Database Management Systems

Isolation, logs and recovery

The notes for the lesson “Isolation, logs and recovery”: chapters 5.7 to 5.9, with their figures, worked examples, exam traps, quizzes and exam question bank, in one file.

3 chapters | Chapters 5.7 to 5.9

About these notes

These are the notes for the University free n easy video lesson **Isolation, logs and recovery** on Database Management Systems: chapters 5.7 to 5.9, with every figure, worked example, exam trap and quiz, and the long-answer exam questions on these chapters.

5.7 Isolation levels and multiple granularity locking

Full serializability makes transactions wait, so SQL lets a transaction choose a weaker **isolation level** and accept some anomalies for speed. And a transaction reading a whole table should not lock a million rows one by one: **multiple granularity locking** lets it take one table lock while others lock single rows.

The four SQL isolation levels

SQL defines the levels by three anomalies. A **dirty read** sees a value another transaction has not yet committed. A **non-repeatable read** reads one row twice and gets different committed values because another transaction updated it in between. In a **phantom**, a query re-run with the same WHERE condition finds new rows another transaction inserted.

LEVEL	DIRTY READ	NON-REPEATABLE READ	PHANTOM
READ UNCOMMITTED	possible	possible	possible
READ COMMITTED	no	possible	possible
REPEATABLE READ	no	no	possible
SERIALIZABLE	no	no	no

A transaction chooses with `SET TRANSACTION ISOLATION LEVEL READ COMMITTED;`. The standard's default is SERIALIZABLE, but PostgreSQL and Oracle default to READ COMMITTED and MySQL InnoDB to REPEATABLE READ. With locks, every level above READ UNCOMMITTED holds X locks to commit; READ COMMITTED frees each S lock after the read, REPEATABLE READ holds S locks to commit, and SERIALIZABLE also locks the key ranges a query searched.

Multiple granularity and intention locks

Data items form a tree: database, then areas, then files (tables), then records. Locking a node **implicitly** locks everything below it. Before granting S on a whole file, the DBMS must know that no record in it is locked X, and checking every record is too slow. So a transaction first marks each ancestor with an **intention lock**: **IS** (will lock something below in S), **IX** (will lock something below in X) or **SIX** (S on this whole node plus IX: read all of it, update a few rows).

REQUESTED \ HELD	IS	IX	S	SIX	X
IS	yes	yes	yes	yes	no
IX	yes	yes	no	no	no
S	yes	no	yes	no	no
SIX	yes	no	no	no	no
X	no	no	no	no	no

Rules: lock the root first; lock a node in S or IS only while holding IS or IX on its parent; lock a node in X, SIX or IX only while holding IX or SIX on its parent; follow 2PL; unlock a node only when none of its children is locked. So locks go on top-down and come off bottom-up. When a transaction holds too many record locks in one table, the DBMS may swap them for one table lock (**lock escalation**).

