

pre-flush 리더 경로

staging 중인 undo 이미지를 락 없이 읽는다

이 PR 은 읽기 처리량을 **2.05배**로 올린다고 주장한다. 그런데 그 방법이 “아직 로그 페이지에 복사되지 않은 메모리 노드를 락 없이 읽는다” 이므로, 리뷰어가 실제로 판정해야 하는 것은 성능이 아니다. 세 가지다.

- 1 읽는 대상이 유효한가**
MVCC 리더가 집은 노드가 그 순간 완성돼 있고, 읽는 동안 해제되지 않는가
- 2 판정이 보수적인가**
“복사됐다” 를 잘못 믿어 필요한 drain 을 건너뛰는 경우가 없는가
- 3 고장 났을 때 어디로 떨어지는가**
새 기계를 못 쓰는 상황에서 옛 경로로 내려가는가, 아니면 잘못된 답을 내는가

오늘 순서 — 문제 → **AS-IS** (왜 그렇게 되어 있는가) → 설계 → 코드와 시나리오 → 판정

MVCC 리더가 `prev_version_lsa` 사슬을 따라갈 때 **hop** 마다 목표 LSA 를 원자적 watermark `copied_lsa` 와 비교한다. 아래면 기존 경로로 가고, 위면 **in-flight** 윈도우에서 staging 중인 노드를 찾아 그 노드의 `udata` 에서 직접 읽는다. 윈도우에도 없으면 watermark 를 한 번 더 읽고, 그래도 위면 그때만 예전처럼 `LOG_CS` 를 잡는다.

	값
변경 규모	17 파일 · +813 / -92 · 커밋 10
새 파일	<code>log_prior_inflight.{hpp,cpp}</code> (+446)
새 인터페이스	없음 — 파라미터 없음, 디스크 포맷 변경 없음, 반환값 변경 없음
새 관측점	<code>statdump</code> 카운터 다섯
함께 들어간 별개 수정	CBRD-27044 — <code>lockfree::tran</code> 회수 영구 스톨

MVCC 리더가 이전 버전 하나를 읽으려다 전역 임계구역 **LOG_CS** 를 **쓰기 모드**로 잡고, prior list 전체를 로그 페이지로 밀어 넣는다 — drain 은 원래 리더의 일이 아니다. 그동안 그 리더는 **heap 페이지 latch** 를 **진 채** 기다리고, 뒤로 writer 전부와 flush 데몬이 선다.



같은 drain 인데 **데몬이 하면 회당 0.3 μ s**, 리더 가드가 하면 회당 **1.6 ms** 다. 네 자릿수 차이가 전부 대기다.

짧은 코드에 문제가 세 겹

① 전역 직렬점

성능. `LOG_CS` 는 로그 모듈 전체를 덮는 하나다. MVCC 리더가 쓰기 모드로 켜면 모든 writer 의 drain 과 flush 데몬의 진행이 멈춘다. 리더를 늘려도 읽기 처리량이 오르지 않는 구조다.

② 비원자 읽기

정확성. `append_lsa` 는 `pageid:48` + `offset:16` 두 비트필드다. 락 없이 구조체 대입으로 읽으면 찢어진 값을 볼 수 있고, 코드의 `TODO` 가 이미 그것을 인정한다.

③ 루프 밖 한 번

구조. 가드는 첫 hop 만 본다. 두 번째 이후 hop 은 `logpb_fetch_page` 가 다시 판정하므로 가드는 그 판정의 **중복** 이고, 오히려 그쪽이 락 안에서 재확인까지 한다.

그리고 이 대기는 전부 **heap 페이지 latch** 를 **원 채** 일어난다 — 비용이 리더 자신에 그치지 않고 같은 페이지를 만지려는 트랜잭션까지 뒤로 민다.

```
// heap_get_visible_version_from_log - src/storage/heap_file.c (AS-IS)
oldest_prior_lsa = *log_get_append_lsa (); /* TODO: fix atomicity */
if (LSA_LT (&oldest_prior_lsa, previous_version_lsa))
{
    LOG_CS_ENTER (thread_p);
    logpb_prior_lsa_append_all_list (thread_p);
    LOG_CS_EXIT (thread_p);

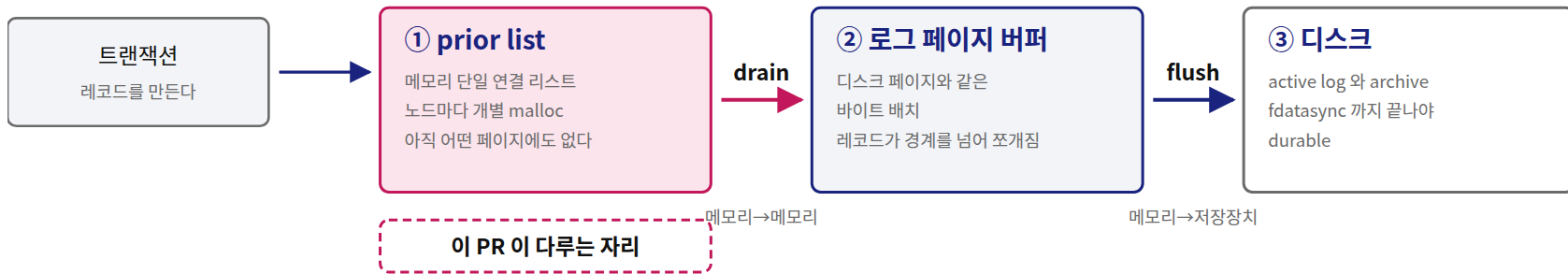
    oldest_prior_lsa = *log_get_append_lsa ();
    assert (!LSA_LT (&oldest_prior_lsa, previous_version_lsa));
}
```

- ▶ 의도는 명확하다 — 아래에서 부를 `log_get_undo_record ()` 가 “이 레코드는 이미 페이지에 있다” 를 전제하므로, 그 전제를 만들어 놓고 들어간다.
- ▶ 실제로 그 함수 맨 앞에 같은 전제를 확인하는 `assert` 가 있다.
- ▶ 리더 사슬은 넷이다 —
`heap_get_visible_version` →
`heap_get_visible_version_from_log`
→ `logpb_fetch_page` →
`log_get_undo_record` .

가드는 `append_lsa < 목표` 로 판정하고 `logpb_fetch_page` 는 `≤` 로 판정한다 — 등호를 포함해 더 보수적이고, 락 안에서 한 번 더 확인한다. **가드에는 그 겹이 없다.**

로그 레코드가 지나는 세 곳

로그 레코드는 만들어져 디스크에 닿기까지 **세 자리**를 거친다. 그 사이의 **두 이동**에 붙는 이름이 **drain** 과 **flush** 인데 흔히 섞어 쓴다 — 이 데크는 그 둘을 갈라 쓴다.



drain = ① → ② (메모리 → 메모리), **flush** = ② → ③ (메모리 → 저장장치). 이 PR 이 다루는 자리는 **① prior list** 하나다.

