CUBRID 잠금 관리자

다중 세분성 잠금, 변환, 교착 상태 감지

2026-05 · 코드 분석 세미나

목차

CUBRID

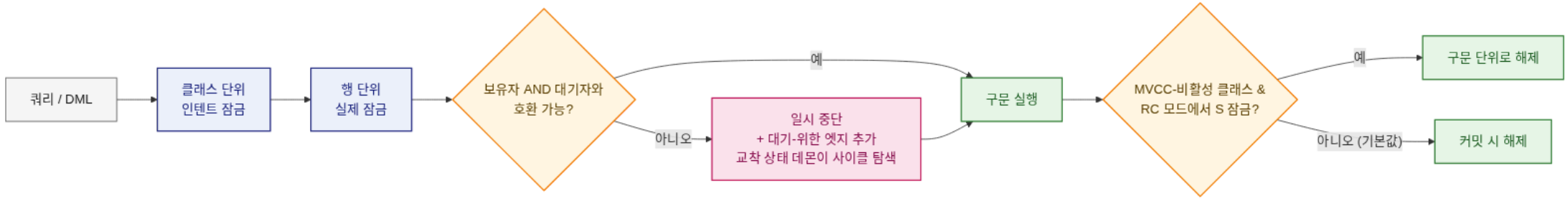
문제 — 잠금 관리자가 존재하는 이유

ACID의 **I(격리성)**: 동시 트랜잭션은 서로의 진행 중인 작업을 볼 수 없어야 한다. 조율 없이는 고전적 이상 현상이 나타난다:

이상 현상	시나리오	무엇이 잘못되는가
더티 읽기	T_1이 T_2의 미커밋 쓰기를 읽음	T_2가 롤백 → T_1은 "존재하지 않았던" 값을 읽은 셈
갱신 손실	T_1과 T_2가 `n`을 읽고 둘 다 `n+1`을 씀	증가분 하나가 조용히 사라짐
반복 불가 읽기	T_1이 R을 두 번 읽는 사이 T_2가 R을 수정	두 번째 읽기가 첫 번째와 다름
팬텀	T_1이 `WHERE color='red'`를 두 번 실행; T_2가 빨간 행 삽입	두 번째 실행에서 새 행이 등장
쓰기 왜곡	T_1, T_2가 R을 읽고 각자 다른 셀을 씀	각 갱신은 전제조건을 통과했지만, 합치면 위반

- **목표**: 가능한 한 많은 동시성을 허용하면서 **직렬화 가능한 스케줄**을 만드는 것.
- **잠금 관리자의 역할**: "나는 이 객체를 읽는/쓰는 중"이라고 선언할 수 있는 기본 단위인 **잠금**을 제공하고, 충돌을 중재한다(허가, 대기열, 에스컬레이션, 교착 상태 피해자로 중단).
- **격리 수준**(읽기 커밋, 반복 읽기, 직렬화 가능)은 **조정 가능한 트레이드오프**다: 더 많은 동시성을 얻는 대가로 어떤 이상 현상을 허용할 것인가.

개요 — 잠금 요청의 흐름



- **두 단위.** 먼저 클래스 단위의 인텐트 잠금을 획득하고, 이후 행 단위의 실제 잠금을 획득한다.
- **호환성은 양방향이다.** 현재 허가된 모드뿐 아니라 이미 대기열에 있는 모드도 함께 확인한다(기아 방지 장치).
- **해제 시점은 범위에 따라 다르다.** MVCC-비활성 클래스에서는 RC가 구문 단위로 해제하며, 그 외의 경우 격리 수준은 해제 시점에 영향을 주지 않는다. 자세한 내용은 §"해제 경로"에서 다룬다.

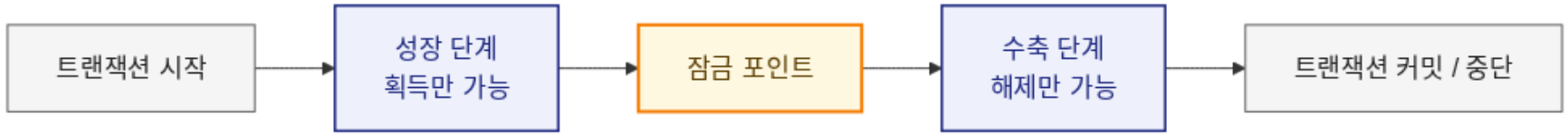
> 이후 슬라이드에서는 각 상자를 확대하여 그 뒤의 자료구조를 명명하고, 파트 III에서 실제 API 표면을 보여준다.

잠금(Lock) vs 래치(Latch)

관점	잠금(Lock)	래치(Latch)
보호 대상	논리적 트랜잭션 직렬화	물리적 자료구조
유지 시간	트랜잭션 전체	매우 짧음
저장 위치	외부 잠금 테이블	페이지 / 구조체 내부
CUBRID	`LK_ENTRY`	`PGBUF_LATCH`

> 이 둘을 혼동하면 이후의 모든 설계 결정이 어긋난다. — **Database Internals**, ch. 5.
> 더 깊은 비교와 힙 삽입 예제는 **부록 A**에서 다룬다.

2PL — 두 단계, 하나의 잠금 포인트



- **성장 단계**: 트랜잭션은 어떤 잠금이든 획득할 수 있다. **하나라도 해제하는 순간**, 수축 단계가 시작된다.
- **수축 단계**: 해제만 허용된다. 더 이상 획득은 불가능하다.
- 그 전환점이 **잠금 포인트**다. 반드시 커밋 시점일 필요는 없지만, 이를 중심으로 스케줄의 순서가 결정된다.
- **정리 (Eswaran 1976)**: 모든 트랜잭션이 2PL을 준수하면 결과 스케줄은 **직렬화 가능**하다.

> 2단계 잠금은 어떤 잠금을 보유하느냐가 아니라, `lock(R)` / `unlock(R)` 호출의 **타이밍**에 관한 규율이다.

