CUBRID Lock Manager

Multi-Granularity Locking, Conversion, Deadlock Detection

2026-05 · Code Analysis Seminar

Agenda

0. **The problem** — what a lock manager is for
1. **Overview** — one diagram of the whole flow
2. **Theory** — 2PL phases, strict 2PL, cost & remedies, MGL, `lub`
3. **Common patterns** — what every DBMS lock manager looks like
4. **CUBRID data structures** — OID keys, `LK_ENTRY`, dual-view threading
5. **Modes, matrix, acquisition, release, escalation**
6. **Deadlock** — the waits-for graph

> Closing: **Beyond CUBRID** — comparative designs.
> Appendix A: Lock vs Latch, in depth.

The problem — what a lock manager is for

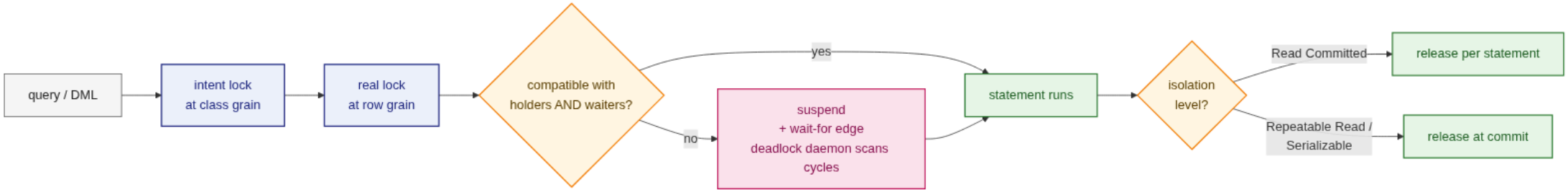
ACID's **I (Isolation)**: concurrent transactions must not see each other's in-progress work. Without coordination, classical anomalies appear:

Anomaly	Scenario	What goes wrong
Dirty read	T_1 reads T_2's uncommitted write	T_2 rolls back → T_1 saw a value that "never existed"
Lost update	T_1 and T_2 both read `n`, both write `n+1`	one of the increments is silently lost
Non-repeatable read	T_1 reads R twice; T_2 updates R between them	second read disagrees with the first
Phantom	T_1 runs `WHERE color='red'` twice; T_2 inserts a red row	second read returns a new row
Write skew	T_1, T_2 both read R; each writes a different cell using R	both updates pass their pre-condition; together they violate it

- **Goal**: produce **serializable schedules** while allowing as much concurrency as possible.
- **Lock manager's role**: provide the primitive — a **lock** — that lets a transaction declare **"I am reading / writing this object,"** and arbitrate conflicts (grant, queue, escalate, abort as deadlock victim).
- **Isolation levels** (Read Committed, Repeatable Read, Serializable) are **tunable trade-offs**: which anomalies are tolerated in exchange for more concurrency.

CUBRID

Overview — how a lock request flows



- **Two grains.** A coarse **intent** lock at the class first, then a fine **real** lock on the row.
- **Compatibility is two-sided.** Check against currently-granted modes **and** against modes already in the wait queue (the starvation guard).
- **Release timing follows isolation level.** Read Committed releases per statement; stricter levels hold to commit.

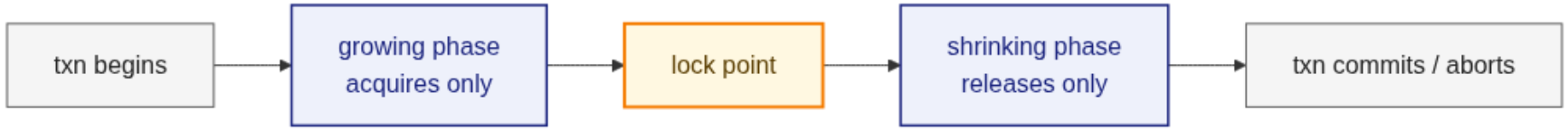
> The rest of the talk zooms into each box, names the data structures behind it, and shows the actual API surface in Part III.

Lock vs Latch

Dimension	Lock	Latch
Protects	logical txn serialization	physical data structure
Lifetime	transaction-long	very short
Stored	external lock table	inside the page / struct
CUBRID	`LK_ENTRY`	`PGBUF_LATCH`

> Conflating the two breaks every other design decision. — **Database Internals**, ch. 5.
> A deeper side-by-side and a heap-insert walk-through are in **Appendix A**.

2PL — two phases, one lock point



- **Growing phase**: a transaction may acquire any lock; **once one is released**, the shrinking phase has begun.
- **Shrinking phase**: only releases are allowed. No further acquires.
- The transition is the **lock point** — it does not have to be at commit time, but everything ordered around it determines the schedule.
- **Theorem (Eswaran 1976)**: if every transaction observes 2PL, the resulting schedule is **serializable**.

> Two-phase locking is a discipline on the **timing** of `lock(R)` / `unlock(R)` calls, not on which locks are held.

CUBRID

2PL variants — basic, strict, rigorous

Variant	When does the shrinking phase end?	Cascading abort?
Basic 2PL	as soon as locks are no longer needed	possible
Strict 2PL	**write** locks held until commit / abort	impossible
Rigorous 2PL	**every** lock held until commit / abort	impossible

- The textbook (Petrov ch. 5; **Database System Concepts**, ch. 14) treats strict and rigorous as practical refinements of basic 2PL.
- **Cascading abort**: if T_1 releases an X lock on R **before** committing and T_2 reads R, then T_1 aborts, T_2 must abort too. Strict 2PL forbids this by construction.
- **CUBRID uses strict 2PL** for write locks. Reads obey 2PL but their **duration** depends on isolation level — Read Committed releases per statement, RR/Serializable hold to commit. The release path (`lock_unlock_object_by_isolation`) is where this choice lives.

> "Strict" is the line WAL recovery assumes — undo at abort works only because no other transaction has seen our writes yet.