AS-IS 호출 경로 — 세 갈래

코드를 열고 따라갈 때 쓰는 지도다. 오른쪽 칸이 **이 PR** 이 각 갈래에 무엇을 얹었는지다.

갈래	진입점부터	무엇을 하는가	이 PR 이 얹는 것
생산 append	<code>prior_lsa_alloc_and_copy_data () / _crumbs ()</code> → <code>prior_lsa_next_record_internal ()</code> → <code>prior_lsa_start_append () ...</code> <code>prior_lsa_end_append ()</code>	락 밖에서 노드 완성 → <code>prior_lsa_mutex</code> 안에서 LSA 발급·링크·attach	register 가 여기 붙는다
복사 drain	<code>logpb_prior_lsa_append_all_list ()</code> → <code>logpb_append_next_record ()</code> (노드마다)	호출 지점 다섯 — flush 데몬 · 직접 flush · <code>logpb_fetch_page</code> 의 락 안 재확인 · 생산자 backpressure · MVCC 리더 가드	발행·retire 가 여기 붙는다
읽기 reader	<code>heap_get_visible_version ()</code> → <code>heap_get_visible_version_from_log ()</code> → <code>logpb_fetch_page ()</code> → <code>log_get_undo_record ()</code>	hop 마다 사슬을 내려간다. heap 페이지 latch 를 쥔 채다	총 판정이 여기 끼어든다

다섯째 호출 지점이 이 리뷰의 출발점이다 — drain 을 부르는 다섯 중 넷은 쓰는 쪽이거나 데몬인데, **하나가 MVCC 리더**다. 4번에서 “역할이 뒤집혀 있다” 고 한 것이 이 표의 그 줄이고, 이 PR 이 지우는 것도 그 줄 하나다.

drain 과 flush 는 언제 도는가

drain — prior list → 페이지 버퍼

① 직접 flush 직전	커밋마다 — 내보내기 전에 항상 비운다
② flush 데몬	주기적 wakeup (그룹 커밋)
③ <code>logpb_fetch_page</code> 의 락 안 재확인	원하는 페이지가 아직 안 만들어졌을 때. 락 안에서 한 번 더 보고 그래도 없으면
④ 생산자 backpressure	prior list 가 상한(64M-256M)에 닿을 때. 서버 모드는 데몬을 깨우고 1 ms 잔다
⑤ MVCC 리더 가드	읽을 버전이 아직 복사 전일 때 — 이 PR 의 대상

```
// prior_lsa_next_record_internal () - log_append.cpp
if (log_Gl.prior_info.list_size >= (INT64) logpb_get_memsized ())
{
    /* 기준은 페이지 버퍼가 아니라 prior list 크기 */
    perfmon_inc_stat (thread_p, PSTAT_PRIOR_LSA_LIST_MAXED);
    /* 서버 모드: 직접 밀지 않고 데몬을 깨우고 잔다 */
    log_wakeup_log_flush_daemon ();
    thread_sleep (1);
    /* 1 msec - log_prior_list_maxed_wait */
    /* 복구·SA 모드면 자기가 LOG_CS 를 잡고 drain */
}
```

리더가 기다리던 것은 flush 가 아니라 drain 이었다. 다섯 갈래 전부 `logpb_prior_lsa_append_all_list ()` 로 들어가 `LOG_CS` 쓰기 모드를 요구한다. 이 PR 은 그중 ⑤ 하나만 "항상"에서 "원도가 못 줄 때만"으로 바꾼다 — 나머지 넷과 flush 정책은 그대로다.

flush — 페이지 버퍼 → 디스크

커밋	<code>log_commit ()</code> · <code>commit_postpone ()</code>
flush 데몬	그룹 커밋 주기
체크포인트 · WAL 규칙	데이터 페이지를 내리기 전에
종료 · 백업 · HA	<code>log_final ()</code> 등

직접 flush 는 항상 drain 을 먼저 한다 — 그래서 커밋마다 ① 이 함께 돈다.

이 PR 은 flush 를 순대지 않는다
그룹 커밋·fsync 배치는 내구성 측이다.

뮤텍스 밖에서 만들고, 안에서 매단다

레코드를 prior list 에 매다는 일은 두 단계로 나뉘어 있다. 이 분할이 뒤에서 안전성 논증의 바닥이 된다.

1단계 — 락 없음

`prior_lsa_alloc_and_copy_data () / _crumbs ()`

- 노드를 malloc 한다
- undo · redo payload 를 복사한다
- 기준을 넘으면 zlib 압축까지 여기서 끝낸다

2단계 — `prior_lsa_mutex` 안

`prior_lsa_next_record_internal ()`

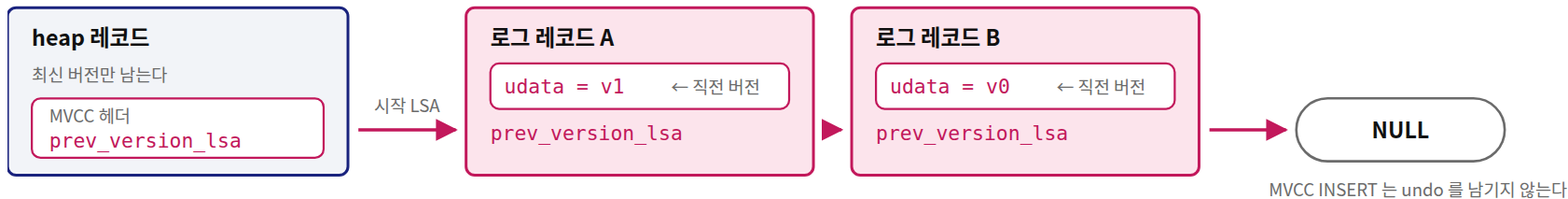
- 전역 커서에서 `start_lsa` 를 발급한다
- 링크 LSA 를 채운다 (back / forw / prev_transa)
- 리스트 꼬리에 매달고 커서를 전진시킨다

attach 반환 — 이 지점 이후 아무도 이 노드에 쓰지 않는다

attach 가 반환된 뒤에는 어떤 스레드도 그 노드의 **payload** 와 **헤더**에 쓰지 않는다. 링크 LSA 만 attach 중에 채워지고, payload 는 그 전에 이미 완성돼 있다.