CUBRID

2PL 변형 — 기본, 엄격, 강제

변형	수축 단계 종료 시점	연쇄 중단 가능성
기본 2PL	잠금이 더 이상 필요 없어지는 즉시	가능
엄격한 2PL	**쓰기** 잠금을 커밋/중단까지 보유	불가능
강제 2PL	**모든** 잠금을 커밋/중단까지 보유	불가능

- 교과서(Petrov ch. 5; **Database System Concepts**, ch. 14)는 엄격과 강제를 기본 2PL의 실용적 개선으로 다룬다.
- **연쇄 중단**: T_1이 R에 대한 X 잠금을 커밋 **전**에 해제하고 T_2가 R을 읽으면, T_1이 중단될 때 T_2도 중단해야 한다. 엄격한 2PL은 이 상황을 구조적으로 차단한다.
- **CUBRID는 X 잠금에 엄격한 2PL을 사용**한다. 모든 X 잠금은 커밋까지 유지된다. 읽기는 MVCC 방식이므로, 사용자 테이블에 대한 일반 SELECT는 행 잠금을 취득하지 않는다. RC-vs-RR S 잠금 해제 조절 옵션은 **MVCC-비활성** 클래스(루트, 시리얼, 콜레이션, HA apply-info)에만 `lock_unlock_object_by_isolation`을 통해 적용된다.

> "엄격"은 WAL 복구가 전제하는 기준선이다. 중단 시의 언두는 다른 트랜잭션이 아직 우리의 쓰기를 보지 못했을 때에만 작동한다.

2PL의 비용 — 네 가지 표준 대응책

- **보장**: 직렬화 가능한 스케줄(어떤 이상 현상도 없음).
- **비용 #1: 경합.** 긴 수축 단계는 작업 부하를 직렬화한다.
- **비용 #2: 교착 상태.** 점유-대기 + 순환 의존은 일반적으로 피할 수 없다.
- **네 가지 표준 대응책**:
  - **타임아웃** — 구현이 쉽지만, 부하 상황에서 오탐이 잦다.
  - **보수적 2PL** — 작업 시작 전 모든 잠금을 미리 획득한다. 교착 상태를 완전히 회피하지만, 호출자가 작업 집합을 미리 알아야 한다.
  - **대기-위한 그래프 사이클 탐색** ← **CUBRID의 주 감지 수단** (`lock_detect_local_deadlock`).
  - **타임스탬프 회피** — **대기-사망(wait-die)**, **상처-대기(wound-wait)**. 트랜잭션 나이를 기준으로 누가 기다릴지 결정한다. 분산 설계(Spanner, CockroachDB)에서 흔하며, 단일 DBMS에서는 드물다.

> CUBRID는 WFG 감지를 주 메커니즘으로 채택하고, WFG 유지가 뒤처질 때는 요청별 타임아웃으로 보완한다.

세 가지 이론적 기반

기본 S/X 잠금을 넘어, 세 가지 이론이 모든 실제 잠금 관리자의 형태를 결정한다:

1. **인텐션 모드** (`IS`, `IX`, `SIX`) — 다중 세분성 잠금. 굵은 잠금이 그 하위에 세밀한 잠금을 취득할 **의도**를 선언함으로써, 두 트랜잭션이 같은 테이블의 서로 다른 행을 잠글 때 테이블 수준에서 불필요한 충돌이 생기지 않는다.
2. **호환성 행렬** — 2×2(S, X) → 5×5(인텐션 포함) → **CUBRID에서 12×12**(BU, SCH-S/M, U, NON2PL 추가).
3. **변환** — 자신의 잠금을 재요청할 때는 **업그레이드**해야지, 자기 자신과 교착 상태에 빠져서는 안 된다. 새 모드는 보유 모드와 요청 모드의 **최소 상한(lub)**이다.
   정의. 잠금 모드 격자에서 `lub(A, B)`는 A와 B를 **동시에 만족하는** 가장 작은 모드다. 예: `lub(S, IX) = SIX`, `lub(IS, S) = S`, `lub(S, X) = X`.

> 파트 II의 자료구조는 이 세 요소를 `LK_ENTRY` + 잠금 모드 행렬 + 변환 테이블로 구현한다.

인텐션 모드가 필요한 이유

인텐션 모드(`IS` , `IX` , `SIX`) 없이는 클래스 단위 잠금에서 두 가지 중 하나를 선택해야 한다:

선택	동시성 특성	문제점
클래스 전체 `S` / `X` 만 사용	두 쓰기 트랜잭션은 **다른 행**에서도 동시에 존재할 수 없음	핫 테이블에서 동시성이 매우 나빠짐
클래스 잠금 없음	행 5에 쓰는 트랜잭션이 DDL이 같은 클래스를 `ALTER TABLE` 중임을 알 수 없음	스키마 격리가 깨짐

인텐션 모드는 두 문제를 동시에 해결한다. 클래스에 `IX` 를 취득하는 것은 "나는 하위의 최소 한 행에 `X` 를 보유할 것"이라고 선언하는 것이다. 그 결과:

- **다른 행**에 쓰는 두 트랜잭션은 클래스에 `IX` 를 동시에 보유할 수 있다 — 충돌 없음(`IX ∧ IX = ✓`).
- DDL은 `SCH-M` 을 취득하며 이는 `IX` 와 충돌한다. 쓰기 트랜잭션은 DDL 커밋까지 기다리지만, **쓰기 간 세밀한 동시성은 유지**된다.
- 클래스 전체에 `S` 가 필요한 읽기 트랜잭션은 `IX` 와 충돌한다 — 누군가가 행을 쓰고 있으므로 정확한 동작이다.

> `IS` / `IX` / `SIX` 없이는 MGL이 S/X 전용으로 무너진다. 이 모드가 있으면 동일한 호환성 행렬에서 **행 단위 데이터 동시성 + 클래스 단위 DDL 보호**를 함께 얻는다.

DBMS에서 공통으로 보이는 일곱 가지 패턴

1. **잠금과 래치** 분리
2. **리소스 해싱** — OID 또는 `(relation, key range)`
3. **집계 모드 캐시** — `total_holders_mode` (O(보유자) → O(1))
4. **이중 뷰 스레딩** — 리소스 뷰 + 트랜잭션 뷰
5. **MGL + 인텐션 모드** — IS, IX, S, SIX, X
6. **변환(lub) 테이블** — `S + IX = SIX`
7. **FIFO 대기열 + 기아 방지 장치** — 보유자 ∧ 대기자

> CUBRID는 이 선택지 공간의 **한 지점**이지, 새로운 발명이 아니다.

