work that was already safely on disk. This is what keeps restart time bounded as the log grows.

Why it costs something

Writing every dirty page to disk takes time and competes with normal processing, which is why checkpoints are taken periodically rather than continuously, and why some systems use fuzzy checkpoints that do not pause the system.

5.33 Deferred and Immediate Update ★★

Approach	Basic idea	Recovery emphasis
Deferred update	Changes are not written to the database until the transaction commits	Primarily REDO of committed transactions; no UNDO needed

Approach	Basic idea	Recovery emphasis
Immediate update	Changes may be written to the database before the transaction commits	Both UNDO for uncommitted work and REDO for committed work

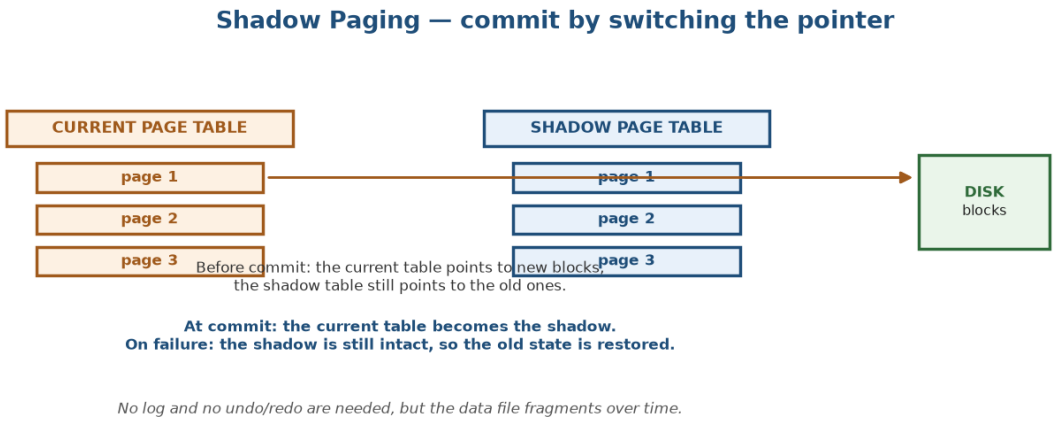
Why deferred update is simpler

Because nothing uncommitted ever reaches the database, a failed transaction simply leaves no trace and there is nothing to undo. The price is that all its changes must be held in memory or a local workspace until commit.

Why immediate update is used

Because holding every change in memory is impractical for a large transaction, most real systems write pages early and rely on the log to undo if the transaction fails.

5.34 Shadow Paging ★★★



Shadow paging is a recovery technique that needs neither a log nor UNDO/REDO.

How it works

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 simply 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

Why it is not universal

Shadow paging works well for a single-user system but is hard to extend to many concurrent transactions, which is why log-based recovery is the norm in commercial DBMSs and shadow paging is

mainly a textbook technique.