로그가 버전 저장소를 겸한다

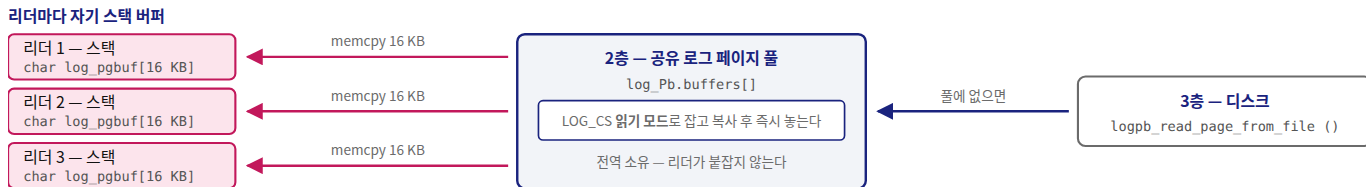
heap 레코드에는 최신 버전만 남는다. MVCC 헤더의 `prev_version_lsa` 가 로그 레코드의 **시작 LSA** 를 가리키고, 그 레코드의 undo 이미지가 직전 버전이다. 사슬은 `NULL` 에서 끝난다 — MVCC INSERT 는 되돌릴 내용이 없어 undo 를 남기지 않는다. 사슬을 한 칸 따라가 이전 버전 하나를 읽는 단위를 **hop** 이라 부른다.



미커밋 갱신자의 **undo** 를 읽는 것이 **MVCC** 읽기의 정상 경로다 — 예외가 아니다. PostgreSQL 은 heap 튜플 사슬, InnoDB 는 undo tablespace 라 **실행 중인 MVCC** 리더가 **WAL** 을 읽는 경로가 없다. CUBRID 에만 있는 제약이다.

이전 버전은 어디서 어떻게 읽어 오나

주소는 알았다. 그런데 그 바이트가 어디에 있느냐에 따라 읽는 방법이 다르고, 소유자도 다르다.



같은 undo 를 봐도 각자 한 벌씩

1층은 이 줄 전체가 없다 — 노드의 udata 를 가리켜 읽는다. 페이지 복사 0, LOG_CS 0, 경계 순회 0.

층	구조체	누가 소유하나	리더가 하는 일
1층 prior list	LOG_PRIOR_NODE	prior list 해제는 epoch 회수	가리켜 읽는다 — 복사 없음
2층 페이지 버퍼	log_Pb.buffers[]	전역 공유 풀	LOG_CS 읽기 모드 + 페이지 통째 memcpy
3층 디스크	파일	파일 시스템	내 스택 버퍼로 read

왜 굳이 복사하나 — 공유 풀의 페이지를 붙잡고 있지 않기 위해서다. LOG_CS 를 읽기 모드로 잡고 memcpy 한 뒤 바로 놓는다. 대가는 리더 수에 비례하는 16 KB 복사, 그리고 캐시가 한 겹도 없다 — hop 마다 버퍼를 무효화하고 다시 쓰므로 같은 트랜잭션이 같은 예전 버전을 반복해 봐도 매번 복사한다 (조인이 최악이다).

진행도를 재는 값이 둘이다

그 바이트가 어느 층에 있는지를 가르는 것은 **진행도**다. 그런데 진행도를 재는 값이 둘이고, 재는 단계가 서로 다르다 — `nxio_lsa` 는 **디스크까지**(“읽지 않는가”), `append_lsa` 는 **복사까지**(“읽을 수 있는가”). `nxio_lsa ≤ append_lsa` 이지만 **한쪽을 다른 쪽에서 유도해서는 안 된다** — 축이 다르다.



`append_lsa` 는 **레코드 하나를 적는 동안 네댓 번 움직인다** — 헤더 · 타입 헤더 · `undo` · `redo` · 마무리. 그래서 중간값은 **레코드 한복판**이다. 결함이 **둘인** 셈이다 — **찢어짐**(원자성)과 **중간값**(의미). 앞엿것은 원자 타입으로 고쳐지지만 **뒤엿것은 안 고쳐진다**. 19번에서 “그냥 atomic 으로 바 꾸면 되지 않나”(S5)가 기각되는 이유다.

문제는 커지는 중이다

여기까지가 지금의 그림이다. 그런데 attach 와 drain 사이의 **미복사 구간**은 앞으로 **커진다**. writer 측(CBRD-26808)이 `prior_lsa_mutex` 를 걷어내는 방향은 어느 형태로 가든 그 둘 사이를 벌리고, 그만큼 MVCC 리더가 그 구간을 만나는 빈도도 오른다 — 옵션 C 에서 그 레코드는 **정의상** 아직 그 트랜잭션의 staging 에 있기 때문이다.

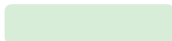
창의 정의

창의 길이

MVCC 리더가 만나는 빈도

현행

attach ~ drain 전



낮음

짧다

옵션 B — 완전 병렬 복사

예약 ~ memcpy 완료 전



낮음

μs 규모

옵션 C — 트랜잭션별 staging

예약 ~ 커밋/크기 K splice 전



높음

트랜잭션 규모

이 PR 의 설계는 그 문제를 통째로 없앤다 — staging 이 곧 읽을 수 있는 버전 저장소이므로, 창이 트랜잭션 규모든 `memcpy` 규모든 MVCC 리더는 **그 자리에서 읽는다**.

MVCC 리더가 목표 버전을 어디서 읽을지 hop 마다 고르고, “아직 복사 안 됐다” 를 기다림이 아니라 다른 읽기 경로로 처리한다.

copied_lsa 의 정의

- ▶ `append_lsa` 와 같은 진행도를 `std::atomic` 으로 복제한 값.
- ▶ 발행이 레코드 경계에서만 일어난다 — 복사가 끝난 직후의 `append_lsa` , 즉 다음 레코드의 시작이다. 그래서 뜻이 하나로 고정된다: 이 값 아래는 전부 prior list 를 떠났다.
- ▶ `append_lsa` 자체는 복사 도중에도 전진해 레코드 중간을 가리킬 수 있다 — 19번 S5 의 기각 사유가 이것이다.
- ▶ 발행은 release, 읽기는 acquire.

한쪽 방향만 안전하다

- ▶ 뒤처지면 — 원도를 보고, 없으면 재검사로 한 번 더 들어갈 뿐이다 (안전).
- ▶ 앞서면 — “이미 페이지에 있다” 고 믿고 필요한 drain 을 건너뛰어 없는 데이터를 읽는다 (치명).

그래서 이 값은 항상 store 이고 절대 `max ()` 가 아니다. 어디서 발행하든 그 시점의 `append_lsa` 를 그대로 store 하기만 하면 틀릴 수 없다.