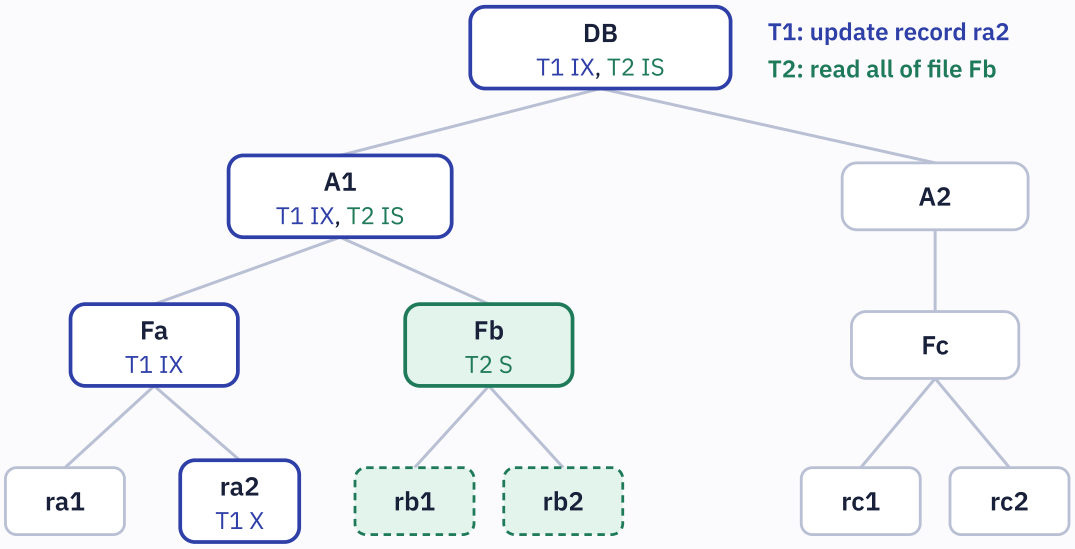


Figure 5.7.1 Two transactions on a granularity tree. T1 (blue) puts IX on every ancestor of ra2, then X on ra2; T2 (green) puts IS on DB and A1 and one S lock on Fb, which covers rb1 and rb2 (dashed). Notice that IX and IS share DB and A1 without conflict.

WORKED EXAMPLE

Question. With the locks of Figure 5.7.1 held, which requests are granted at once? (i) T3 wants X on file Fa. (ii) T4 wants S on the whole database. (iii) T5 wants to read record rb1. (iv) T6 wants to update record rc1. (4 marks)

1. (i) T3 gets IX on DB and A1 (IX is compatible with IX and IS), but X on Fa clashes with T1's IX there: **T3 waits**.
2. (ii) S on DB clashes with T1's IX on DB: **T4 waits**.
3. (iii) IS on DB, A1 and Fb are compatible with IX, IS and S; then S on rb1: **granted**.
4. (iv) IX on DB, IX on A2, IX on Fc, X on rc1: nothing conflicts, so **granted**.

EXAM TRAPS

- **Non-repeatable read vs phantom:** the first is an existing row changing value; the second is new rows appearing for a condition.
- **IX is compatible with IX.** Two transactions may both intend to update different rows of one table.
- **Order:** lock root to leaf, unlock leaf to root.

CHECK YOURSELF

- REMEMBER **List the four SQL isolation levels from weakest to strongest.** 2 marks

  Read uncommitted, read committed, repeatable read, serializable.

- UNDERSTAND **Why are intention locks needed?** 3 marks

  Without them, locking a whole table would mean checking every record below it for conflicting locks; intention locks on the ancestors show the conflict at once.

- APPLY **Under READ COMMITTED, T1 reads a balance of ₹5,000; T2 sets it to ₹4,000 and commits; T1 reads it again. What does T1 see, and what would it see under REPEATABLE READ?** 2 marks

  READ COMMITTED: **₹4,000** (a non-repeatable read). REPEATABLE READ: **₹5,000**; with locks, T2's update waits until T1 ends.

- APPLY **Which locks does a transaction take to read all of table Fc and update one of its rows?** 2 marks

  IX on DB, IX on A2, **SIX** on Fc, then X on the row.

- ANALYZE **A dashboard shows today's sales count every minute. Which isolation level would you choose, and what risk do you accept?** 3 marks

  READ COMMITTED: no uncommitted sales are shown and billing is not blocked. A count a moment out of date is harmless when it refreshes every minute.

BEYOND THE SYLLABUS

This is not for the exam. Many systems isolate with snapshots, not locks. **Snapshot isolation** avoids the three standard anomalies yet allows **write skew**; PostgreSQL's SERIALIZABLE adds checks to catch it. Search next: "A Critique of ANSI SQL Isolation Levels" and "write skew snapshot isolation".

5.8 Log-based recovery and write-ahead logging

The **log** is a sequential, append-only record of every change made to the database: which transaction changed which item, from what value to what value, and when each transaction started, committed or aborted. After a crash, the recovery system reads the log to **undo** the

changes of transactions that had not committed (atomicity) and to **redo** the changes of those that had (durability). **Write-ahead logging** (WAL) is the rule that makes this possible: the log record of a change reaches stable storage before the change itself reaches the database on disk.

You need: read, write and output (Chapter 5.1); atomicity and durability (Chapter 5.2).

Failures and storage

Three kinds of failure matter. A **transaction failure** stops one transaction: a logical error (bad input) or a system error (chosen as a deadlock victim). A **system crash** (power cut, operating-system or DBMS fault) wipes main memory but leaves the disks intact. A **disk failure** destroys data on a disk; it is repaired from a backup copy plus the log kept elsewhere. Storage is likewise **volatile** (main memory, lost in a crash), **non-volatile** (disk and SSD) or **stable** (never lost, approximated by keeping copies on separate devices). The log lives on stable storage.