Cost of 2PL — and the four standard remedies

- **Guarantee**: serializable schedules (no anomaly).
- **Cost #1: contention**. Long shrinking phases serialize the workload.
- **Cost #2: deadlocks**. Hold-and-wait + circular dependency is unavoidable in general.
- **Four standard remedies**:
  - **Timeout** — cheap to implement, prone to false positives under load.
  - **Conservative 2PL** — acquire every lock up front, before doing any work. Avoids deadlock entirely, but caller must know its working set.
  - **Waits-for graph cycle scan** ← **CUBRID's primary detector** (`lock_detect_local_deadlock`).
  - **Timestamp avoidance** — **wait-die**, **wound-wait**. Decides who waits using transaction age; common in distributed designs (Spanner, CockroachDB), rare in monolithic DBMSs.

> CUBRID picks WFG detection as the primary mechanism and falls back to per-request timeout when WFG maintenance lags.

Three theoretical building blocks

Beyond plain S/X locks, three pieces of theory shape every real lock manager:

1. **Intention modes** (`IS`, `IX`, `SIX`) — multi-granularity locking. A coarse lock **declares intent** to take finer locks underneath, so two transactions can lock different rows of the same table without false table-level conflict.
2. **Compatibility matrix** — 2×2 (S, X) → 5×5 (with intention) → **12×12 in CUBRID** (add `BU`, `SCH-S/M`, `U`, `NON2PL`).
3. **Conversion** — re-requesting your own lock should **upgrade**, not deadlock with yourself. The new mode is the **least upper bound** (`lub`) of held and requested.
   Definition. In the lock-mode lattice, `lub(A, B)` is the smallest mode that **satisfies both** A and B. E.g. `lub(S, IX) = SIX`, `lub(IS, S) = S`, `lub(S, X) = X`.

> The data structures in Part II realize these three pieces as `LK_ENTRY` + the lock-mode matrix + a conversion table.

Why intention modes exist

Without intention modes (`IS`, `IX`, `SIX`) you face a binary choice for class-grain locking:

Choice	Concurrency story	Problem
Whole-class `S` / `X` only	Two writers can never coexist, even on **different rows**	dreadful concurrency on hot tables
No class lock at all	Writer on row 5 can't tell a DDL is mid-`ALTER TABLE` on the same class	breaks schema isolation

Intention modes solve both at once. Taking `IX` on a class **declares** "I will hold an `X` on at least one row underneath." Then:

- Two writers on **different rows** can both hold `IX` on the class — no conflict (`IX ∧ IX = ✓`).
- DDL takes `SCH-M`, which conflicts with `IX` — writers wait for the DDL to commit, but **fine-grained concurrency between writers is preserved**.
- A reader who needs `S` on the whole class still conflicts with `IX` — correct, because someone's writing a row.

> Without `IS` / `IX` / `SIX`, MGL collapses back to S/X-only. With them, you get **row-grain data concurrency + class-grain DDL protection** in the same compatibility matrix.

Seven common DBMS patterns

1. **Lock vs latch** separation
2. **Resource hashing** — OID, or `(relation, key range)`
3. **Aggregate-mode cache** — `total_holders_mode` (O(holders) → O(1))
4. **Dual-view threading** — resource view + transaction view
5. **MGL + intention modes** — IS, IX, S, SIX, X
6. **Conversion (lub) table** — `S + IX = SIX`
7. **FIFO queue + starvation guard** — holders ∧ waiters

> CUBRID is **one point** in this dial-space — not an invention.

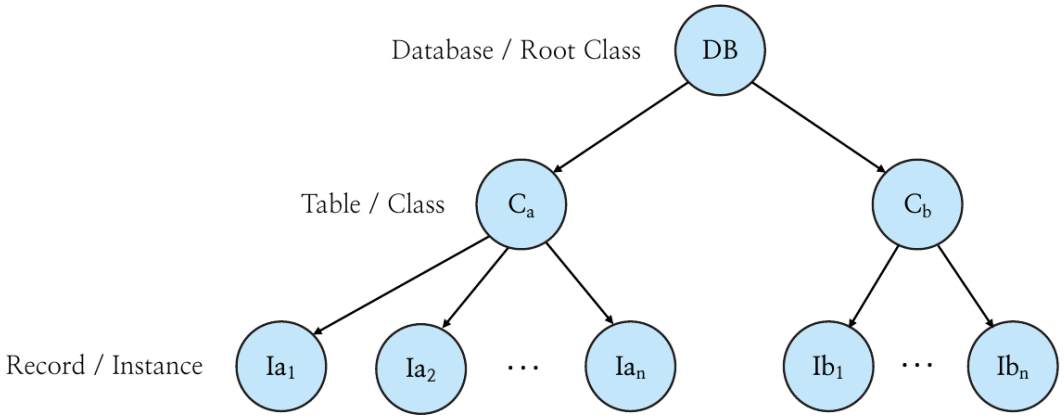
Part II

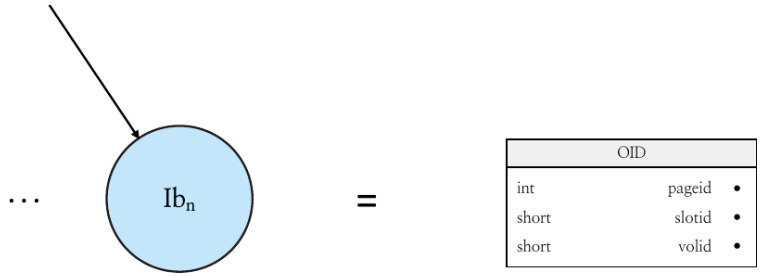
How does CUBRID turn the dials?