층을 가르는 값이 copied_lsa 다

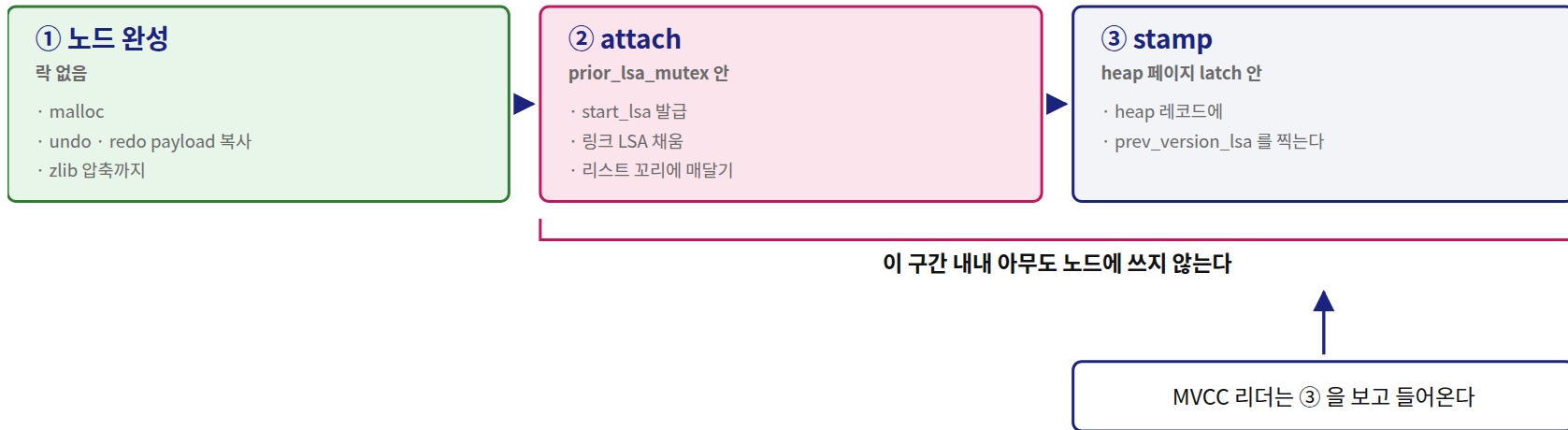
MVCC 리더가 목표 LSA 를 `copied_lsa` 와 비교한다. 위면 **1층** — 윈도우에서 staging 노드를 직접 읽고, I/O 도 대기도 `LOG_CS` 도 없다. 아래면 2층(페이지 버퍼) 또는 3층(디스크)이고, 그 둘은 기존 코드 그대로다.



이 PR 이 더하는 것은 **1층**뿐이다. 윈도우는 노드를 복사하지 않는다 — 최근 등록된 노드를 LSA 로 찾는 **고정 크기 색인**이고 prior list 의 꼬리만 덮는다.

MVCC 리더가 보는 노드는 완성돼 있다

“다른 스레드가 쓰고 있는 메모리를 읽는 것 아닌가” 가 가장 먼저 의심스럽다. 그렇지 않고, 근거는 운이 아니라 **코드 순서**다.



MVCC 리더가 진입하는 유일한 입구는 heap 의 `prev_version_lsa` 이고, 그 값이 찍히는 것은 attach 가 반환된 뒤다. 부분적으로 쓰인 노드를 볼 창이 없다 — 그러므로 필요한 것은 **동기화가 아니라 수명 관리**다.

#	불변식	깨지면
I1	LSA 는 단조 증가하며 절대 되풀이되지 않는다	슬롯 재사용 판별이 무너져 남의 노드를 읽는다
I2	attach 반환 뒤 MVCC 리더가 읽는 필드 에 아무도 쓰지 않는다	반쯤 쓰인 undo 이미지를 읽는다
I3	<code>copied_lsa ≤ append_lsa</code> — 앞서지 않는다	필요한 drain 을 건너뛰어 없는 데이터를 읽는다
I4	retire 순서 = register 순서 (대상은 항상 링의 head)	남의 슬롯을 풀고, 찾을 수 있는 노드를 해제한다
I5	finalize 시점에 pin 보유 MVCC 리더가 없다	MVCC 리더가 사라진 테이블에서 디스크립터를 얻는다

I1-I4 는 코드 몇 개를 같이 읽으면 이 자리에서 닫힌다. I5 만 코드 밖이다 — finalize 는 `LOG_CS` 쓰기 모드에서 돌지만, **윈도를 읽는 리더는 `LOG_CS` 를 잡지 않는다** (그게 1층의 요점이다). 그래서 락이 그 리더를 배제해 주지 못한다 — 코드 주석도 “이미 pin 을 든 리더는 커버되지 않는다” 고 적는다. I5 는 “그 시점에 클라이언트 트랜잭션이 없다” 는 **운영 사실**에 기댄다.

왜 더 간단한 방법이 아니었나

거르는 기준은 셋이다 — **P-1** 끝이 보장된 대기, **P-2** 리더 경로에서 전역 직렬점 제거, **P-3** LSA 전체 순서에 새 제약을 더하지 않기.

후보	판정
S0 부분 drain — 필요한 레코드를 가진 부분만 밀어 넣는다	기각 — MVCC 리더가 여전히 삽입 경로 동기화에 개입 (P-2 미통과)
S1 복사 진행 watermark 대기	기각 — 전역 연속값이라 무관한 트랜잭션의 미복사 구간까지 함께 기다린다
S2 staging 중인 노드를 직접 읽는다	채택
S3 undo 이미지 별도 캐시	보류 — 갱신마다 복사가 하나 늘고 캐시 크기 산정 문제. S2 의 대안으로 남김
S5 <code>append_lsa</code> 자체를 atomic 으로	기각 — 뜻이 다르다. 그 값은 레코드 중간을 가리킬 수 있다

결정적 칸은 **현행 단독 가치** 였다 — S1 은 writer 측 병렬화가 들어와야 의미가 생기지만, S2 는 지금 코드에서 그 자체로 `LOG_CS` 를 제거한다.

1부 정산 — 무엇이 닫혔고 무엇이 남았나

1부에서 닫은 것

- ☐ 설계에 동의하는가 — `copied_lsa` 의 한쪽 방향 안전성, 3층, 완성 시점 논증
- ☐ 불변식 I1-I4 — 여기서 닫혔는가, 아니면 더 볼 코드가 있는가
- ☐ 기각된 대안 넷 — 더 간단한 길이 없다는 데 동의하는가