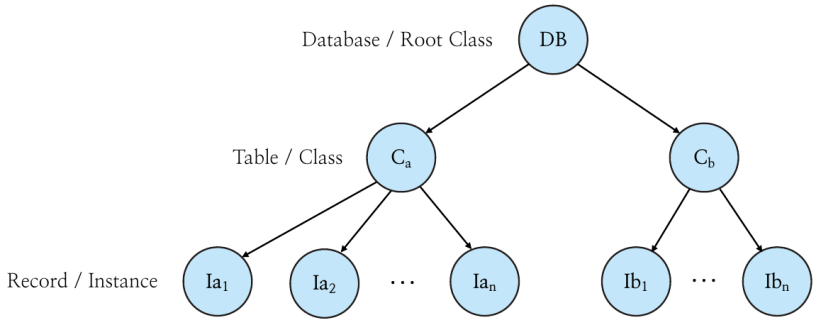
파트 II

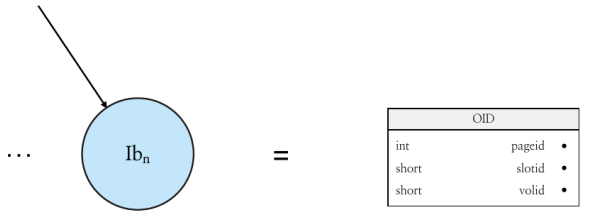
CUBRID는 어떻게 다이얼을 조정하는가?

CUBRID

OID로 모든 것에 이름 붙이기

CUBRID의 세분화 구조





데이터베이스, 테이블, 리소스 자원들은
모두 OID로 표현될 수 있습니다

```
typedef struct db_identifier DB_IDENTIFIER;
struct db_identifier
{
  int pageid;
  short slotid;
  short volid;
};

typedef DB_IDENTIFIER OID;
```

dbtype_def.c: 969

CUBRID

세 가지 핵심 타입

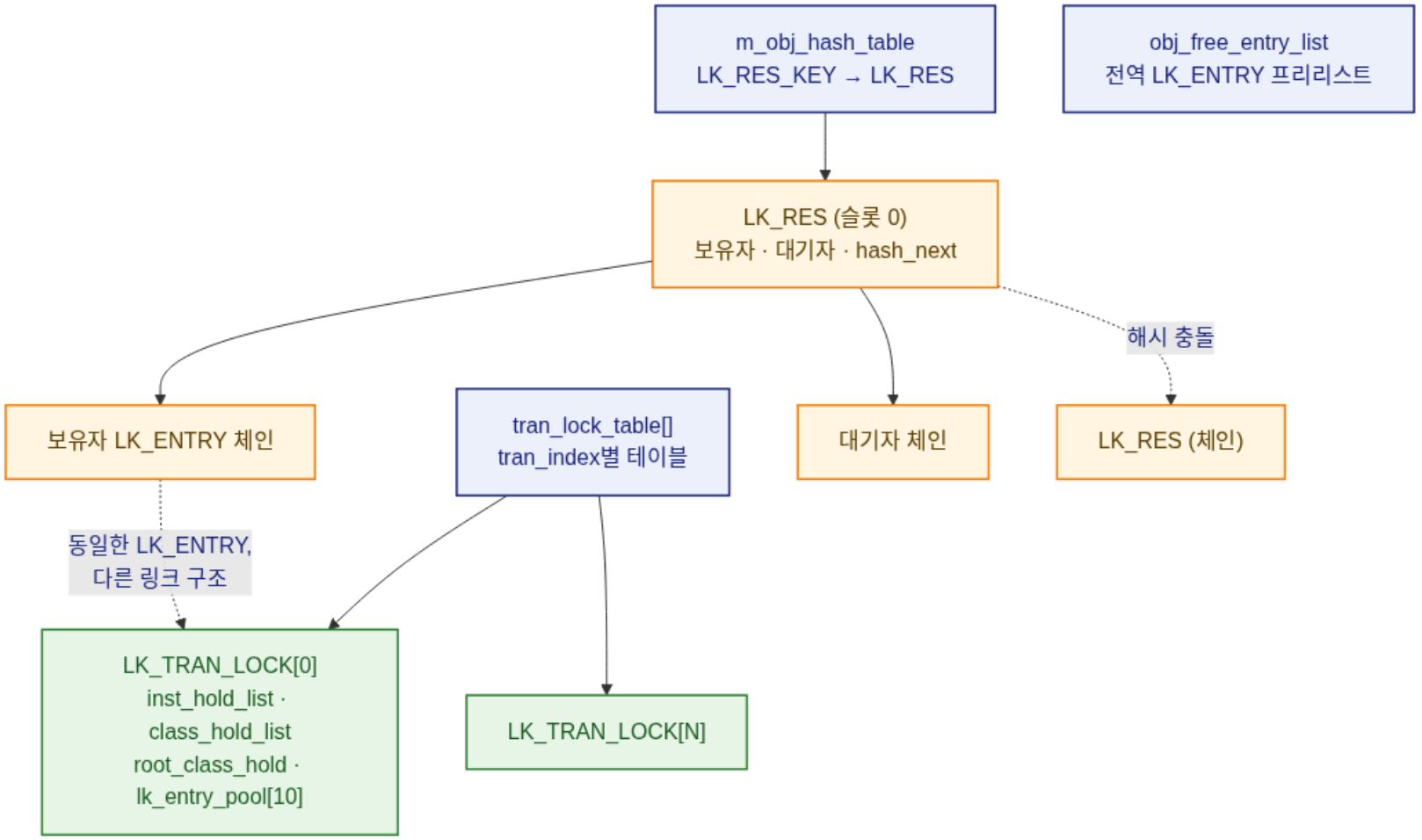
```
// src/transaction/lock_manager.h
struct lk_res_key { LOCK_RESOURCE_TYPE type; OID oid; OID class_oid; };

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
};
```

> 집계 필드 덕분에 각 호환성 검사가 O(보유자)에서 O(1)로 줄어든다.

전역 잠금 테이블 — `lk_global_data`



세 전역 구조: 해시 테이블(`MAX_NTRANS × 300` 슬롯), 10개 로컬 풀을 가진 트랜잭션별 테이블, 초과분을 위한 공유 프리리스트.