Log records, undo and redo

- **<T_i start>**: T_i has started.
- **<T_i, X, V_1, V_2>**: T_i wrote item X, changing it from the old value V_1 to the new value V_2.
- **<T_i commit>** and **<T_i abort>**: T_i has committed or aborted.

undo(T_i) scans T_i's update records **backward**, setting each item back to its old value; for each it writes a redo-only record <T_i, X, V_1>, and at the end <T_i abort>. **redo(T_i)** scans them **forward**, setting each item to its new value. Both are **idempotent**: doing them twice has the same effect as once. That matters, because the system can crash again in the middle of recovery.

After a crash, scan the log. If it has <T_i start> and <T_i commit>, **redo** T_i. If it has <T_i start> but neither commit nor abort, **undo** T_i. (A transaction that had already aborted before the crash has its undo steps in the log as redo-only records, so replaying them forward finishes its rollback.)

Undo logging, redo logging and undo/redo logging

How much the log must hold depends on when changed blocks may reach the disk. Two buffer policies decide it. **Steal** lets the buffer manager write a block changed by an uncommitted transaction (so undo may be needed); **no-steal** forbids it. **Force** writes all of a transaction's blocks to disk at commit (so redo is never needed); **no-force** does not.

SCHEME	LOG RECORD KEEPS	RULE	AFTER A CRASH
Undo logging (immediate modification, steal + force)	old value	Log record on disk before the changed block; all changed blocks on disk before <T commit>	Undo uncommitted, scanning backward; nothing to redo
Redo logging (deferred modification, no-steal + no-force)	new value	No changed block reaches disk until <T commit> and all of T's records are on disk	Redo committed, scanning forward; nothing to undo
Undo/redo logging (immediate modification, steal + no-force)	old and new values	Log record on disk before the changed block (WAL); commit may come before or after	Redo committed forward, then undo uncommitted backward

In Silberschatz's terms, redo logging is **deferred database modification** (writes wait until commit) and the other two are **immediate database modification**. Real systems use undo/redo logging with steal and no-force, because it lets the buffer manager write any block at any time and never forces a slow flush at commit.

The write-ahead logging rules

1. T_i enters the committed state only after $<T_i$ commit> has been written to stable storage.
2. Before $<T_i$ commit> is written to stable storage, all other log records of T_i must be there.
3. Before a block in main memory is written to the database on disk, every log record about data in that block must be on stable storage.

Log records first collect in a **log buffer** in memory and are forced to disk together, so one disk write can carry the commit records of many transactions (**group commit**).