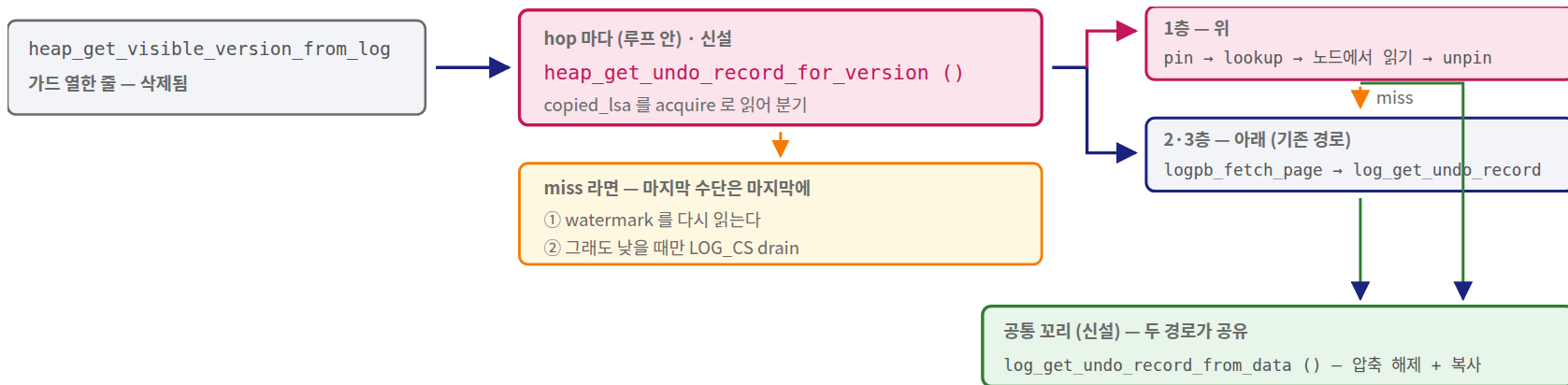
2부에서 판정할 것

- ☐ 코드가 그 설계대로인가 — 리더 경로 · 링 · 수명 · 카운터
- ☐ 시나리오 4 · 5 — 락 없는 안전성이 사실상 이 둘에 걸려 있다
- ☐ 미검증 분류와 합의 항목 — 머지 조건을 정한다

코드 밖으로 나간 것을 여기 적어 둔다 — I5 는 “그 시점에 클라이언트 트랜잭션이 없다” 는 운영 사실에 기댄다. 1부에서 못 닫은 질문도 이 자리에 남기고 2부를 연다. · 여기서 쉰다 — 2부는 이 장부터 다시 연다.

판정이 루프 안으로 들어왔다

가드 열한 줄이 사라지고, hop 마다 신설 함수 `heap_get_undo_record_for_version ()` 이 `copied_lsa` 를 acquire 로 읽어 분기한다. miss 면 먼저 watermark 를 **다시 읽고**, 그래도 낮을 때만 `LOG_CS` drain 이다.



새 경로가 기존 경로를 대체하지 않는다 — 기존 경로는 그대로 남고 그 앞에 층 판정 한 겹이 붙는다. 압축 해제와 복사는 신설된 **공통 꼬리** 하나를 두 경로가 공유하므로 반환값 의미가 같릴 자리가 없다.

리더 경로 — 코드에 어떻게 반영됐나

PR 에서 찾아가기 — `src/storage/heap_file.c` → `heap_get_undo_record_for_version ()` · `heap_get_visible_version_from_log ()`

`src/storage/heap_file.c` — 가드 열한 줄이 사라진 자리

```
- oldest_prior_lsa = *log_get_append_lsa (); /* TODO: fix atomicity issue on x86 */
- if (LSA_LT (&oldest_prior_lsa, previous_version_lsa))
- { LOG_CS_ENTER (thread_p); logpb_prior_lsa_append_all_list (thread_p); LOG_CS_EXIT (thread_p); }
```

... 그 자리에 `hop` 마다 부르는 신설 함수가 들어왔다

```
+ LOG_LSA copied_lsa = log_Gl.append.get_copied_lsa ();
+ if (LSA_LE (&copied_lsa, version_lsa))
+ {
+   if (log_get_undo_record_from_inflight (thread_p, version_lsa, recdes, &window_scan))
+   { perfmon_inc_stat (thread_p, PSTAT_LOG_INFLIGHT_WINDOW_HIT); return window_scan; }
+   perfmon_inc_stat (thread_p, PSTAT_LOG_INFLIGHT_WINDOW_MISS);
+   copied_lsa = log_Gl.append.get_copied_lsa (); /* ← 재확인 */
+   if (LSA_LE (&copied_lsa, version_lsa)) { LOG_CS_ENTER ... } /* 마지막 수단 */
```

삭제된 줄과 신설 함수가 **같은 파일**이다 — 지운 판정이 어디로 옮겨갔는지 한 화면에서 대조된다. `LOG_CS_ENTER` 가 **가장 안쪽 분기**로 밀려난 것이 이 diff 의 요점이다.

윈도가 못 줄 때 — 미스 사유 다섯

22번 코드의 두 번째 `if (LSA_LE (&copied_lsa, version_lsa))` 에 도달하는 경우다. 조회가 빈손이었다고 곧바로 `LOG_CS` 로 가지 않고 **watermark** 를 한 번 더 읽는 이유가 여기 있다.

#	미스 사유	언제 생기는가	재확인이 거르나	그래서
1	등록 대상이 아닌 레코드	<code>udata</code> 가 없거나 MVCC 세 타입이 아니다 — 애초에 실리지 않는다	×	강제 drain
2	<code>attach</code> 때 링이 찼다	flush 가 밀려 링이 가득 — 등록을 건너뛴다 (holder OOM 도 같게 degrade)	×	강제 drain
3	스캔 상한보다 깊다	노드는 링에 있는데 <code>4Ki</code> 밖이라 포기했다	×	강제 drain
4	<code>pin</code> 할 것이 없다	윈도가 내려갔거나 복구-standalone 모드 — 조회조차 안 한다	×	강제 drain
5	방금 복사돼 슬롯이 반납됐다	판정과 조회 사이에 drain 이 지나갔다	✓	2·3층

다섯 중 하나만 걸러진다. drain 이 `copied_lsa` 를 retire 보다 먼저 올리므로, 슬롯이 사라졌다면 watermark 는 이미 올라가 있다 — 재확인 없이 갔으면 헛일을 `LOG_CS` 안에서 했다. 나머지 넷은 실제로 아직 안 복사된 것이라 옛 경로로 간다. 이 분기가 최악의 경계이고, 그 최악이 AS-IS 와 같다.

윈도는 **노드를 복사하지 않는다**. 최근 등록된 노드를 LSA 로 찾는 **고정 크기 색인**이고, 슬롯이 가리키는 곳에 리더가 원하는 바이트가 그대로 있다.

슬롯 한 칸 = 16 B



회수되면 NULL_LSA 로 되돌린다 — 빈 칸 표시가 곧 키다

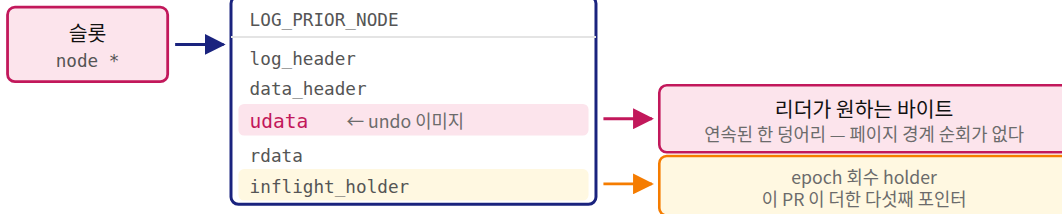
64Ki (= 65,536) 칸 × 16 B = 1 MiB 고정 · 정적 배열

인덱스 = seq & (2¹⁶ - 1) — 마스크 한 번, 랩어라운드 처리가 없다

8 B 키가 성립하는 근거:

LOG_LSA = pageid:48 + offset:16 → 정확히 8 B

포인터 사슬 — 리더가 원하는 바이트까지



크기 관계 — 윈도우는 **prior list** 의 꼬리만 덮는다

■ 링 1 MiB

prior list — 기본 64M~256M

그래서 flush 가 밀리면 링이 먼저 찬다. 그때 등록이 멈추고 리더는 옛 경로로 떨어진다.