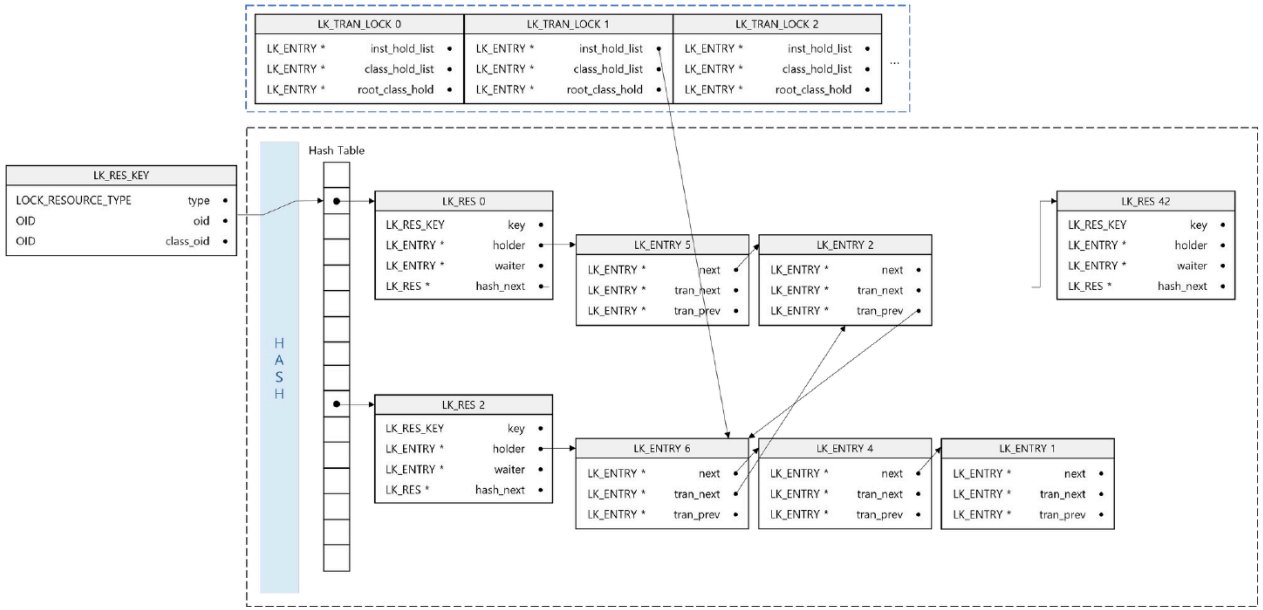
엔트리 초기화 — 허가 vs 차단

```
// 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
  /* ... */
}
```

같은 구조체, 두 가지 생성자. **보유자**는 `granted_mode`가 활성 필드이고, **대기자**는 `blocked_mode`가 활성이다. **변환** 진행 중에는 두 필드를 동시에 사용한다. `granted_mode`는 현재 모드이고 `blocked_mode`는 업그레이드 목표다.

이중 뷰 스레딩



- 동일한 `LK_ENTRY`가 **두 리스트에 동시에** 속한다.
- 리소스 뷰: **R을 누가 보유하는가?** → `next`. 트랜잭션 뷰: **T는 무엇을 보유하는가?** → `tran_next/prev`.
- 커밋 시에는 트랜잭션 리스트를 순회한다. 호환성 검사는 리소스 측의 집계값을 읽는다.

12개 모드 어휘

값	모드	의미
0	`NA` / `NULL`	플레이스홀더
1	`NON2PL`	RC에서 MVCC-비활성 클래스의 S 잠금을 조기 해제한 기록자
3	`SCH-S` / `SCH-M`	스키마 안정 / 변경 (DDL은 SCH-M 취득)
4	`IS` / `IX`	인텐션 공유 / 인텐션 배타
5	`S`	공유
7	`BU`	일괄 갱신 (loaddb 경로)
8	`SIX`	공유 + 인텐션 배타
9	`U`	공유 상태로 X 업그레이드 예정
10	`X`	배타

> Gray의 5개 모드(`IS / IX / S / SIX / X`) + CUBRID 추가 7개.

SQL 구문이 잠금 모드에 매핑되는 방식

하나의 구문은 일반적으로 **두 개의** 잠금을 취득한다. 클래스 단위 **인텐트** 잠금(`lock_scan` 경유)과 행 단위 **실제** 잠금(`lock_object` 경유)이다.

SQL	클래스 단위 (`lock_scan`)	행 단위 (`lock_object`)
`SELECT` **(일반)**	`IS`	MVCC-활성 클래스에서는 **없음** · MVCC-비활성 클래스(시리얼 등)에서는 `S`
`SELECT … FOR UPDATE`	`IX`	`U` ← "업그레이드 예정 S"
`INSERT`	`IX`	새 행에 `X`
`UPDATE`	`IX`	각 갱신 행에 `X`
`DELETE`	`IX`	각 행에 `X`
`LOAD DATA` / 일괄 적재	`BU`	`X`
`CREATE / ALTER / DROP TABLE`	`SCH-M`	— (DDL은 클래스 수준만)
DDL 진행 중 클래스에 대한 `SELECT`	`SCH-S`	— ← `SCH-M` 뒤에서 대기

- `IX` 행이 반복되는 것은 의도적이다. **모든** 행 단위 쓰기는 **동일한** 클래스 단위 인텐트 모드를 취득한다. 호환성 행렬이 나머지를 처리한다. `U`(업그레이드 예정 `S`)는 예외로, 변환 슬라이드에서 다시 설명한다.

호환성 행렬 + 기아 방지 장치

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

새 요청은 `total_holders_mode`와 `total_waiters_mode` **모두**와 호환되어야 한다. 후자가 **기아 방지 장치**로, 연속적인 `S` 요청이 대기 중인 `X`를 앞질러 가는 것을 막는다.

CUBRID

작동 예제 — 기아 방지 장치

행 R에 세 트랜잭션이 이 순서로 접근한다:

t	동작	보유자	대기자	total_holders	total_waiters	결과
1	T1: S 요청	T1(S)	–	S	NULL	**허가**
2	T2: X 요청	T1(S)	T2(X)	S	X	**대기** (X ∧ S = ✗)
3	T3: S 요청	T1(S)	T2(X)	S	X	**대기** — 보유자와는 호환되지만, 대기자와 S ∧ X = ✗

- t = 3에서 "보유자만" 검사했다면 T3에게 허가했을 것이다(S ∧ S = ✓).
- 양방향 검사가 T3을 T2 뒤에 줄 세운다 — 공정성이 요구하는 정확한 동작이다.

> 대기자 검사 없이는 꾸준한 S 요청이 대기 중인 모든 X를 무한정 굶길 것이다. PostgreSQL은 이를 **strong-lock fairness**라는 이름으로 동일하게 구현하고 있다.