CUBRID

Naming everything with OID

CUBRID의 세분화 구조





데이터베이스, 테이블, 리소스 자원들은
모두 OID로 표현될 수 있습니다

```
typedef struct db_identifier DB_IDENTIFIER;
```

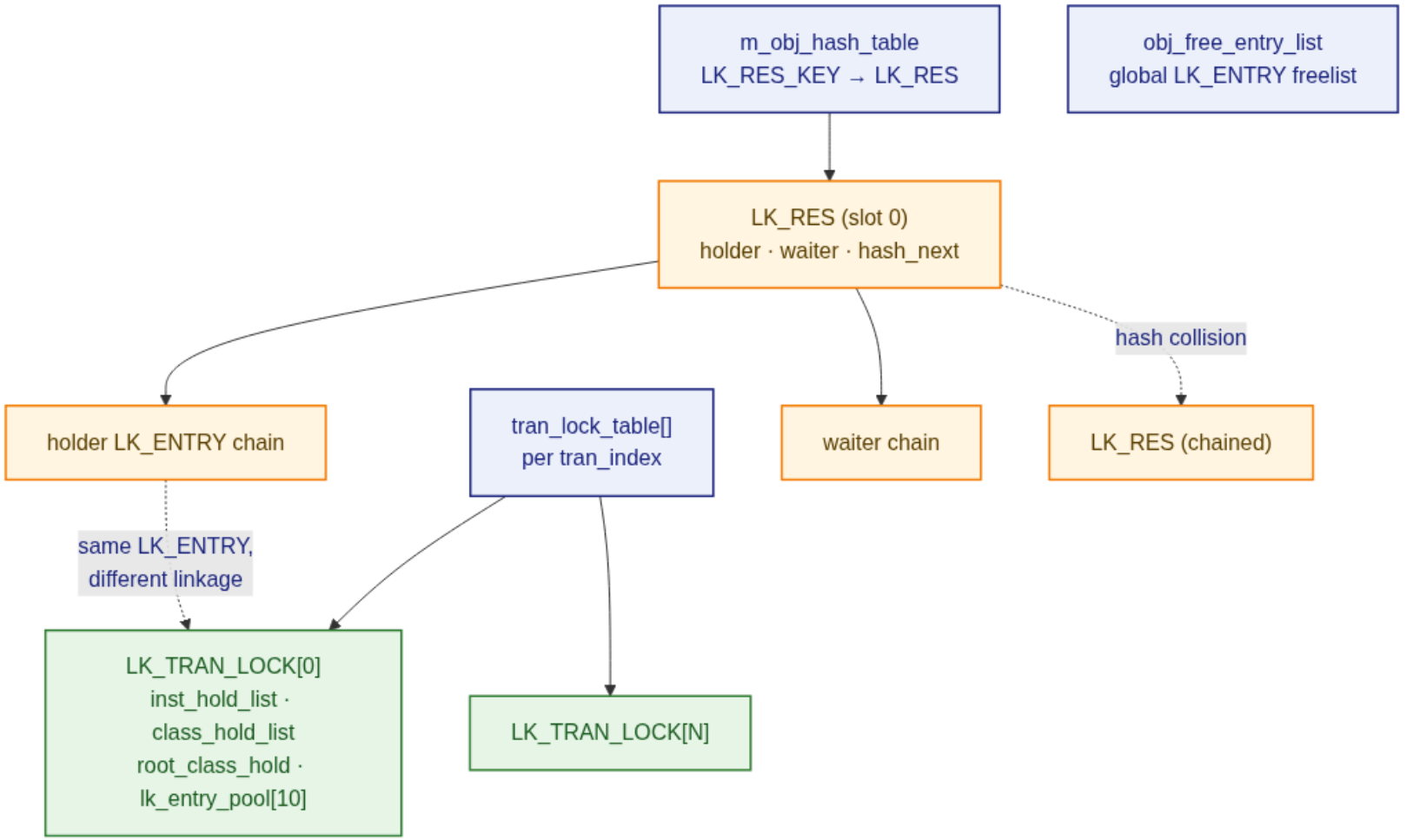

CUBRID

Three core types

```
// src/transaction/lock_manager.h
struct lk_res_key { LOCK_RESOURCE_TYPE type; OID oid; OID class_oid; };

struct lk_res {
  LK_RES_KEY key;
  LOCK       total_holders_mode;   // aggregate of granted modes
  LOCK       total_waiters_mode;   // aggregate of waiting modes
  LK_ENTRY  *holder, *waiter, *non2pl;
  pthread_mutex_t res_mutex;
};

struct lk_entry {
  LK_RES   *res_head;
  int       tran_index;
  LOCK      granted_mode, blocked_mode;
  int       count;                // re-entrant counter
  LK_ENTRY *next;                 // resource list (holder or waiter)
  LK_ENTRY *tran_next, *tran_prev; // transaction list
  LK_ENTRY *class_entry;          // parent class, one hop
  int       ngranules;            // children below this intention lock
};
```

> The aggregate fields turn each compatibility check from O(holders) into O(1).

The global lock table — `lk_global_data`



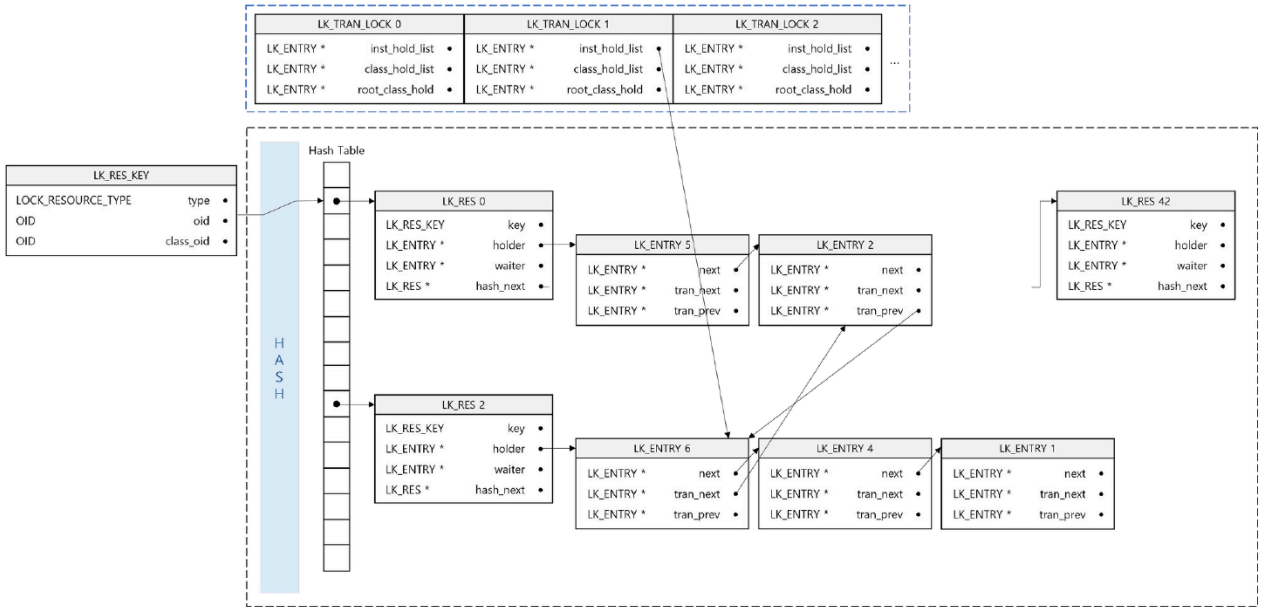
Three globals: hash table (`MAX_NTRANS × 300` slots), per-tran table with a 10-entry local pool, shared freelist for overflow.