키를 **start_lsa** 하나로 쓸 수 있는 것이 **설계 전체의 전제**다 — 8바이트라 **std::atomic<LOG_LSA>** 가 lock-free 이고, 그래서 리더가 락 없이 키를 읽고 노드 포인터의 유효성까지 같은 값으로 판별한다 (27번 발행 규약).

PR에서 찾아가기 — `src/transaction/log_prior_inflight_ring.hpp` · `log_append.hpp`

`log_prior_inflight_ring.hpp` — 슬롯 한 칸

```
+ struct slot
+ {
+   std::atomic<LOG_LSA> start_lsa;      /* the key, and the publication flag - NULL_LSA once unlinked */
+   std::atomic<log_prior_node *> node; /* ordered by start_lsa above, so relaxed access is enough */
+ };
+ slot &slot_at (std::uint64_t sequence) { return m_slots[sequence & (CAPACITY - 1)]; }
```

`log_append.hpp` — `LOG_PRIOR_NODE` 에 더한 다섯째 포인터

```
+ /* Non-NULL while registered in the in-flight window. Holds the node for epoch
+  * reclamation, and tells the drain to retire it instead of freeing it. */
+ log_prior_inflight_holder *inflight_holder;
```

`slot_at` 의 `& (CAPACITY - 1)` 이 마스크 인덱스이고, 그래서 램어라운드 코드가 없다. 노드 구조체가 바뀌었다는 사실은 `log_append.hpp` 쪽 hunk 에서만 보인다 — **따로 짚지 않으면 놓친다.**

register

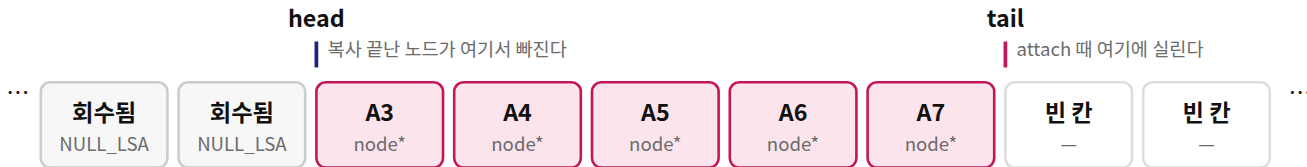
append 경로, `prior_lsa_mutex` 안 단일 **writer**. `tail` 만 전진시키므로 LSA 순서가 그대로 링 순서가 된다.

retire

drain 루프가 **노드 하나를 페이지로 복사한 직후**, `LOG_CS` 안에서 그 노드를 뺀다 — `copied_lsa` 를 올린 **다음**이다. **head** 는 이때 **전진한다**. 등록 순서와 같아 대상은 항상 head 이고, 아니면 fatal 이다.

pin_lookup

리더 경로, **락 없는 wait-free 스캔**. 최신 슬롯부터 거꾸로 훑되 **4Ki** 슬롯까지만 보고 포기한다.



발행 규약 — 쓰는 순서와 읽는 순서가 반대다

쓰는 쪽 (register) ① node 를 relaxed 로 먼저 → ② start_lsa 를 release 로 — 이것이 발행 → ③ tail 을 release 로 올려 스캔 범위에 편입

읽는 쪽 (pin_lookup) start_lsa 를 acquire 로 먼저 읽는다 — 그래야 node 포인터가 유효하다

생성 카운터가 없는 이유 — LSA 는 되풀이되지 않으므로, 노드 포인터를 집은 뒤 `start_lsa` 를 한 번 더 읽어 같으면 그 포인터는 그 LSA 의 것이다.

PR에서 찾아가기 — `src/transaction/log_prior_inflight_ring.hpp` → `push ()` · `pop_if_head ()` · `find ()`

push (): 쓰는 쪽 — attach 때, `prior_lsa_mutex` 안. `tail` 이 여기서 전진한다

```
+ s.node.store (node, std::memory_order_relaxed);
+ s.start_lsa.store (start_lsa, std::memory_order_release); /* publishes the slot */
+ m_tail.store (tail + 1, std::memory_order_release);      /* ... and admits it to the scan range */
```

pop_if_head (): 빼는 쪽 — 복사 직후, `LOG_CS` 안. `head` 가 여기서 전진한다

```
+ if (s.node.load (std::memory_order_relaxed) != node) return false; /* head 가 그 노드가 아니면 안 뺀다 */
+ /* Unlink before the owner retires the node: a later reader cannot reach it, and one already
+  * holding it is the owner's problem, not the ring's. */
+ s.start_lsa.store (NULL_LSA, std::memory_order_release); /* ← 묘비. 키를 죽여 조회에서 지운다 */
+ m_head.store (head + 1, std::memory_order_release);      /* ← 그 다음에 head 전진 */
```

find (): 읽는 쪽 — 키를 먼저, 최신부터 거꾸로

```
+ if (lsa.is_null ()) return NULL; /* NULL_LSA is the unlink marker, not a key */
+ LOG_LSA slot_lsa = s.start_lsa.load (std::memory_order_acquire);
```

순서가 세 겹으로 걸려 있다 — ① 묘비가 `head` 전진보다 먼저 (뒤집히면 `head` 가 지나간 슬롯에 살아 있는 키가 남는다) · ② **unlink** 가 **retire** 보다 먼저 (이후 도착하는 리더는 도달 자체가 불가능해진다) · ③ 둘 다 `release` 라 리더의 `acquire` 와 짝을 이룬다. 이미 집은 리더는 **pin 이 맞는다** — 링의 묶이 아니다.

epoch reclamation — 다섯 줄

- ▶ MVCC 리더는 읽기 전에 현재 전역 id 를 적고(**pin**), 다 읽으면 지운다(**unpin**).
- ▶ 노드를 버릴 때는 즉시 해제하지 않고 그때의 id 를 찍어 대기열에 넣는다(**retire**).
- ▶ 해제는 **최소 활성 id** 보다 작은 노드에만 일어난다.
- ▶ **pin 구간**(`start_tran` ~ `end_tran`)이 열려 있는 동안 해제가 이뤄진다 — 노드가 아니라 **시점**을 잡는다. unlink 와 회수 스캔 사이의 **seq_cst** **울타리**가 그 짝이다 (29번).
- ▶ drain 순서 — ① 페이지 복사 → ② `copied_lsa` 를 먼저 발행 → ③ 링에서 빼고 회수로.