CUBRID

잠금 변환 — `lub` 규칙

- `lub` = **최소 상한(least upper bound)**. 강도 순으로 정렬된 잠금 모드 격자에서, `lub(A, B)`는 A와 B를 **동시에** 만족하는 **가장 작은** 모드다.
- 자신의 잠금을 재요청할 때는 **업그레이드**해야지, 자기 자신과 교착 상태에 빠져서는 안 된다.
- 규칙: `granted_mode ← lub(granted_mode, requested_mode)`
  - `lub(S, IX) = SIX` — 공유와 쓰기 인텐션 모두 필요
  - `lub(IS, S) = S` — `S`는 이미 `IS`를 내포함
  - `lub(S, X) = X` — `X`는 이미 `S`를 내포함
- 구현: `(보유, 요청)`으로 인덱싱된 **정방형 테이블** — `src/transaction/lock_table.c`.
- 변환 중인 보유자는 업그레이드가 완료될 때까지 `granted_mode`(현재)와 `blocked_mode`(목표) 두 필드를 모두 설정한다.

작동 예제 A — 충돌 없는 자기 업그레이드

T1이 같은 행 `r`을 읽은 후 쓰는 일반적인 DML 패턴:

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

단계	T1의 요청	`granted_mode`	`blocked_mode`	`LK_ENTRY` 동작
1	행 5에 `S` 잠금	`S`	`NULL_LOCK`	새 엔트리; `r`의 보유자 리스트에 추가
2	요청 모드 `X` 도달	—	—	변환 테이블: `lub(S, X) = X`. 다른 보유자 없음 ⇒ **인플레이스 업그레이드**
3	업그레이드 후	`X`	`NULL_LOCK`	**동일한** `LK_ENTRY`, 모드 필드만 변경

- `LK_RES`의 집계도 함께 변경된다: `total_holders_mode`가 `S` → `X`. 보유자 **카운트**는 1 유지.
- 자신의 잠금을 재요청하는 것은 "잠금 해제 + 재획득"이 **아니다**. 변환 테이블은 교착 상태 가능성이 있는 그 과정을 원자적 전환 하나로 대체한다.

왜 중요한가. 변환 테이블 없이는 모든 업그레이드가 보유 잠금을 먼저 해제해야 하므로, 다른 트랜잭션이 끼어들 틈이 생긴다. 그러면 T1은 자신이 방금 해제한 잠금을 기다리며 교착 상태에 빠질 수 있다.

작동 예제 B — 충돌이 있는 자기 업그레이드 대기

같은 코드지만, **T1이 시작할 때 다른 트랜잭션 T2가 이미 행 `r`에 `S`를 보유**하고 있다.

t	T1 (업그레이더)	T2 (다른 읽기)	r의 보유자	T1의 `LK_ENTRY`
1		r에 `S` 획득	T2(S)	—
2	r에 `S` 획득 → **허가** (S ∧ S = ✓)		T1(S), T2(S)	`granted_mode = S`
3	`X` 요청 (UPDATE)	(S 보유 중, 트랜잭션 진행 중)	T1(S), T2(S)	`granted_mode = S` · `blocked_mode = X` ← **대기 중**
4	r의 대기자 리스트에서 **일시 중단**	커밋 → S 해제	T1(S)	변동 없음
5	깨어남; `total_holders_mode = S`(자신) 호환 통과		T1(X)	`granted_mode = X` · `blocked_mode = NULL_LOCK`

사례 A와의 차이: 단계 3에서 T1은 **동시에** 보유자(`S`)이면서 대기자(for `X`)다. T1은 대기 중에도 `S`를 해제하지 않는다. 이것이 `LK_ENTRY`가 두 개의 모드 필드를 갖는 이유다.

CUBRID

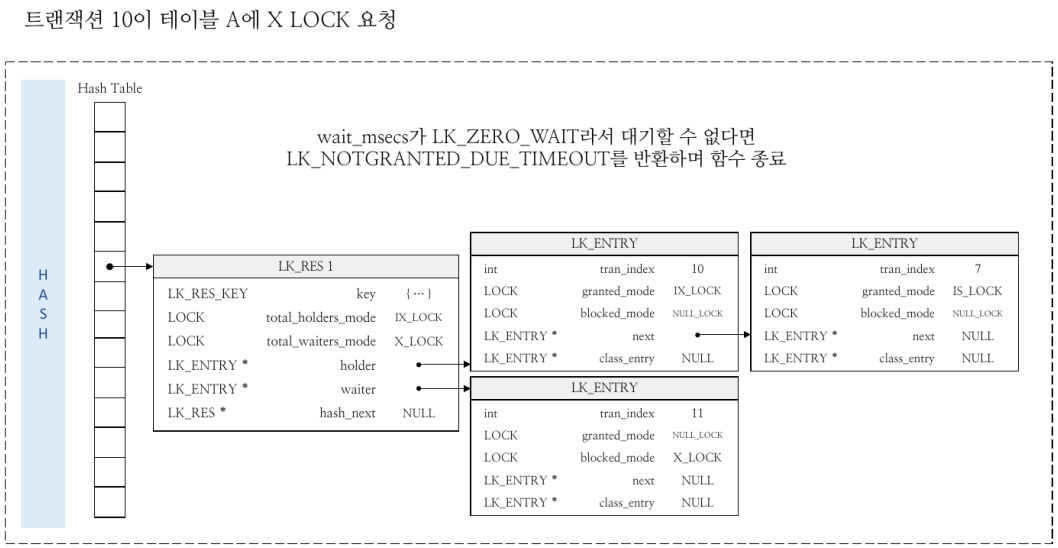
자기 업그레이드 — 사례 A vs 사례 B 비교

	(충돌 없음)	(충돌 보유자 존재)
단계 2 전 보유자 리스트	`T1(S)`	`T1(S), T2(S)`
변환 요청 시점	`granted_mode = S → X` 직접 전환	`granted_mode = S`, `blocked_mode = X`
T1이 대기 중 `S`를 해제하는가?	해당 없음 — 대기 없음	**아니오**, S를 계속 보유
업그레이드 차단 해제 조건	없음 — 즉시	T2의 커밋(또는 롤백)
T1의 `LK_ENTRY` count	항상 1	항상 1
사용되는 모드 필드	`granted_mode` 만	**`granted_mode` 와 `blocked_mode` 모두**

동일한 `LK_ENTRY` 형태, 두 가지 코드 경로. 변환 테이블은 둘 모두를 포함한다. 행렬의 모든 셀에 `lub`가 정의되어 있으므로, 항상 업그레이드 **목표** 모드가 존재한다.

