

UNIT 9

Transaction and Concurrency Control

(4 Hours) — Database Management System (CACS255) · BCA II Year / IV Semester

⚠ Important Note — Please Read

These notes were prepared quickly from the course syllabus and standard textbook knowledge. **They may contain errors, typing mistakes or incomplete points.** Please take care: cross-check definitions, formulas, SQL syntax and examples with your class notes, the prescribed textbook and your teacher before relying on them for exams or practicals.

9.1 Transaction Concept

A **transaction** is a **logical unit of work** — a collection of operations that performs a single task and accesses or changes the database. It starts with a BEGIN and ends with a COMMIT (success) or ROLLBACK/ABORT (failure). Either **all** its operations happen or **none** of them.

ACID Properties

Property	Meaning	Ensured by
Atomicity	All operations of the transaction are done, or none are (“all or nothing”).	Recovery manager (undo using log)
Consistency	A transaction takes the database from one consistent state to another consistent state.	Programmer + integrity constraints
Isolation	Concurrent transactions do not interfere; each seems to run alone.	Concurrency-control manager
Durability	After commit, the changes survive any later system failure.	Recovery manager (redo using log)

9.2 Simple Transaction Model

The database is accessed through two basic operations: **read(X)** transfers data item X from the database to a local buffer of the transaction, and **write(X)** transfers X from the buffer back to the database.

Example 9.1 — Transfer of Rs. 500 from account A to account B

T1: read(A); A := A - 500; write(A); read(B); B := B + 500; write(B); commit

Initially A = 5000 and B = 3000, so A + B = 8000.

Why the ACID properties matter

Atomicity: if the system fails after write(A) but before write(B), Rs. 500 disappears (A = 4500, B = 3000). Atomicity forces the partial change to be undone.

Consistency: A + B must still be 8000 after the transaction.

Isolation: if another transaction reads A and B between the two writes it sees 7500 — an inconsistent total.

Durability: once “committed” is shown to the user, A = 4500 and B = 3500 must persist even if the power fails.

Atomicity and Durability — Transaction States

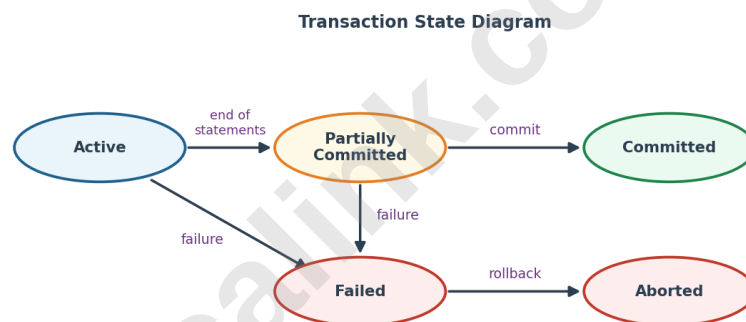


Figure 9.1 — State diagram of a transaction

State	Description
Active	Initial state; the transaction is executing.
Partially committed	After the last statement has executed (results may still be only in memory).
Failed	Normal execution can no longer proceed (error, crash, logical problem).
Aborted	Transaction rolled back and database restored to the state before it began. Then it may be restarted or killed.
Committed	Successfully completed; changes are permanent.

Recovery techniques for atomicity and durability

- **Shadow-copy scheme:** updates are made on a copy of the database; a *db-pointer* is switched to the new copy only on commit. Simple, but inefficient for large databases.

- **Log-based recovery (write-ahead logging):** every update is first recorded in a **log** (old value + new value) on stable storage. After a crash the system **undoes** uncommitted transactions and **redoes** committed ones. Checkpoints reduce the log to be scanned.

9.3 Transaction Isolation and Serializability

When many transactions run at once, their operations are **interleaved**. This improves throughput and reduces waiting, but without control it causes problems:

Problem	Description
Lost update	Two transactions read the same value and both write; one update overwrites the other.
Dirty read	A transaction reads data written by another transaction that has not yet committed (and may roll back).
Non-repeatable read	Reading the same row twice in a transaction gives different values because another transaction updated it in between.
Phantom read	Running the same query twice gives a different set of rows because another transaction inserted/deleted rows.

Schedules and Serializability

A **schedule** is the chronological order in which instructions of concurrent transactions are executed. A **serial schedule** runs transactions one after another; it is always correct. A schedule is **serializable** if its effect is equivalent to some serial schedule.