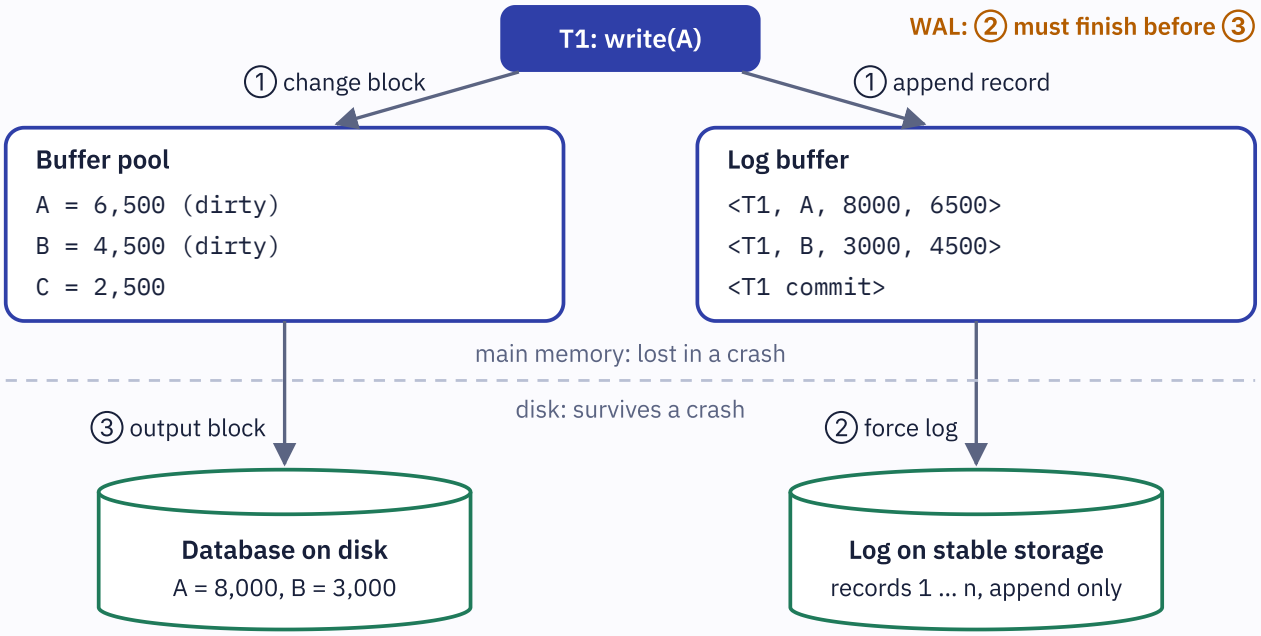


Figure 5.8.1 The path of one update under write-ahead logging. Step ① changes A in the buffer and appends its log record; the disk still holds the old A. Notice that the log record must reach stable storage (②) before the block holding A may be written to the database (③); if the power fails between them, the log still has the old value needed to undo.

WORKED EXAMPLE

Question. A = ₹8,000, B = ₹3,000 and C = ₹2,500. The system uses undo/redo logging. T1 moves ₹1,500 from A to B; T2 withdraws ₹400 from C. The log is: (1) <T1 start> (2) <T1, A, 8000, 6500> (3) <T1, B, 3000, 4500> (4) <T1 commit> (5) <T2 start> (6) <T2, C, 2500, 2100> (7) <T2 commit>. Give the recovery actions and final values if the crash comes just after record (a) 3, (b) 6, (c) 7 reaches the disk. (9 marks)

CRASH AFTER	T1	T2	ACTIONS IN ORDER	A, B, C
(a) record 3	start, no commit: undo	not started	B := 3000, A := 8000 (backward); log <T1, B, 3000>, <T1, A, 8000>, <T1 abort>	8,000, 3,000, 2,500
(b) record 6	commit: redo	start, no commit: undo	Redo forward: A := 6500, B := 4500. Undo: C := 2500; log <T2, C, 2500>, <T2 abort>	6,500, 4,500, 2,500
(c) record 7	commit: redo	commit: redo	A := 6500, B := 4500, C := 2100	6,500, 4,500, 2,100

1. Check (a): A + B = 11,000 before and after, and T1 left no trace: atomicity holds.
2. Check (b): T1's transfer survives (durability) though the disk may have held only old values; T2's withdrawal is gone (atomicity).
3. Under redo logging (deferred modification), uncommitted values never reach the disk, so (a) needs no action and (b) needs only the redo of T1. The final values are the same.

EXAM TRAPS

- **Undo backward, redo forward.** If T updates A twice, undoing forward would leave the middle value.
- **Redo every committed transaction**, even if you think its blocks reached the disk. Recovery cannot tell, and redo is idempotent.
- **WAL means log before data**, never data before log.
- **Deferred modification needs no old values**; immediate modification needs them for undo.
- **The commit point is the commit record on stable storage**, not the moment the data blocks reach the disk.

CHECK YOURSELF

– REMEMBER **List the types of log records used in log-based recovery.** 2 marks

<Ti start>, <Ti, X, old, new>, <Ti commit>, <Ti abort> (and redo-only <Ti, X, old> written during undo).

– REMEMBER **State the three write-ahead logging rules.** 3 marks

Commit only after the commit record is on stable storage; all of a transaction's log records reach stable storage before its commit record; all log records about a block reach stable storage before that block is written to the database.

– UNDERSTAND **Why must undo and redo be idempotent?** 2 marks

The system may crash during recovery and run recovery again; repeating an undo or redo must leave the same result as doing it once.

– APPLY **Log: <T1 start>, <T1, X, 50, 80>, <T2 start>, <T2, Y, 20, 35>, <T1 commit>, <T2, X, 80, 90>, crash. Give the final X and Y after undo/redo recovery.** 3 marks

Redo T1: X = 80. Undo T2 backward: X = 80, then Y = 20. Final **X = 80, Y = 20**.

– APPLY **A DBMS uses steal and no-force. Which of undo and redo does its recovery need?** 2 marks

Both. Steal can put uncommitted data on disk (needs undo); no-force can leave committed data off disk (needs redo).

– ANALYZE **Compare undo logging and redo logging for a transaction that changes 10,000 blocks.** 4 marks

Undo logging forces all 10,000 blocks to disk before commit, so commit is slow, but the buffer may write them early (steal). Redo logging commits after one log write, but must keep all 10,000 changed blocks in memory until commit (no-steal), which may not fit. Undo/redo logging avoids both limits.