읽는 법이 정해져 있다 — 정상 상태는 `hit` 이 크고 `reader_guard` 가 0 이다. 0 이 아니게 되는 순간이 이 설계가 무너지는 신호이고, 그때 나머지 둘이 사유를 말해 준다.

카운터 다섯

- ▶ `window_hit` / `window_miss` — 원한 버전이 아직 staging 이 있는가.
- ▶ `drain_reader_guard` — **MVCC 리더가 유발한** drain 의 횟수와 시간.
- ▶ `drain_flush` — 직접 flush 안의 drain.
- ▶ `list_maxed_wait` — 생산자가 상한에 닿아 1 ms 잠든 시간.

노드 수명 — pin 구간이 무엇을 지키는가

PR 에서 찾아가기 — `log_manager.c` → `log_get_undo_record_from_inflight ()` · `log_prior_inflight.cpp` → `pin_lookup ()` · `retire ()`

log_manager.c — 리더가 pin 을 여닫는 전 구간은 이 세 줄뿐이다

```
+ LOG_PRIOR_NODE *node = log_prior_inflight_pin_lookup (thread_p, *lsa, pin);
+ *scan_out = log_get_undo_record_from_node (thread_p, node, recdes);    /* ← 이 안에서만 읽는다 */
+ log_prior_inflight_unpin (pin);
```

log_prior_inflight.cpp — pin 이 슬롯 읽기보다 반드시 앞선다

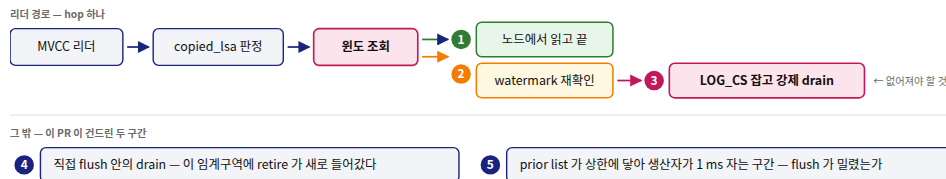
```
+ /* Pin before reading any slot: a node retired from here on cannot be freed until this thread
+  * leaves, because the drain tags what it retires with a later transaction id. */
+ tdes->start_tran ();                /* ← 여기부터 '나는 시점 g 를 본다' */
+ LOG_PRIOR_NODE *node = log_Inflight_ring.find (lsa);
+ if (node == NULL) { tdes->end_tran (); return NULL; }    /* miss 면 즉시 닫는다 */
```

retire 쪽 — 이 구간이 열려 있는 동안 해제가 미뤄진다

```
+ std::atomic_thread_fence (std::memory_order_seq_cst);
+ tdes->retire_node (*holder);    /* 최소 활성 id 보다 작은 노드만 실제로 해제된다 */
```

구간이 하는 일은 하나다 — 이 구간이 열려 있는 동안 해제가 미뤄진다. 노드 하나를 붙잡는 것이 아니라 **시점**을 선언하는 것이라, 이 시점 이후 retire 되는 모든 노드가 함께 산다. unlink 는 막지 않는다 — 링에서 빠져 뒤에 오는 리더는 못 찾지만, 이미 포인터를 든 이 리더는 계속 읽는다. 그래서 **구간은 짧아야 한다**: 길면 최소 활성 id 가 낮게 묶여 회수 대기열이 자란다 (I5 도 이 구간이 닫혀 있어야 성립한다).

카운터 다섯 — 어디를 재고 무엇을 뜻하는가



#	이름 · 종류	재는 구간	무엇을 답하는가
1	<code>Num_log_inflight_window_hit</code> 누적	원도 조회가 노드를 찾은 지점	이 설계가 일하고 있는가
2	<code>Num_log_inflight_window_miss</code> 누적	조회가 빈손으로 돌아온 지점	아니었는가 — 사유 다섯을 합친 값 (23번)
3	<code>log_prior_drain_reader_guard</code> counter+timer	LOG_CS 진입~해제 전 구간	없애려던 바로 그 비용이 남아 있는가
4	<code>log_prior_drain_flush</code> counter+timer	직접 flush 안의 drain	그 임계구역에 retire 가 들어갔다 — 무거워지지 않았는가
5	<code>log_prior_list_maxed_wait</code> counter+timer	상한에서 생산자가 1 ms 자는 구간	flush 가 밀렸는가 — 링이 차는 원인 쪽

읽는 법 — 정상은 1 이 크고 3 이 0 이다. 3 이 0 이 아니게 되는 순간이 이 설계가 무너지는 신호이고, 그때 2 와 5 가 사유를 말해 준다.

기존 진단 카운터는 남기지 않았다 — M0 단계의 mutex·LOG_CS 대기과 drain 호출자별 분해, retired-backlog 게이지는 진단이 끝나 정리했고
`_flush_daemon` / `_flush_direct` 둘은 4 하나로 합쳤다.

PR에서 찾아가기 — `src/base/perf_monitor.c` · `perf_monitor.h` (`PSTAT_* enum`)

`perf_monitor.c` — `statdump`에 나가는 이름이 여기서 정해진다

```
+ /* prior list, charged to whoever paid for it */
+ PSTAT_METADATA_INIT_COUNTER_TIMER (PSTAT_LOG_PRIOR_LIST_MAXED_WAIT,  "log_prior_list_maxed_wait"),
+ PSTAT_METADATA_INIT_COUNTER_TIMER (PSTAT_LOG_PRIOR_DRAIN_FLUSH,      "log_prior_drain_flush"),
+ PSTAT_METADATA_INIT_COUNTER_TIMER (PSTAT_LOG_PRIOR_DRAIN_READER_GUARD, "log_prior_drain_reader_guard"),
+
+ /* in-flight window: whether a reader found the version it wanted still staged */
+ PSTAT_METADATA_INIT_SINGLE_ACC (PSTAT_LOG_INFLIGHT_WINDOW_HIT,  "Num_log_inflight_window_hit"),
+ PSTAT_METADATA_INIT_SINGLE_ACC (PSTAT_LOG_INFLIGHT_WINDOW_MISS, "Num_log_inflight_window_miss"),
```

이 문자열이 곧 인터페이스다 — 한 번 나가면 모니터링이 그 이름에 붙는다. 35번 합의 항목의 “이름을 지금 확정할 것인가”가 가리키는 줄이 이것이다.