Entry initialization — granted vs blocked

```
// lock_initialize_entry_as_granted — src/transaction/lock_manager.c
static void granted (LK_ENTRY *e, int tran, LK_RES *res, LOCK lock)
{
  e->tran_index   = tran;
  e->res_head     = res;
  e->granted_mode = lock;          // → this entry is a holder
  e->blocked_mode = NULL_LOCK;
  e->count        = 1;
  /* list pointers nulled */
}

// lock_initialize_entry_as_blocked — same file
static void blocked (LK_ENTRY *e, THREAD_ENTRY *th, int tran,
                     LK_RES *res, LOCK lock)
{
  e->thrd_entry   = th;
  e->granted_mode = NULL_LOCK;
  e->blocked_mode = lock;          // → this entry is a waiter
  /* ... */
}
```

Same struct, two constructors. For a **holder**, `granted_mode` is the live field; for a **waiter**, `blocked_mode`. A **conversion** in progress uses **both** — `granted_mode` is the current mode, `blocked_mode` is the upgrade target.

Dual-View Threading



- The same `LK_ENTRY` lives in **two lists at once**.
- Resource view: **who holds R?** → `next`. Txn view: **what does T hold?** → `tran_next/prev`.
- Commit walks the txn list. Compatibility check reads the resource-side aggregate.

A 12-mode vocabulary

Value	Mode	Meaning
0	`NA` / `NULL`	placeholder
1	`NON2PL`	tracker for locks released early under RC
3	`SCH-S` / `SCH-M`	schema stability / modification (DDL takes SCH-M)
4	`IS` / `IX`	intention shared / exclusive
5	`S`	shared
7	`BU`	bulk update (loaddb path)
8	`SIX`	shared + intention exclusive
9	`U`	shared, intends to upgrade to X
10	`X`	exclusive

> Gray's 5 modes (`IS / IX / S / SIX / X`) + CUBRID's 7 extras.

How SQL statements map to lock modes

A statement typically takes **two** locks: one **intent** at class grain (via `lock_scan`), one **real** at row grain (via `lock_object`).

SQL	Class grain (`lock_scan`)	Row grain (`lock_object`)
`SELECT` **(plain)**	`IS`	`S` (or none, depending on iso level)
`SELECT … FOR UPDATE`	`IX`	`U` ← "S that intends to upgrade"
`INSERT`	`IX`	`X` on the new row
`UPDATE`	`IX`	`X` on each updated row
`DELETE`	`IX`	`X` on each row
`LOAD DATA` / bulk load	`BU`	`X`
`CREATE / ALTER / DROP TABLE`	`SCH-M`	— (DDL is class-level only)
`SELECT` against a class mid-DDL	`SCH-S`	— ← waits behind `SCH-M`

- The `IX` row repeats deliberately — **every** row-level write takes the **same** class-level intent mode. The compatibility matrix does the rest.
- `U` is the upgrade-ready variant of `S`. It blocks other `U`s and `X`s but coexists with plain `S` readers — the right shape for a row you're **about to write**.

Compatibility matrix + starvation guard

	NULL	SCH-S	IS	S	IX	BU	SIX	X	SCH-M
SCH-S	✓	✓	✓	✓	✓	✓	✓	✓	✗
IS	✓	✓	✓	✓	✓	✗	✓	✗	✗
S	✓	✓	✓	✓	✗	✗	✗	✗	✗
IX	✓	✓	✓	✗	✓	✗	✗	✗	✗
BU	✓	✓	✗	✗	✗	✓	✗	✗	✗
SIX	✓	✓	✓	✗	✗	✗	✗	✗	✗
X	✓	✓	✗	✗	✗	✗	✗	✗	✗
SCH-M	✓	✗	✗	✗	✗	✗	✗	✗	✗

A new request must be compatible with **both** `total_holders_mode` and `total_waiters_mode` — the latter is the **starvation guard** that stops a stream of `S` from leapfrogging a waiting `X`.

CUBRID

Worked example — the starvation guard at work

Three transactions touch row R, in this order:

`t`	Action	Holders	Waiters	`total_holders`	`total_waiters`	Outcome
1	T1: `S` request	T1(S)	–	S	NULL	**granted**
2	T2: `X` request	T1(S)	T2(X)	S	X	**wait** (X ∧ S = ✗)
3	T3: `S` request	T1(S)	T2(X)	S	X	**wait** — compatible with holders, but `S ∧ X = ✗` against waiters

- At `t = 3`, a "holders-only" test would have granted T3 (`S ∧ S = ✓`).
- The two-sided test queues T3 behind T2 — exactly what fairness demands.

> Without the waiters check, a steady stream of `S` requests would starve every waiting `X` indefinitely. PostgreSQL has the same rule under the name **strong-lock fairness**.