BEYOND THE SYLLABUS

This is not for the exam. The log has become the backbone of modern databases. PostgreSQL ships its WAL to standby servers for replication and keeps archived WAL for point-in-time recovery. SQLite offers a WAL mode in which readers keep working while a writer appends to the log. Some cloud databases send only the log to their storage layer and rebuild pages from it there. Search next: "PostgreSQL WAL point-in-time recovery" and "SQLite write-ahead logging".

5.9 Checkpoints, shadow paging and ARIES recovery

Log-based recovery (Chapter 5.8) works but, without help, scans the whole log. A **checkpoint** limits how far back recovery must look. **Shadow paging** is a different design that needs no undo or redo at all. **ARIES** is the recovery algorithm behind most commercial systems.

Checkpoints

A simple checkpoint pauses updates and (1) writes all log records in memory to stable storage, (2) writes all modified buffer blocks to disk, (3) writes the record **<checkpoint L>**, where L lists the transactions active at that moment. On recovery, find the last checkpoint record. Transactions that committed before it need nothing, since their blocks are on disk. Only transactions in L or started after the checkpoint matter: redo those with a commit record, undo those without.

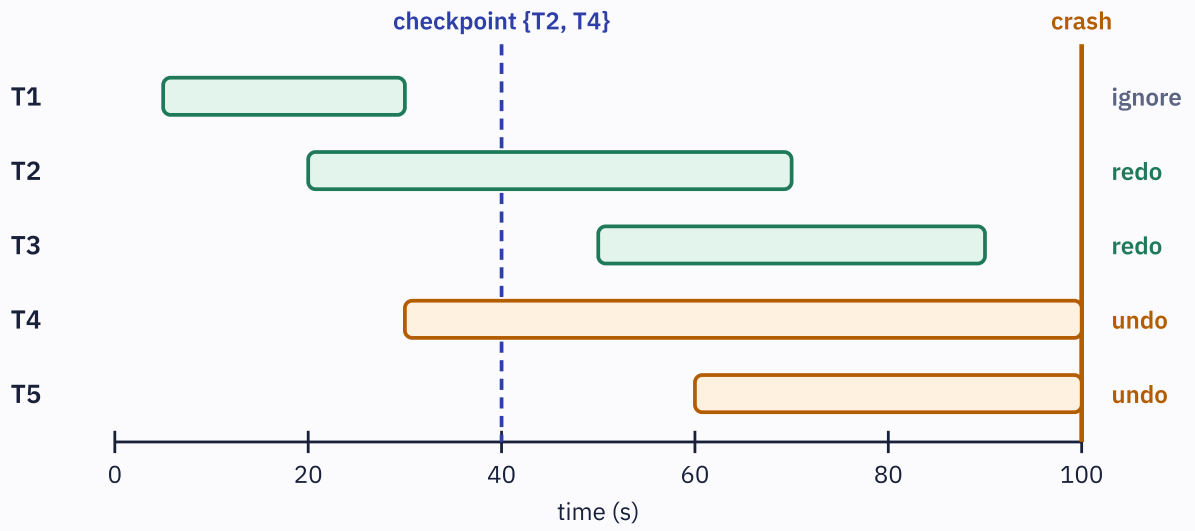
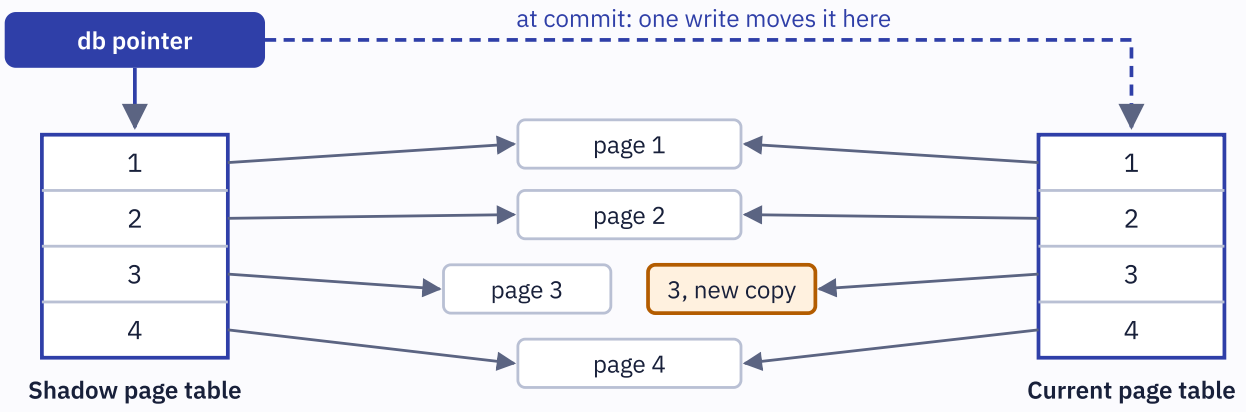