Conflict serializability. Two instructions of different transactions **conflict** if they access the **same data item** and **at least one is a write** (read–write, write–read, write–write; read–read never conflicts). If a schedule can be turned into a serial schedule by swapping *non-conflicting* adjacent instructions, it is conflict serializable.

Test using a precedence graph: draw a node for each transaction and an edge $T_i \rightarrow T_j$ if an operation of T_i conflicts with a *later* operation of T_j . **The schedule is conflict serializable if and only if the graph has no cycle.** A topological order of the graph gives an equivalent serial order.

Example 9.2 — Test serializability

S1: R1(A), W1(A), R2(A), W2(A) S2: R1(A), W2(A), W1(A)

Solution

S1: W1(A) before R2(A) and W2(A) gives edge $T1 \rightarrow T2$ only. No cycle $\rightarrow$ **conflict serializable** (equivalent to T1, T2).

S2: R1(A) before W2(A) gives $T1 \rightarrow T2$; W2(A) before W1(A) gives $T2 \rightarrow T1$. Cycle $\rightarrow$ **not conflict serializable**.

*Note: **View serializability** is a weaker condition (same initial reads, same read-from relations, same final writes); every conflict-serializable schedule is view serializable, but not the reverse.*

Transaction Isolation and Atomicity — Recoverability

- **Recoverable schedule:** if T_j reads data written by T_i , then T_i must commit *before* T_j commits. Otherwise, if T_i aborts, the committed T_j would have used data that never existed (atomicity lost).
- **Cascading rollback:** failure of one transaction forces rollback of others that read its dirty data.
- **Cascadeless schedule:** a transaction may read a value only *after* the writer has committed — this avoids cascading rollback.

Transaction Isolation Levels

The SQL standard defines four levels; a higher level gives more correctness but less concurrency.

Isolation level	Dirty read	Non-repeatable read	Phantom
READ UNCOMMITTED	Possible	Possible	Possible
READ COMMITTED	Prevented	Possible	Possible
REPEATABLE READ	Prevented	Prevented	Possible*
SERIALIZABLE	Prevented	Prevented	Prevented