CUBRID

획득 흐름



- **두 가지 API 표면.** `lock_scan(class_oid, mode)` — 클래스 단위 인텐션 잠금. `lock_object(record_oid, …)` — 인스턴스 단위 실제 잠금.
- `lock_internal_perform_lock_object` 내부: 새 리소스 → 허가. 그 외에는 `total_holders_mode ∧ total_waiters_mode` 호환성 검사 → 허가, 대기자로 삽입, 또는 일시 중단.
- `enum LOCK_WAIT_STATE`의 깨어남 원인: `LOCK_RESUMED`, `_TIMEOUT`, `_ABORTED`(교착 상태 피해자), `_INTERRUPT`.

획득 상태 기계 — `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 */   ...
}
```

> 세 가지 결과. 호환성 검사는 집계값 **두 개 모두**(`보유자 | 대기자`)를 읽는다 — 바로 기아 방지 장치다.

해제: 하나의 조절 장치가 아닌 세 가지 범위

```
// lock_unlock_object — src/transaction/lock_manager.c
if (force) {                                // commit / rollback path
  lock_internal_perform_unlock_object (..., false, true);
  return;
}
if (lock != S_LOCK) return;                 // X is commit-bound
switch (logtb_find_isolation (tran_index)) {
  case TRAN_SERIALIZABLE: case TRAN_REPEATABLE_READ: return;
  case TRAN_READ_COMMITTED:
    lock_unlock_object_by_isolation (...); break;
}
```

- **X 잠금** — 항상 커밋까지 유지된다. 위의 `lock != S_LOCK` 단락은 소스에서 *"이것들은 해제하지 않는다"*를 의미한다.
- **S 잠금** — MVCC-활성 클래스에서는 없다(일반 SELECT는 행 잠금을 취득하지 않으며, 가시성은 스냅샷 기반). MVCC-비활성 클래스(루트, 시리얼, 콜레이션, HA apply-info)에서는 RC가 `lock_unlock_object_by_isolation`을 통해 구문 단위로 해제하고, RR / SERIALIZABLE은 커밋까지 유지한다.
- **클래스 잠금** — 항상 커밋까지 유지된다(인텐션 잠금은 자식보다 오래 유지됨).

작동 예제 — RC와 RR의 같은 타임라인 비교

> **적용 범위.** 교과서적인 2PL 시나리오다. CUBRID에서는 MVCC-비활성 클래스(시리얼 등)에만 해당한다. MVCC 사용자 테이블에서는 T2가 `S` 잠금을 취득하지 않으며, 동일한 이상 현상은 스냅샷 타이밍으로 처리된다 — cubrid-mvcc.md 참고.

T2가 **한 트랜잭션 안에서** 두 번 읽는다. T1은 두 읽기 사이에 수정하고 커밋한다. T2는 두 번째 읽기에서 무엇을 보는가?

`t`	T1 (쓰기)	T2 (읽기, 한 트랜잭션)
1		`SELECT WHERE id=r` → `1000` · `S` 취득
2	`UPDATE WHERE id=r` — `X` 요청	
3	T1 **커밋 (가능한 경우)**	
4		`SELECT WHERE id=r` — 두 번째 읽기

T2 격리 수준	`S` 해제 시점	`t = 2`에서 T1	`t = 4`에서 T2 읽기
RC	구문 종료 시 (`t = 1`)	`X` 허가 → 커밋	**1100** — **반복 불가 읽기**
RR	T2 커밋 시	T2의 `S` 때문에 `X` **대기**	**1000** — **차단됨**

잠금 에스컬레이션

```
// lock_escalate_if_needed — src/transaction/lock_manager.c
if (!lock_check_escalate (th, class_entry, tran_lock))
  return LK_NOTGRANTED;                          // threshold not reached
if (class_entry->granted_mode == IX_LOCK
    || class_entry->granted_mode == SIX_LOCK)
  max_class_lock = X_LOCK;                       // writer  → class X
else
  max_class_lock = S_LOCK;                       // reader  → class S
granted = lock_internal_perform_lock_object (... , max_class_lock,
            LK_FORCE_ZERO_WAIT, &class_entry, NULL);
```

- 한 클래스에 대한 인스턴스 잠금이 `lock_escalation_threshold`를 초과하면: 행 단위 잠금을 모두 삭제하고 **클래스 단위** 잠금 하나를 취득한다.
- IX / SIX → X 선택은 **휴리스틱**이다 — 행 단위 S와 X를 개수 세는 것보다 저렴하다.
- `LK_ENTRY`는 `ngranules`(이 인텐션 잠금 하위의 자식 수)와 `class_entry`(부모 클래스 단방향 포인터)를 보유한다. 에스컬레이터는 두 필드를 모두 사용한다.
- 트레이드오프: 메모리 절감 ↔️ 같은 클래스의 다른 트랜잭션에 대한 의도치 않은 직렬화.

작동 예제 — 대량 DML에서의 에스컬레이션

유지보수 작업이 실행된다:

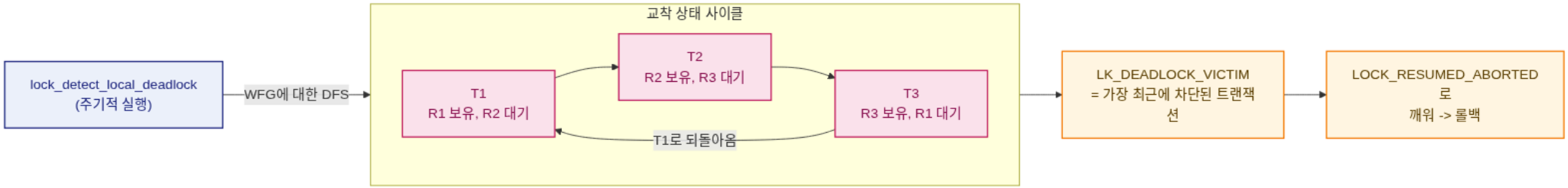
```
UPDATE accounts SET status = 'archived' WHERE created_at < '2020-01-01';
-- imagine 50,000 rows match
```

단계	잠금 관리자의 동작	`LK_ENTRY` 수 (이 트랜잭션)
1	`lock_scan(accounts, IX)` → 클래스 단위 인텐트	**1** (클래스 IX)
2	`lock_object(row_1, X)`, `(row_2, X)`, …	1 + N 행 엔트리
3	행 `N = lock_escalation_threshold` (예: 10,000)에서 `lock_escalate_if_needed` 발동	에스컬레이션 트리거
4	모든 행 단위 `X` 엔트리 삭제; 클래스 `IX` → 클래스 `X` 변환	**1** (클래스 X)
5	나머지 40,000행은 클래스 `X`로 보호 — 새 행 단위 엔트리 없음	**1** 유지

트레이드오프의 득실:

- 메모리가 `O(접촉한 행 수)`에서 `O(1)`로 줄어든다.
- 이후 이 클래스에 대한 호환성 검사는 50,000번의 리스트 탐색이 아닌 행렬 조회 한 번으로 완료된다.
- 다른 트랜잭션이 `accounts`의 **어떤** 행에도 접근하려면 이제 클래스 `X`에서 차단된다 — 트랜잭션 기간 동안 동시성이 무너진다.

교착 상태 — 대기-위한 그래프



- T1이 T2가 보유한 잠금을 기다린다 → WFG 엣지 T1 → T2 추가(`LK_WFG_EDGE`).
- `lock_detect_local_deadlock` 이 WFG를 순회(DFS)한다. 사이클 발견 → **피해자 = 가장 최근에 차단된 트랜잭션** → `LOCK_RESUMED_ABORTED` → 롤백.
- "**로컬**" — 분산 교착 상태는 타임아웃으로 처리된다.

교착 상태 감지기 — `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;
}
```