Lock conversion — the `lub` rule

- **`lub` = least upper bound**. In the lock-mode lattice ordered by strength, `lub(A, B)` is the **smallest** mode that satisfies **both** A and B simultaneously.
- Re-requesting your own lock should **upgrade**, not deadlock with yourself.
- Rule: `granted_mode ← lub(granted_mode, requested_mode)`
  - `lub(S, IX) = SIX` — need both shared **and** intention-to-write
  - `lub(IS, S) = S` — `S` already implies `IS`
  - `lub(S, X) = X` — `X` already implies `S`
- Implementation: a **square table** indexed by `(held, requested)` — `src/transaction/lock_table.c`.
- A holder mid-conversion sets both `granted_mode` (current) and `blocked_mode` (target) until the upgrade succeeds.

Worked example A — self-upgrade with no conflict

T1 reads then writes the same row `r` — a common DML shape:

```
BEGIN;
SELECT * FROM accounts WHERE id = 5;        -- step 1
UPDATE accounts SET balance = 200 WHERE id = 5;  -- step 2
COMMIT;
```

Step	What T1 asks for	`granted_mode`	`blocked_mode`	What `LK_ENTRY` does
1	`S` lock on row 5	`S`	`NULL_LOCK`	new entry; placed in `r`'s holder list
2	requested mode `X` arrives	—	—	conversion table: `lub(S, X) = X`. No other holder ⇒ **in-place upgrade**
3	after upgrade	`X`	`NULL_LOCK`	**same** `LK_ENTRY`, mode field flipped

- The aggregate on `LK_RES` also flips: `total_holders_mode` goes `S` → `X`. The holder **count** stays at 1.
- Re-requesting your own lock is **not** an "ungrant + reacquire". The conversion table replaces that potentially deadlock-prone sequence with one atomic transition.

Why this matters. Without the conversion table, every upgrade would have to release the held lock first — opening a window where another transaction could slip in. T1 could then deadlock waiting for the lock it just released.

CUBRID

Worked example B — self-upgrade waiting on a conflict

Same code, but **another transaction T2 already holds `S` on row `r`** when T1 begins.

`t`	T1 (the upgrader)	T2 (other reader)	r's holders	T1's `LK_ENTRY`
1		acquires `S` on r	T2(S)	—
2	acquires `S` on r → **granted** (S ∧ S = ✓)		T1(S), T2(S)	`granted_mode = S`
3	requests `X` (UPDATE)	(still holding S, mid-txn)	T1(S), T2(S)	`granted_mode = S` · `blocked_mode = X` ← **pending**
4	**suspended** on r's waiter list	commits → releases S	T1(S)	unchanged
5	wakes; compat against `total_holders_mode = S` (own) passes		T1(X)	`granted_mode = X` · `blocked_mode = NULL_LOCK`

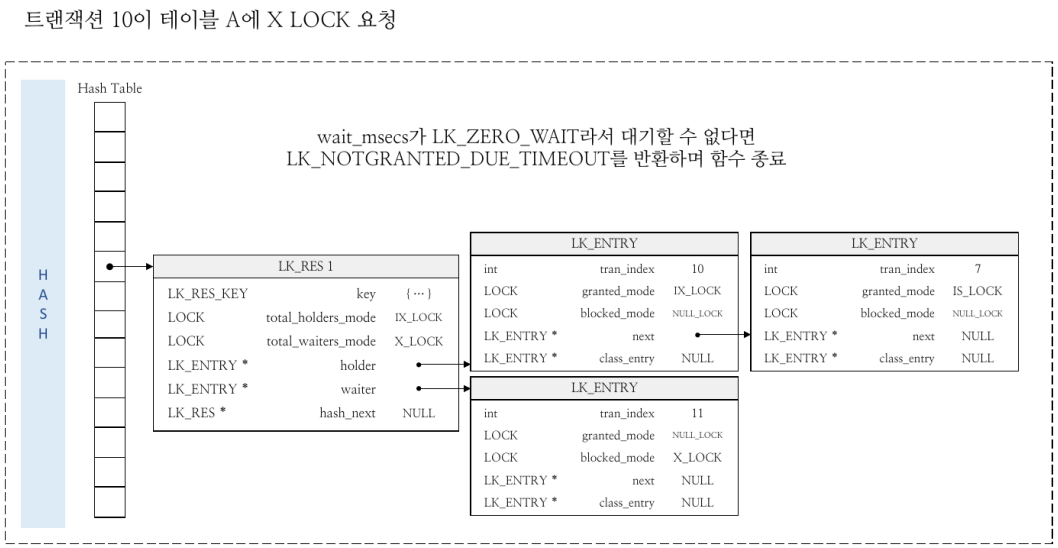
Different from Case A: at step 3, T1 is **simultaneously** a holder (`S`) and a waiter (for `X`). T1 never releases its `S` while waiting — that's why `LK_ENTRY` carries two mode fields instead of one.

Self-upgrade — Case A vs Case B at a glance

	(no conflict)	(conflicting holder)
Holder list before step 2	`T1(S)`	`T1(S), T2(S)`
At conversion request	`granted_mode = S → X` directly	`granted_mode = S`, `blocked_mode = X`
Does T1 release `S` while waiting?	n/a — never waits	**no**, holds S throughout
What unblocks the upgrade	nothing — immediate	T2's commit (or rollback)
`LK_ENTRY` count for T1	1 (always)	1 (always)
Mode fields used	`granted_mode` only	**both** `granted_mode` and `blocked_mode`

> Same `LK_ENTRY` shape, two different code paths. The conversion table covers both: every cell of the matrix has a `lub` defined, so there's always a target mode to upgrade **to**.