Setting an isolation level

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;    -- T-SQL and MySQL
BEGIN TRANSACTION;
    SELECT Balance FROM Account WHERE AccNo = 1;
COMMIT;
```

*Note: *Per the SQL standard. Some systems (e.g. MySQL InnoDB) also prevent phantoms at REPEATABLE READ. Default level: SQL Server and Oracle — READ COMMITTED; MySQL InnoDB — REPEATABLE READ.*

9.4 Lock-Based Protocols

A **lock** controls concurrent access to a data item. A transaction must obtain the proper lock before reading or writing an item.

- **Shared lock (S):** for reading; many transactions can hold S on the same item.
- **Exclusive lock (X):** for reading and writing; only one transaction can hold it and no S locks may coexist.

Requested \ Held	S	X
S	✓ Compatible	✗ Wait
X	✗ Wait	✗ Wait

Two-Phase Locking Protocol (2PL)

Locking alone does not guarantee serializability. **2PL** requires each transaction to work in two phases:

- **Growing phase:** the transaction may acquire locks but cannot release any.
- **Shrinking phase:** the transaction may release locks but cannot acquire any new ones. The first unlock is the **lock point**.

2PL ensures **conflict serializability**, but it does **not** prevent deadlock or cascading rollback. **Strict 2PL** holds all exclusive locks until commit/abort (prevents cascading rollback); **rigorous 2PL** holds all locks until commit/abort.

Deadlock Handling

A **deadlock** occurs when each transaction in a set waits for a lock held by another in the set, so none can proceed.

Example 9.3 — Deadlock

T1 holds X-lock(A) and requests X-lock(B); T2 holds X-lock(B) and requests X-lock(A). Both wait forever.

Approach	Method
Prevention — Wait-Die (non-preemptive)	Older transaction requesting a lock held by a younger one waits ; a younger one requesting a lock held by an older one dies (is rolled back).

Approach	Method
Prevention — Wound-Wait (preemptive)	Older transaction requesting a lock held by a younger one wounds it (younger is rolled back); a younger one requesting a lock held by an older one waits .
Timeout	A transaction waiting longer than a set time is rolled back. Simple, but may abort needlessly.
Detection	Build a wait-for graph (edge $T_i \rightarrow T_j$ if T_i waits for T_j). A cycle means deadlock.
Recovery	Choose a victim (least cost), roll it back (fully or partially) and avoid starvation by not always picking the same victim.

9.5 Multiple Granularity

Locking can be done at different sizes (**granularity**): **database** → **file (table)** → **page** → **record**. Coarse granularity gives low locking overhead but less concurrency; fine granularity gives more concurrency but more overhead. The granularities form a tree. To lock a node correctly, the system uses **intention locks** on its ancestors:

Lock	Meaning
IS (Intention-Shared)	Some descendant node will be locked in shared mode.
IX (Intention-Exclusive)	Some descendant node will be locked in exclusive mode.
SIX (Shared + Intention-Exclusive)	The node is locked in S mode, and some descendant will be locked in X mode.

Compatibility matrix (✓ = can be held together):

	IS	IX	S	SIX	X
IS	✓	✓	✓	✓	✗
IX	✓	✓	✗	✗	✗
S	✓	✗	✓	✗	✗
SIX	✓	✗	✗	✗	✗
X	✗	✗	✗	✗	✗

- Locks are acquired from the **root down** to the node; released from the **node up** to the root.
- To lock a node in S or IS, the transaction must hold IS or IX on its parent; to lock in X, IX or SIX, it must hold IX or SIX on the parent.

9.6 Timestamp-Based Protocol

Instead of locks, each transaction T_i gets a unique **timestamp** $TS(T_i)$ when it enters the system; an older transaction has a smaller timestamp. The protocol makes the schedule equivalent to the serial order of timestamps. For each data item Q it keeps:

- **W-timestamp(Q)**: largest timestamp of any transaction that executed $write(Q)$ successfully;
- **R-timestamp(Q)**: largest timestamp of any transaction that executed $read(Q)$ successfully.

Operation by T_i	Condition	Action
$read(Q)$	$TS(T_i) < W\text{-timestamp}(Q)$	T_i would read an overwritten value → reject; roll back T_i.
$read(Q)$	$TS(T_i) \geq W\text{-timestamp}(Q)$	Execute ; $R\text{-timestamp}(Q) = \max(R\text{-timestamp}(Q), TS(T_i))$.
$write(Q)$	$TS(T_i) < R\text{-timestamp}(Q)$	A later transaction already read Q → reject; roll back T_i.
$write(Q)$	$TS(T_i) < W\text{-timestamp}(Q)$	T_i tries to write an obsolete value → reject; roll back T_i.
$write(Q)$	Otherwise	Execute ; $W\text{-timestamp}(Q) = TS(T_i)$.

- Rolled-back transactions get a **new timestamp** when restarted.
- **Thomas' write rule**: for a write with $TS(T_i) < W\text{-timestamp}(Q)$ (but $\geq R\text{-timestamp}(Q)$), simply *ignore* the obsolete write instead of rolling back T_i . This allows some schedules that are view (but not conflict) serializable.
- The protocol is **deadlock-free** (no waiting), but starvation is possible and cascading rollbacks may occur.

Point	Lock-based (2PL)	Timestamp-based
Order decided by	Order in which locks are obtained	Fixed in advance by timestamps
Waiting	Transactions wait for locks	No waiting; conflicting transactions are rolled back
Deadlock	Possible	Not possible
Overhead	Lock table, deadlock handling	Timestamps stored for each data item

Key Takeaways — Unit 9

- ✓ A transaction is a logical unit of work with the ACID properties: atomicity, consistency, isolation, durability.

- ✓ States: active → partially committed → committed; failure leads to failed → aborted.
- ✓ Concurrency causes lost update, dirty read, non-repeatable read and phantoms; isolation levels control them.
- ✓ A schedule is conflict serializable iff its precedence graph has no cycle.
- ✓ Two-phase locking (growing then shrinking) guarantees serializability but can deadlock; handle with wait-die/wound-wait, timeouts, or wait-for-graph detection.
- ✓ Multiple granularity uses intention locks (IS, IX, SIX) on a database–file–page–record hierarchy.
- ✓ Timestamp ordering avoids locks and deadlock by comparing TS(T) with R- and W-timestamps.

— *End of Unit 9* —

Reminder: these notes may contain errors — verify with your textbook and teacher.

bcalink.com