Figure 5.9.1 Five transactions around a checkpoint at 40 s and a crash at 100 s; green bars committed before the crash, orange bars were still running. Notice that T1 needs nothing, because the checkpoint already wrote its blocks to disk; T4 is in the checkpoint's list, so recovery must look back past the checkpoint to undo it.

Shadow paging

The database is a set of pages found through a **page table**. When a transaction starts, the table on disk becomes the **shadow page table** and is never changed; the transaction works with a **current page table**. The first write to a page copies it to a free page and points the current table at the copy. To commit: write the changed pages and the current table to disk, then overwrite the one **db pointer** at a fixed disk address so it points to the current table. That single write is the commit point. After a crash the pointer still names the shadow table, so no undo or redo is needed. The costs: copying and writing the page table, pages scattered over the disk, garbage collection of old pages, and poor support for concurrent transactions.



The shadow table is never changed by the transaction.
The current table's entry 3 points to the copied, updated page.

Figure 5.9.2 Shadow paging after the transaction has updated page 3. Notice that the old page 3 is untouched and still reachable from the shadow table; until the db pointer is switched, a crash simply leaves the database as it was.

ARIES

ARIES (Mohan and others, IBM, 1992) uses undo/redo logging with steal and no-force. Each log record has a **log sequence number (LSN)**; each page stores the LSN of the last update applied to it (**PageLSN**); a **dirty page table** gives, for each changed page not yet on disk, the first LSN that dirtied it (**RecLSN**). Undo writes **compensation log records (CLRs)**, so undo work is never repeated after a second crash. **Fuzzy checkpoints** record the transaction and dirty page tables without stopping work. Recovery has three passes:

1. **Analysis**, forward from the last checkpoint: rebuild both tables; transactions still active at the crash are the **losers**; RedoLSN = the smallest RecLSN.
2. **Redo**, forward from RedoLSN: **repeat history**, losers included. Skip an update if its page is not in the dirty page table, its LSN < the page's RecLSN, or the page's PageLSN ≥ its LSN.
3. **Undo**, backward: roll back the losers, writing CLRs.

WORKED EXAMPLE

Question. For Figure 5.9.1 (checkpoint at 40 s with L = {T2, T4}, crash at 100 s), say what recovery does with each transaction. (5 marks)

1. T1 ran 5–30 s and committed before the checkpoint: **nothing**.
2. T2 (20–70 s, in L) and T3 (50–90 s, after the checkpoint) have commit records: **redo** both.
3. T4 (from 30 s, in L) and T5 (from 60 s) have no commit record: **undo** both, backward. T4's undo reaches back to 30 s, before the checkpoint.

EXAM TRAPS

- **Transactions committed before the checkpoint are neither redone nor undone.**
- **Shadow paging's commit point is the db pointer write**, not the page writes.
- **ARIES redoes losers too**, then undoes them.

CHECK YOURSELF

– REMEMBER **Name the three passes of ARIES.** 2 marks

Analysis, redo, undo.

– UNDERSTAND **Why does shadow paging need neither undo nor redo?** 2 marks

Before commit, the disk's db pointer names the untouched shadow table; commit is one atomic pointer write, after which all new pages are already on disk.

– APPLY **Checkpoint L = {T6}. After it, T7 starts and commits, and T8 starts; T6 commits; then a crash. Classify T6, T7, T8.** 3 marks

Redo **T6** and **T7** (commit records); undo **T8**.

– APPLY **Dirty page table: P3 (RecLSN 120), P7 (RecLSN 95). Where does redo start? Is the update with LSN 130 on P7 redone if P7's PageLSN on disk is 135?** 3 marks

Redo starts at **LSN 95**. The update is **skipped**: PageLSN 135 ≥ 130, so it is already on the page.