Acquisition flow



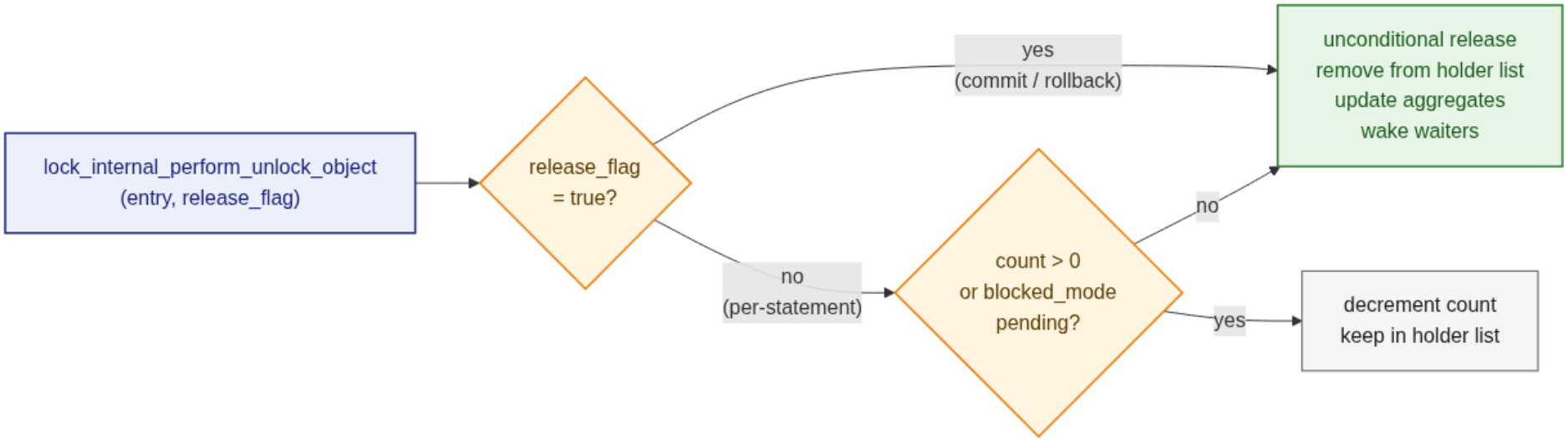
- **Two API surfaces.** `lock_scan(class_oid, mode)` — intention lock at class grain. `lock_object(record_oid, …)` — real lock at instance grain.
- Inside `lock_internal_perform_lock_object`: fresh resource → grant. Otherwise compatibility against `total_holders_mode ∧ total_waiters_mode` → grant, enqueue as waiter, or suspend.
- Wake reasons in `enum LOCK_WAIT_STATE`: `LOCK_RESUMED`, `_TIMEOUT`, `_ABORTED` (deadlock victim), `_INTERRUPT`.

Acquisition state machine — `lock_internal_perform_lock_object`

```
// src/transaction/lock_manager.c  (compressed; ... = elided)
static int
acquire (THREAD_ENTRY *th, int tran, const OID *oid,
         const OID *class_oid, LOCK lock, int wait_msecs)
{
  if (class_oid && !OID_IS_ROOTOID (class_oid))
    lock_escalate_if_needed (th, class_entry, tran);   // may subsume

  /* find or insert the resource */
  lk_Gl.m_obj_hash_table.find_or_insert (th, key, &res);

  LOCK agg = res->total_holders_mode | res->total_waiters_mode;

  if (!res->holder && !res->waiter)   /* grant_fresh */          ...
  else if (compatible (agg, lock))    /* grant_via_compat */     ...
  else                                /* enqueue + suspend */    ...
}
```

> Three outcomes; the compatibility test reads **both** aggregates (`holders | waiters`) — that's the starvation guard.

Release: isolation level decides



- **Read Committed** — instance locks released at statement end (`release_flag = false` path; RC's small footprint).
- **Repeatable Read / Serializable** — released only at commit / rollback (`release_flag = true` path; strict 2PL).
- **Class locks** — always held to end of transaction (intention locks must outlive their children).
- `SCH-M` — special-cased; conflicts with every mode except `NULL`. Acquisition is uniform across isolation levels — **isolation is encoded in the release path**.

Worked example — RC vs RR on the same timeline

Two sessions on row `r` (balance starts at `1000`). T2 reads twice **in one transaction**; T1 updates and commits between the reads. Question: what does T2 see at the second read?

`t`	T1 (writer)	T2 (reader, one txn)
1		`SELECT balance WHERE id=r` → reads `1000` · takes `S`
2	`UPDATE balance=1100 WHERE id=r` — requests `X`	
3	T1 **commits (if able to)**	
4		`SELECT balance WHERE id=r` — second read
5		T2 commits

The outcome depends entirely on **whether T2 still holds its `S`** when T1 reaches `t = 2`:

T2's isolation	T2's `S` released at	T1 at `t = 2`	T1 at `t = 3`	T2 reads at `t = 4`
Read Committed	end of statement (`t = 1`)	`X` **granted**	**commits**	**1100** ← T1's value is **already committed** · **non-repeatable read**
Repeatable Read	T2 commit (`t = 5`)	`X` **waits** for T2's `S`	still waiting	**1000** · **non-repeatable read forbidden**

Lock escalation

- A single transaction acquiring more than `lock_escalation_threshold` instance locks on one class:
  - drop the per-row locks → take one **class-grain** lock instead.
- A pressure-relief valve for memory.
- This is why `LK_ENTRY` carries two fields:
  - **`ngranules`** — children below this intention lock
  - **`class_entry`** — one-hop pointer to the parent class's entry
- Trade-off: less memory ↔️ unintended serialization.

> Implementation: `lock_escalate_if_needed` in `src/transaction/lock_manager.c`.