작동 예제 — 두 트랜잭션, 두 행

설정: `accounts (id, balance)`, 초기값 `A.balance = 1000`, `B.balance = 1000`. 두 세션이 동시에 시작된다:

트랜잭션	구문	의도
T1	`UPDATE accounts SET balance = balance - 100 WHERE id = 'A'`	A에 X 잠금
T1	`UPDATE accounts SET balance = balance + 100 WHERE id = 'B'`	B에 X 잠금
T2	`UPDATE accounts SET balance = balance -  50 WHERE id = 'B'`	B에 X 잠금
T2	`UPDATE accounts SET balance = balance +  50 WHERE id = 'A'`	A에 X 잠금

T1은 A에서 B로 이체하고, T2는 B에서 A로 이체한다. 스케줄은 스케줄러가 두 트랜잭션을 어떻게 교차 실행하느냐에 달려 있으며, 그 중 하나는 불운을 맞게 된다.

작동 예제 — 교차 실행이 사이클을 만든다

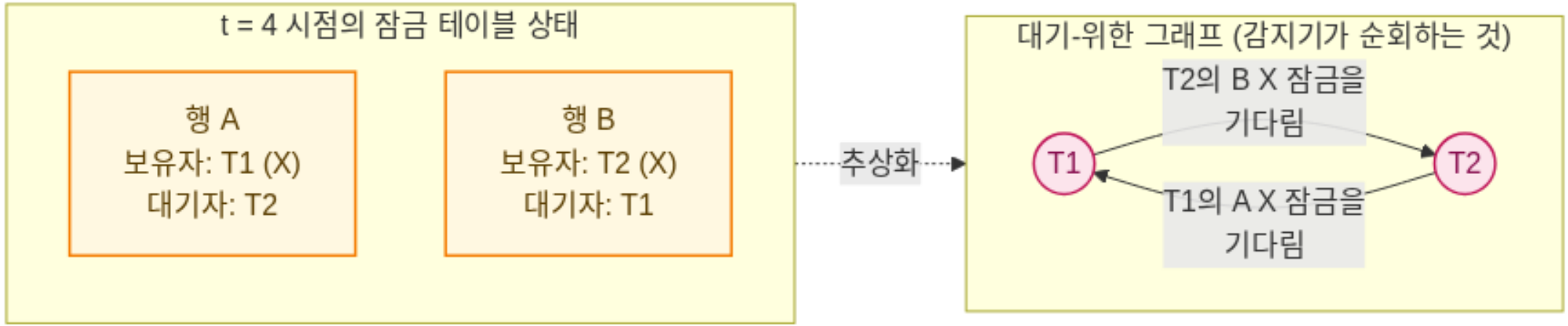
t	T1 동작	T2 동작	잠금 테이블 상태
1	A X 잠금 → **허가**		A: 보유자 = T1
2		B X 잠금 → **허가**	A: T1 ; B: T2
3	B X 잠금 → **대기**		A: T1 ; B: T2, 대기자 T1 · WFG: **T1 → T2**
4		A X 잠금 → **대기**	A: T1, 대기자 T2 ; B: T2, 대기자 T1 · WFG: **T1 → T2 → T1** (사이클)

- `t = 4` 이후 **두 트랜잭션 모두** `LOCK_SUSPENDED` 상태다.
- 외부 개입 없이는 어느 쪽도 진행하지 못한다. WFG에 길이 2의 닫힌 사이클이 형성되었다.

> 요청별 타임아웃도 결국 이 상황을 해결하겠지만, 몇 분의 낭비가 생긴다 — 감지기는 수십 밀리초 안에 해결한다.

CUBRID

작동 예제 — 사이클 시각화



- **왼쪽**: 구체적인 잠금 테이블 — 각각 보유자와 대기자를 가진 두 개의 `LK_RES` 레코드.
- **오른쪽**: 추상적인 WFG — 활성 대기자마다 노드 하나, `"대기자가 보유자를 기다림"` 관계마다 방향 있는 엣지 하나.
- **감지기는 행을 검사하지 않는다.** 오른쪽 그래프를 순회한다.

> WFG는 잠금 테이블로부터 도출된다: 리소스의 모든 `(대기자, 보유자)` 쌍이 방향 엣지 `대기자 → 보유자`가 된다. 그 그래프의 사이클 ≡ 교착 상태.