안전성을 어디서 판정하는가

#	상황	무엇을 확인하는가
1	원도 히트	설계가 노리는 정상 경로 — LOG_CS · 뮤텍스 · I/O · 대기 하나도 없다
2	판정 후 drain 이 지나감	재확인이 왜 필수인지 — 가장 흔한 미스 사유를 락 없이 걸러낸다
3	진짜 미스	최악이 AS-IS 와 같다는 것 — 못 쓰이면 옛 경로로 떨어진다
4	슬롯 재사용	생성 카운터 없이 판별하는 방법 — 키를 두 번 읽는다
5	읽는 중 retire	노드가 발밑에서 사라지지 않는 이유 — 두 규율이 전부다
6	원도를 쓸 수 없음	폴백 방향이 늘 한쪽이라는 것
7	사슬이 세 층에 걸침	판정이 hop 마다 있어야 하는 이유

락 없는 설계의 안전성은 사실상 4 와 5 에 걸려 있다. 리뷰에서 이 둘을 먼저 본다.

회수가 영구히 멈춘다 (CBRD-27044)

`INVALID_TRANID` 는 sentinel 이면서 `uint64` 최댓값이다. 이 겹침이 두 가지 실패를 만들었다.

① 한순간 아무도 쓰지 않음

- 모든 디스크립터의 `m_tranid` 가 `INVALID`
- 최소값 계산 결과 = `INVALID` (= `uint64` 최댓값)
- 그 값이 캐시된다

② 다음 회수 패스

- 최솟값이 최댓값이므로 대기열 전부 회수
- — 여기까지는 맞는 동작이다
- 그리고 그 최댓값을 회수 기준선으로 기록한다

③ 이후 모든 패스

- `min_tran_id` ≤ 최댓값은 항상 참
- 첫 줄에서 `return` 한다
- 다시는 회수하지 않는다

관측된 모습 — retire 30,411 대 reclaim 24

retire 는 계속되고 회수만 멈춘다. 메모리 사용이 계속 자란다.

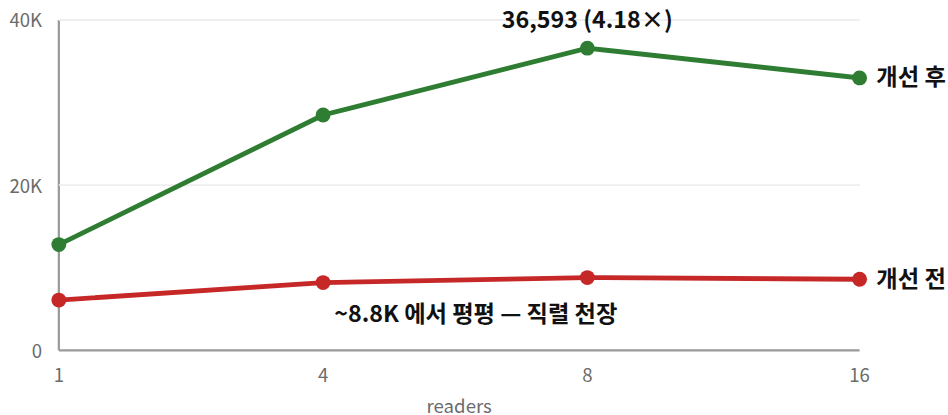
왜 지금까지 안 드러났는가

기존 사용처는 retire 량이 적고, 테이블이 완전히 비는 순간이 드물다.

관측값은 **retire 30,411 대 reclaim 24**. 수정은 같은 함수 두 줄이다 — 들어올 때 sentinel 이면 **재계산**, 나갈 때 sentinel 은 **기록하지 않음**. 커밋이 자족적이라 분리할 수 있지만, **분리하면 CBRD-27044 가 먼저** 들어가야 한다.

강제 drain 0, 읽기 천장 4.6x

성능은 이 리뷰의 주제가 아니지만 설계 판단의 근거이므로 결론만 본다. 개선 전은 리더를 늘려도 8.8K 근처에서 **평평하다** — 전역 임계구역이 만든 직렬 천장이다.



카운터 전환 (readers = 8)

강제 drain **10,940 → 0**원도 hit **17,578**

두 수가 같지는 않다 — 셀마다 강제 drain 이 0 이 된다

쓰기는 느려지지 않는다

1,195~1,232 → 1,278~1,303 TPS

YCSB 는 양방향으로 평평 (-1.4% / +6.6%)

읽기 **8,346 → 82,650 QPS (워터마크 수정 단독 A/B 2.05x)**, 쓰기는 느려지지 않는다. 강제 drain **~10,940 → 0**. 다만 hot 행 넷에 연산마다 커밋하는 **합성 최악 부하**이다. YCSB 처리량은 같은 세션 develop 대조가 없어 미확정이며, 원도 조회 빈도는 행 수를 늘리면 얇아진다. 곡선은 증분 단계 빌드의 A/B 스왑이라 8r 한 점의 재측정과 숫자가 다르며, 판정은 같다.

미검증 다섯 — 넷은 다했다

완료 — 기존 MVCC·스냅샷·vacuum·복구 스위트 ✓ CI 회귀 통과, **이 리뷰**의 전제

실행 완료 — ① PR의 링 유닛 테스트 (`test_inflight_ring.cpp`)가 **자료구조 수준**에서 링 full·스캔 상한 경계·동시 push/pop/find를 덮고, ② 상수를 낮춘 빌드 + **버스트 갱신** 부하가 **리더 폴백**을 실제로 태웠다

arm	용량 / 상한	강제 drain	값 손상
대조군	64Ki / 4Ki	0	0
링 full	64 / 64	96	0
스캔 상한	64Ki / 8	71	0

SA 모드(디스크립터 없는 해제) — `csql rc 0, checkdb -S rc 0`, 값 정확

남은 하나 — 값 수준 일치 검증 → 스냅샷별 기대값 계산이 필요한 별도 TC PR

합의만 필요한 항목

- ▶ 링 용량 64Ki (1 MiB 정적) — 파라미터로 뺄 것인가.
- ▶ 스캔 상한 4Ki — 측정값이 2,000–2,500 이라 **여유가 2배 미만**이다.
- ▶ 카운터 다섯의 이름 — `statdump` 이름은 사실상 인터페이스다.
- ▶ 순서 위반 시 fatal — 서버를 내리는 것이 옳은가.
- ▶ CBRD-27044 를 같은 PR로 둘 것인가.

초안의 재현 처방이 틀렸다 — 링 점유를 정하는 것은 용량이 아니라 **drain 빈도**다. 용량을 **1024배** 줄여도 강제 drain 은 0 이었고, 커밋마다 flush 가 도는 부하에서 drain 당 레코드가 약 3개뿐이다. 채우려면 **커밋을 미뤄 flush 를 굵겨야** 한다.

Thank you

Q & A

- ▶ PR 7580 — CBRD-27048 리더 경로 · CBRD-27044 회수 스톱
- ▶ 층 판정 진입점 — `src/storage/heap_file.c`
- ▶ 원도 — `src/transaction/log_prior_inflight.cpp`