Worked example — escalation under bulk DML

A maintenance job runs:

```
UPDATE accounts SET status = 'archived' WHERE created_at < '2020-01-01';
-- imagine 50,000 rows match
```

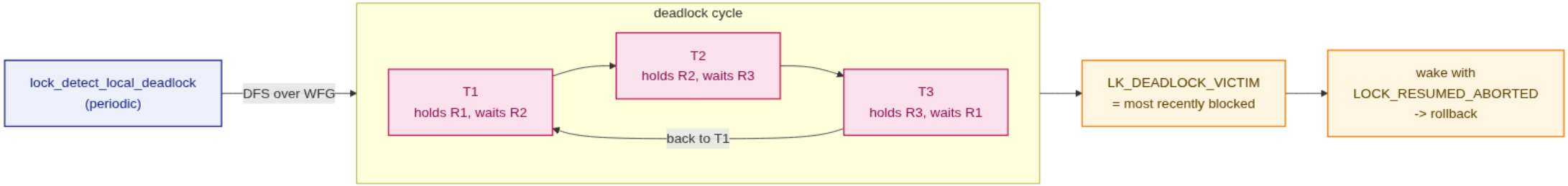
Step	What the lock manager does	`LK_ENTRY` count (this txn)
1	`lock_scan(accounts, IX)` → class-level intent	**1** (class IX)
2	`lock_object(row_1, X)`, `(row_2, X)`, …	1 + N row entries
3	At row `N = lock_escalation_threshold` (e.g. 10,000): `lock_escalate_if_needed` fires	escalation triggered
4	Drop **all** per-row `X` entries; convert class `IX` → class `X`	**1** (class X)
5	Remaining 40,000 rows are protected by the class `X` — no new per-row entries	stays at **1**

What the trade-off costs and buys:

- ✅ Memory drops from `O(rows touched)` to `O(1)`.
- ✅ Subsequent compatibility checks for this class are one matrix lookup, not 50,000 list scans.
- ❌ Other transactions wanting **any** row in `accounts` now block on the class `X` — concurrency collapses for the duration of the txn.

CUBRID

Deadlock — Waits-For Graph



- T1 waits on a lock held by T2 → add WFG edge T1 → T2 (`LK_WFG_EDGE`).
- `lock_detect_local_deadlock` walks the WFG (DFS); cycle → **victim = most-recently-blocked** → `LOCK_RESUMED_ABORTED` → rollback.
- "**Local**" — distributed deadlocks fall back to timeout.

Deadlock detector — `lock_detect_local_deadlock`

```
// src/transaction/lock_manager.c  (pseudocode — actual is longer)
int
lock_detect_local_deadlock (THREAD_ENTRY *thread_p)
{
  /* 1. snapshot active waiters into WFG nodes */
  for (i = 0; i < num_trans; i++)
    if (waiters[i].state == LOCK_SUSPENDED)
      add_wfg_node (i, waiters[i].wait_stime);

  /* 2. DFS over wait-for edges; back-edges = cycles */
  for (i = 0; i < num_nodes; i++)
    if (!visited[i])
      dfs (i, &cycle_found);

  /* 3. pick the most-recently-blocked txn in any cycle */
  if (cycle_found)
    {
      victim = pick_by_max (wfg.nodes, .thrd_wait_stime);
      victim->state = LOCK_RESUMED_ABORTED;   // → rollback in acquisition loop
    }
  return cycle_found ? LK_DEADLOCK_FOUND : NO_ERROR;
}
```

> Runs **periodically**, not on every wait. Victim selection is implicit in WFG insertion order — see analysis doc §Open Questions #1.

Worked example — two transactions, two rows

Setup: `accounts (id, balance)`, initial `A.balance = 1000`, `B.balance = 1000`. Two sessions start at the same time:

Txn	Statement	Intent
T1	`UPDATE accounts SET balance = balance - 100 WHERE id = 'A'`	X lock on A
T1	`UPDATE accounts SET balance = balance + 100 WHERE id = 'B'`	X lock on B
T2	`UPDATE accounts SET balance = balance -  50 WHERE id = 'B'`	X lock on B
T2	`UPDATE accounts SET balance = balance +  50 WHERE id = 'A'`	X lock on A

> T1 transfers from A to B; T2 transfers from B to A. The schedule depends on how the scheduler interleaves the two — and one of them is unlucky.

Worked example — interleaving creates the cycle

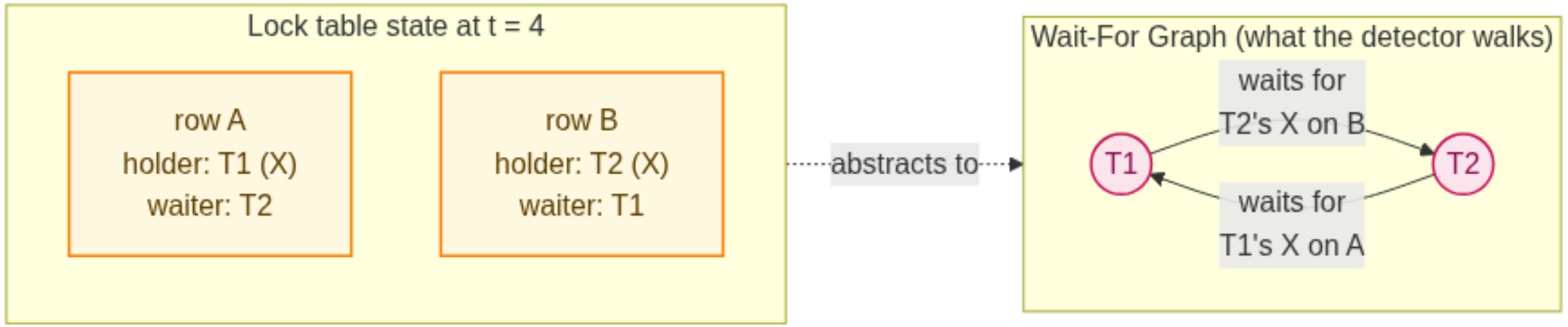
t	T1 acts	T2 acts	Lock table state
1	X-lock A → **granted**		A: holder = T1
2		X-lock B → **granted**	A: T1 ; B: T2
3	X-lock B → **wait**		A: T1 ; B: T2, waiter T1 · WFG: **T1 → T2**
4		X-lock A → **wait**	A: T1, waiter T2 ; B: T2, waiter T1 · WFG: **T1 → T2 → T1** (cycle)

- After `t = 4`, **both** transactions are in `LOCK_SUSPENDED`.
- Neither will progress without external intervention. The WFG has a closed cycle of length 2.

> Per-request timeout would eventually break this, but minutes of wasted wait — the detector resolves it in tens of milliseconds.

CUBRID

Worked example — the cycle, visualized



- **Left**: the concrete lock table — two `LK_RES` records, each with a holder and a waiter.
- **Right**: the abstract WFG — one node per active waiter, one directed edge per `"waiter waits for the holder"` relationship.
- **Detector never inspects rows.** It walks the graph on the right.