작동 예제 — 감지기가 해결한다

`lock_detect_local_deadlock`의 한 틱이 발동된다(매 대기마다가 아닌 **주기적으로** 실행됨):

1. **스냅샷.** `lk_Gl`의 활성 대기자를 순회한다. T1과 T2 모두 `LOCK_SUSPENDED` 상태이므로 WFG에 추가한다.
2. **DFS.** T1에서 시작 → T2로 엣지를 따라감 → T1으로 엣지를 따라감. DFS 스택에 이미 있는 노드로의 **역방향 엣지** → 사이클 발견.
3. **피해자.** 사이클 내 노드 중 `thrd_wait_stime`이 **가장 최근인** 것을 선택한다. 여기서는 **T2**(`t = 4`에 차단됨).
4. **깨우기.** T2의 대기 상태를 `LOCK_RESUMED_ABORTED`로 설정한다.
5. **롤백.** T2의 획득 루프가 중단 플래그를 확인 → 트랜잭션 롤백 트리거 → B에 대한 X 잠금 해제.
6. **T1 차단 해제.** B에 대한 T1의 대기 중인 X 호환성 검사가 통과된다(보유자 없음). T1이 허가되어 나머지 구문을 실행하고 결국 커밋한다.

> 실행 중인 트랜잭션에서 잠금을 **빼앗는** 일은 없다. 패자가 롤백 비용을 치른다. 승자는 약간의 추가 대기만 느낄 뿐이다.

CUBRID

작동 예제 — 세 트랜잭션 사이클

세 트랜잭션, 세 행. 각각 행 하나를 잡은 다음 다음 행을 요청한다 — 순환적인 `T1 → T2 → T3 → T1` 체인.

t	T1	T2	T3	WFG 상태
1	R1에 X → **허가**			—
2		R2에 X → **허가**		—
3			R3에 X → **허가**	—
4	R2에 X → **대기**			T1 → T2
5		R3에 X → **대기**		T1 → T2 → T3
6			R1에 X → **대기**	T1 → T2 → T3 → **T1** (사이클)

- **DFS**가 T1에서 T1→T2→T3 순으로 순회하다가, T1으로의 역방향 엣지 — 깊이 3에서 사이클 발견.
- **피해자** = **T3**(`thrd_wait_stime` 최신). T3 중단 → R3 해제 → T2 깨어남 → R2 해제 → T1 깨어남.

슬라이드에 담지 못한 내용 — 소스에서 읽기

핵심 결정 지점(구조체 레이아웃, 엔트리 초기화, 획득, 해제, 에스컬레이션, 교착 상태 감지)은 각 슬라이드에 인라인으로 표시되어 있다. 아래 항목들은 슬라이드에 담기지 않았으므로 소스에서 직접 읽어야 한다.

주제	심볼	파일
깨어남 상태 열거형 (전체 8개 값)	`LOCK_WAIT_STATE`	`lock_manager.c`
모듈 초기화 / 종료	`lock_initialize` , `lock_finalize`	`lock_manager.c`
`NON2PL` 엔트리 생성자	`lock_initialize_entry_as_non2pl`	`lock_manager.c`
12×12 호환성 / 변환 테이블 (소스)	`lock_Comp` , `lock_Conv`	`lock_table.c`
대기-위한 그래프 추상화 (헤더 + 구현)	`wfg_*` 패밀리	`wait_for_graph.{h,c}`

> 줄 번호는 리팩터링에 따라 변한다. `git grep -n '<symbol>' src/transaction/` 이 유용하다.
> 분석 문서(`knowledge/code-analysis/cubrid/cubrid-lock-manager.md`)는 `updated:` 날짜 기준의 구체적인 줄 번호 위치 힌트 테이블을 유지한다.

CUBRID 너머 — 연구 동향

방향	한 줄 요약
Bamboo (2021)	커밋 전 X 잠금 해제. CUBRID의 `NON2PL`을 일반화.
Brook-2PL (2025)	정적 의존성 사전 분석 → 교착 상태 없는 2PL.
TXSQL (2025)	적응형 잠금 모드 조정 + 경합 인지 스케줄링.
OCC (Hekaton / Silo / Cicada)	잠금 테이블을 커밋 시점 검증으로 대체.
SSI (PostgreSQL)	술어 잠금 — 직렬화 가능성으로 가는 저비용 경로. CUBRID의 SERIALIZABLE은 first-updater-wins 방식의 스냅샷 격리 (`ER_MVCC_SERIALIZABLE_CONFLICT`)이며, 범위 잠금이 아님.
VLL	잠금 테이블 자체가 병목일 때 분할.

> CUBRID의 다이얼 위치: **교과서적 2PL + MGL + WFG 감지**. 잘 정립된 선택이다.

감사합니다

Q & A

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

부록

부록 A — 잠금(Lock) vs 래치(Latch), 나란히 비교

비슷하게 들리는 두 단어. 같은 DBMS 안에서도 완전히 다른 기계다.

관점	잠금(Lock)	래치(Latch)
보호 대상	트랜잭션 직렬화 순서 **(논리적)**	물리적 구조 무결성 **(분할, 재균형 중)**
유지 시간	트랜잭션 전체 (또는 RC에서 구문 단위)	임계 구역 기간 (마이크로초)
저장 위치	외부 잠금 테이블(`LK_RES` , `LK_ENTRY`)	페이지 / 구조체 내부(`PGBUF_LATCH`)
획득 규율	2PL (성장 → 수축)	래치 커플링, 고정 잠금 순서, 설계상 교착 상태 없음
세분성	OID 형태: 데이터베이스, 클래스, 인스턴스	페이지, 리스트, 해시 버킷
충돌 해결	대기 → WFG 사이클 탐색 → 피해자 롤백	스핀 또는 슬립; 규율을 지키면 교착 상태 없음

> **Database Internals** ch. 5의 첫 번째 원칙: 이 둘을 혼동하지 말라.

부록 A — 잠금(Lock) vs 래치(Latch), 코드에서 (CUBRID 힙 삽입)

```
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
```

- 페이지 래치는 **마이크로초** 동안만 존재한다 — 바이트가 변경되는 동안만.
- 행 잠금은 **분에서 시간** 단위까지 유지된다 — 트랜잭션이 커밋할 때까지.
- 두 시간 단위는 자릿수 차이가 난다. 이를 혼동하면 구조적 버그가 된다:
  - **페이지 단위 임계 구역 동안 잠금을 보유하면** 전체 작업 부하가 그 페이지를 통해 직렬화된다.
  - **네트워크 왕복 동안 래치를 보유하면** 전체 작업 부하가 그 스레드를 통해 직렬화된다.
- B-트리 분할은 **래치만** 사용한다 — 트랜잭션 가시성 의미가 없다.