– ANALYZE **Why do large DBMSs prefer ARIES to shadow paging?** 3 marks

Shadow paging scatters pages, copies large page tables at each commit and handles concurrency poorly. ARIES keeps pages in place, supports steal and no-force buffering and fine-grained locking.

BEYOND THE SYLLABUS

This is not for the exam. Shadow paging lives on as copy-on-write, in the LMDB storage engine and in file systems such as ZFS and Btrfs. ARIES ideas run in IBM Db2, SQL Server and many others. Search next: "ARIES recovery algorithm" and "copy-on-write B-tree".

Key terms of this lesson

TERM	MEANING OR FORMULA	CHAPTER
Isolation levels	Read uncommitted, read committed, repeatable read, serializable; anomalies: dirty read, non-repeatable read, phantom	5.7
Intention locks	IS, IX, SIX on ancestors; lock root to leaf, unlock leaf to root	5.7
Update log record	<T_i, X, old value, new value>	5.8
Recovery rule	Start and commit: redo (forward). Start, no commit or abort: undo (backward)	5.8
WAL	Log records reach stable storage before the data block, and before the commit record	5.8
Checkpoint	<checkpoint L>: only transactions in L or started later need redo or undo	5.9
ARIES	LSN, PageLSN, dirty page table, CLRs; passes: analysis, redo (repeat history), undo	5.9

Exam question bank

Long-answer exam questions on these chapters, each with an outline of the points an examiner looks for. Answer one in full on paper before you read its outline.

Remember and understand

- REMEMBER **Q3. Explain the four SQL isolation levels and the anomalies each permits. Explain multiple granularity locking with intention lock modes and their compatibility matrix.** 13 marks

  Dirty read, non-repeatable read, phantom with examples; the level table; SET TRANSACTION syntax; how locks implement each level. Granularity tree (Figure 5.7.1); IS, IX, SIX; 5 × 5 matrix; locking rules (root first, parent modes, 2PL, bottom-up release); lock escalation.

- UNDERSTAND **Q4. Classify failures and storage types. Explain log-based recovery with deferred and immediate database modification, and state the write-ahead logging rules.** 13 marks

  Transaction failure, system crash, disk failure; volatile, non-volatile, stable storage. Log record types; undo backward, redo forward, idempotence; deferred (redo only, new values) versus immediate (undo and redo, old and new values); undo, redo and undo/redo logging rules with steal/force; three WAL rules (Figure 5.8.1); group commit.

Apply

- APPLY **Q8. Initially A = 100, B = 200, C = 300, D = 400. Log: <T1 start>, <T1, A, 100, 150>, <T2 start>, <T1 commit>, <T2, B, 200, 250>, <checkpoint {T2}>, <T3 start>, <T3, C, 300, 350>, <T2, A, 150, 175>, <T3 commit>, <T4 start>, <T4, D, 400, 450>, crash. Give the redo and undo lists and the final values.** 13 marks

  T1 committed before the checkpoint: nothing (Chapter 5.9). Redo T3: C = 350. Undo T4 and T2, backward: D = 400, A = 150, B = 200 (T2's first update lies before the checkpoint, so the scan goes back past it). Write redo-only records and <T4 abort>, <T2 abort>. Final **A = 150, B = 200, C = 350, D = 400** (Chapters 5.8 and 5.9).

Analyze and evaluate

– ANALYZE **Q11. Explain ARIES: its data structures, the three recovery passes and why it repeats history. Given a dirty page table P2 (RecLSN 210), P5 (RecLSN 180), where does redo start, and is an update with LSN 200 on P2 redone?** 13 marks

LSN, PageLSN, transaction table, dirty page table, CLRs, fuzzy checkpoints; analysis, redo, undo (Chapter 5.9). Repeating history brings every page to its exact pre-crash state so that logical undo and fine-grained locks work. Redo starts at LSN **180**; LSN 200 on P2 is **skipped**, since 200 < RecLSN 210 means the change was already on disk.

– EVALUATE **Q12. Compare shadow paging with log-based recovery. Why is shadow paging rare in large multi-user DBMSs?** 8 marks

Figure 5.9.2; commit by atomic pointer switch; no undo or redo. Against: page-table copying, fragmentation, garbage collection, poor concurrency. Log-based recovery with WAL keeps pages in place and allows steal and no-force (Chapters 5.8 and 5.9).