> The WFG is derived from the lock table: every `(waiter, holder)` pair on a resource becomes a directed edge `waiter → holder`. A cycle in that graph ≡ a deadlock.

Worked example — the detector resolves it

A tick of `lock_detect_local_deadlock` fires (it runs **periodically**, not on every wait):

1. **Snapshot.** Walk active waiters in `lk_Gl`. Both T1 and T2 are `LOCK_SUSPENDED`; add them to the WFG.
2. **DFS.** Start at T1 → follow edge to T2 → follow edge to T1. **Back-edge** to a node already on the DFS stack → cycle found.
3. **Victim.** Among nodes in the cycle, pick the one whose `thrd_wait_stime` is **latest** — the most-recently-blocked. Here that's **T2** (it blocked at `t = 4`).
4. **Wake.** Set T2's wait state to `LOCK_RESUMED_ABORTED`.
5. **Rollback.** T2's acquisition loop sees the abort flag → triggers transaction rollback → releases its X lock on B.
6. **T1 unblocks.** Compatibility test for T1's pending X on B now passes (no holder). T1 is granted, runs the rest of its statements, eventually commits.

> No lock is ever **taken away** from a running txn. The loser pays the rollback cost; the winner only notices a bit of extra wait.

CUBRID

Worked example — a three-transaction cycle

Three transactions, three rows. Each grabs one row and then reaches for the next — a circular `T1 → T2 → T3 → T1` chain.

t	T1	T2	T3	WFG state
1	X on R1 → **granted**			—
2		X on R2 → **granted**		—
3			X on R3 → **granted**	—
4	X on R2 → **wait**			T1 → T2
5		X on R3 → **wait**		T1 → T2 → T3
6			X on R1 → **wait**	T1 → T2 → T3 → **T1** (cycle)

- **DFS** from T1 walks T1→T2→T3, back-edge to T1 — cycle at depth 3.
- **Victim** = **T3** (latest `thrd_wait_stime`). T3 aborts → R3 freed → T2 wakes → R2 freed → T1 wakes.

CUBRID

Source walkthrough — where to read next

Symbol names are the stable handle. Line numbers drift with refactors; `git grep -n '<symbol>' src/transaction/` is your friend.

Topic	Symbol(s)	File
Core types	`lk_entry`, `lk_res_key`, `lk_res`	`lock_manager.h`
Wake reasons	`enum LOCK_WAIT_STATE`	`lock_manager.c`
Init / finalize	`lock_initialize`, `lock_finalize`	`lock_manager.c`
Entry constructors	`lock_initialize_entry_as_{granted,blocked,non2pl}`	`lock_manager.c`
Acquire	`lock_object` → `lock_internal_perform_lock_object`	`lock_manager.c`
Release	`lock_unlock_object` → `..._by_isolation`	`lock_manager.c`
Escalation	`lock_escalate_if_needed`	`lock_manager.c`
Deadlock	`lock_detect_local_deadlock`	`lock_manager.c`
Compatibility + conversion tables	(static 12 × 12 arrays)	`lock_table.c`
Waits-for graph abstraction	(header + impl)	`wait_for_graph.{h,c}`

> The analysis doc (`knowledge/code-analysis/cubrid/cubrid-lock-manager.md`) keeps a position-hint table with concrete line numbers as of its `updated:` date — refresh it whenever you traverse the code.

Beyond CUBRID — research frontiers

Direction	One line
Bamboo (2021)	Release X locks before commit. Generalizes CUBRID's `NON2PL`.
Brook-2PL (2025)	Static dependency pre-analysis → deadlock-free 2PL.
TXSQL (2025)	Adaptive lock-mode adjustment + contention-aware scheduling.
OCC (Hekaton / Silo / Cicada)	Replace the lock table with commit-time validation.
SSI (PostgreSQL)	Predicate locking — a cheaper path to serializability.
VLL	Partition the lock table itself when **it** is the bottleneck.

> CUBRID's dial position: **textbook 2PL + MGL + WFG detection**. A well-defended point.

Thank you

Q & A

- Analysis: `knowledge/code-analysis/cubrid/cubrid-lock-manager.md`
- Code: `src/transaction/lock_manager.{h,c}` · `lock_table.{h,c}` · `wait_for_graph.{h,c}`

Appendix

Appendix A — Lock vs Latch, side by side

Two words that sound alike. Same DBMS, completely different machines.

Dimension	Lock	Latch
What it protects	transactional serialization order **(logical)**	physical structure integrity **(mid-split, mid-rebalance)**
Held for	duration of the transaction (or per-statement under RC)	duration of a critical section (microseconds)
Stored	external lock table (`LK_RES` , `LK_ENTRY`)	inside the page / structure itself (`PGBUF_LATCH`)
Acquisition discipline	2PL (growing → shrinking)	latch coupling, fixed lock order, deadlock-free by design
Granularity	OID-shaped: database, class, instance	page, list, hash bucket
Conflict resolution	wait → WFG cycle scan → victim rollback	spin or sleep; never deadlocks if discipline is followed

> **Database Internals** ch. 5's first hard rule: don't conflate them.

Appendix A — Lock vs Latch, in code (CUBRID heap insert)

```
PGBUF_LATCH page (X)              ← physical: nobody else may mutate this page
allocate slot, write record       ← page is now correct in memory
lock_object on (page, slot) OID   ← logical: claim transactional visibility
UNLATCH page                      ← physical exclusion ends here

... continue with the rest of the txn (latch released, lock retained) ...

at commit: lock_unlock_object     ← logical lock finally released
```

- Page latch lives for **microseconds** — only while the bytes mutate.
- Row lock lives for **minutes to hours** — until the txn commits.
- The two timescales differ by orders of magnitude. Conflating them is a structural bug:
  - **holding a lock during a page-level critical section** serializes the entire workload through that page.
  - **holding a latch across a network round-trip** serializes the entire workload through that thread.
- B-tree split uses **latches only** — no transactional visibility implications.