– EVALUATE **Q13. Design the transaction handling for an IRCTC-style Tatkal booking system where lakhs of users book seats in the first minutes. Choose and justify an isolation level, lock granularity, a deadlock strategy and a recovery approach.** 16 marks

Seat allocation needs SERIALIZABLE (or row locks with SELECT ... FOR UPDATE) so no seat is sold twice; searches can run at READ COMMITTED. Row-level X locks with IX on the table (Chapter 5.7), not table locks. Lock seats in a fixed order (by coach and berth) to prevent deadlock, with wait-for-graph detection as backup (5.6). Strict 2PL to avoid cascading rollback (5.4). WAL with group commit for durable, fast commits; checkpoints to keep restart short (5.8, 5.9). Payment confirmation only after commit (5.1).

Glossary (terms used in this lesson)

ARIES	A recovery algorithm using LSNs, a dirty page table and compensation log records, with analysis, redo and undo passes.
Checkpoint	A log record <checkpoint L> written after flushing log and data, so recovery can start near it.
Conflict	Two operations of different transactions on the same item, at least one of them a write.
Deadlock	A set of transactions each waiting for a lock held by another in the set.

Dirty read	Reading a value written by a transaction that has not committed.
Intention lock	IS, IX or SIX lock on an ancestor node, announcing finer locks below it.
Isolation level	How much a transaction is protected from others: read uncommitted, read committed, repeatable read or serializable.
Log	An append-only record of transaction starts, updates (old and new values), commits and aborts on stable storage.
Phantom	A row that appears when a query with the same condition is run again, inserted meanwhile by another transaction.
Shadow paging	Recovery by keeping the old page table untouched and switching a single pointer to the new one at commit.
Stable storage	Storage assumed never to lose data, approximated by copies on independent devices.

Sources and further reading

Textbooks

- Abraham Silberschatz, Henry F. Korth and S. Sudarshan, *Database System Concepts*, 7th ed., McGraw-Hill: the chapters "Transactions", "Concurrency Control" and "Recovery System" (states, schedules, locking, timestamps, validation, multiple granularity, deadlocks, log-based recovery, ARIES).
- Ramez Elmasri and Shamkant B. Navathe, *Fundamentals of Database Systems*, 7th ed., Pearson: the chapters on transaction processing concepts and theory, concurrency control techniques, and database recovery techniques (including shadow paging).
- Raghu Ramakrishnan and Johannes Gehrke, *Database Management Systems*, 3rd ed., McGraw-Hill: overview of transaction management, concurrency control and crash recovery (ARIES).
- Hector Garcia-Molina, Jeffrey D. Ullman and Jennifer Widom, *Database Systems: The Complete Book*, 2nd ed., Pearson: the chapters on transactions, concurrency control and coping with system failures.
- Jim Gray and Andreas Reuter, *Transaction Processing: Concepts and Techniques*, Morgan Kaufmann, 1993.

Original papers

- Theo Härder and Andreas Reuter, "Principles of Transaction-Oriented Database Recovery", *ACM Computing Surveys*, 1983 (the ACID acronym).

- K. P. Eswaran, J. N. Gray, R. A. Lorie and I. L. Traiger, "The Notions of Consistency and Predicate Locks in a Database System", *Communications of the ACM*, 1976 (two-phase locking).
- H. T. Kung and John T. Robinson, "On Optimistic Methods for Concurrency Control", *ACM Transactions on Database Systems*, 1981.
- C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh and Peter Schwarz, "ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging", *ACM Transactions on Database Systems*, 1992.
- Hal Berenson and others, "A Critique of ANSI SQL Isolation Levels", *ACM SIGMOD*, 1995.

Web resources

The writing session's network could not open web pages, so none of these was opened; product defaults and settings named in the book (isolation-level defaults, `deadlock_timeout`, MySQL error 1213) are written from memory and should be checked against current documentation.

- Slides and resources for Silberschatz, Korth and Sudarshan: db-book.com.
- NPTEL, "Data Base Management System": nptel.ac.in/courses/106105175.
- PostgreSQL documentation, "Transaction Isolation": postgresql.org/docs/current/transaction-iso.html.
- SQLite documentation, "Write-Ahead Logging": sqlite.org/wal.html.

Syllabus: the University free n easy universal syllabus, built from top universities' syllabi.

Built for the University free n easy YouTube channel from the universal syllabus of Database Management Systems (Oct 2026). Written with top universities' syllabi and standard textbooks; check a formula against your prescribed book before an exam, and report mistakes on the channel.