Checkpoint, Vacuum & Log Reclamation in CUBRID

Why archive logs stop being purged — and why it's checkpoint, not vacuum

2026-06 · hgryoo · Code Analysis Seminar · CBRD-26957

Agenda

0. **The problem** — archives grow to disk-full; two clues point **away** from vacuum
1. **Theory** — the three retention horizons (REDO / UNDO / GC)
2. **How CUBRID reclaims a log** — checkpoint · vacuum · who deletes · the MIN (concept + code each)
3. **The incident (CBRD-26957)** — a hot page starves the checkpoint's flush
4. **Why checkpoint froze but vacuum didn't** — the sync-vs-async asymmetry, the fix

> Source: `NOTES_checkpoint_flush_log_reclamation.md`. Code anchors @ `cubrid 5fd7b76c1` (11.5).

The problem — archives never get purged

QA loads 140 tables concurrently (sysbench, 140 connections, autocommit). Archive logs are **created but never deleted** — `lgar000 … lgar454` in 90 min, **0 removed** — until the disk fills.

- **Clue 1** — every purge attempt fails the same way: `last_arv_num_to_delete is -1`, and `Checkpoint finished` count = **0**.
- **Clue 2** — yet `cubrid vacuumdb` shows `keep_from_log_pageid` climbing **0 → head** the whole time: vacuum is demonstrably **keeping up**.
- Two clues, one verdict: it is **not** vacuum falling behind. The stall is one layer over — in the **checkpoint**. The rest of the talk shows why.

Theory — when may a WAL log record be discarded?

A log record may be dropped only when **no consumer still needs it**. Textbook framing lists two consumers (crash recovery, vacuum); the code splits crash recovery in two, so there are **three** independent horizons.

Horizon	Axis	Held by	Advances when
REDO	LSA	oldest **unflushed** dirty page	a checkpoint flushes that page
UNDO	LSA	oldest **active** transaction	that transaction ends
GC / vacuum	MVCCID	oldest **visible** snapshot	snapshot releases + block vacuumed

- Two horizons live on the **LSA axis** (physical log position), one on the **MVCCID axis** (logical version time).
- The log may be truncated only up to the **MIN** of all three — whichever consumer is furthest behind wins.

How CUBRID reclaims a log

checkpoint · vacuum · who deletes · the MIN — concept + code

CUBRID

Fuzzy checkpoint — pins REDO, records UNDO

```
// logpb_checkpoint — log_page_buffer.c:6877 (fuzzy)
LOG_CS_ENTER (thread_p);                  // :6916
prior_lsa_alloc_and_copy_data (LOG_START_CHKPT);
LOG_CS_EXIT (thread_p);     // :6978 ← BEFORE flush
pgbuf_flush_checkpoint (...);  // :6986 (no LOG_CS)
// on completion → advance
//   last_arv_num_for_syscrashes          // :7300
```

- **Fuzzy = no quiescing**: pages flush while transactions keep writing; the oldest-still-dirty LSA becomes the redo start.
- The log critical section is dropped **before** the flush (`:6978`).
- Job: advance **REDO** + capture **UNDO** (active-tran snapshot). It never touches the GC horizon.

The redo point — and the record holds no MVCC

```
// pgbuf_flush_seq_list — page_buffer.c:4370 (redo)
if (LSA_LT (&bcb->oldest_unflush_lsa,
            chkpt_smallest_lsa))
    LSA_COPY (chkpt_smallest_lsa,
       &bcb->oldest_unflush_lsa);  // → chkpt_lsa :7187
// log_rec_chkpt — log_record.hpp:345 (NO MVCC)
struct log_rec_chkpt {
    LOG_LSA redo_lsa; int ntrans; int ntops; };
```

- Redo LSA = smallest `oldest_unflush_lsa` the checkpoint couldn't flush → becomes `chkpt_lsa`.
- The record holds only LSA-axis facts — **no MVCC**. Vacuum's horizon rides in the log header (`prior_update_header_mvcc_info`, `log_append.cpp:1326`), merely **flushed** here.
- **No completion → `chkpt_lsa` / syscrash never advance** — hold for Part 3.

CUBRID

MVCC vacuum — advances the GC horizon

```
// vacuum_master_task::execute — vacuum.c:3001
oldest_visible =
    update_global_oldest_visible ();      // :3027
for (block = first_unvacuumed; valid; block = next) {
    if (block.newest_mvccid >= oldest_visible)
        break;              // :3105 not safe → stop
    dispatch_worker (block);   // reclaim dead vers.
}
keep_from_log_pageid =
    first_page_of (first_unvac_block); // :5782 floor
```

- Reads old log **blocks** in MVCCID order; reclaims versions older than **oldest-visible**.
- Gate: a block runs only when `newest_mvccid < oldest_visible`.
- `keep_from_log_pageid` = first unvacuumed block (the archive floor).
- A long txn holding oldest-visible low stalls **this** horizon — but the workload was autocommit.

Who actually deletes an archive? (not vacuum)

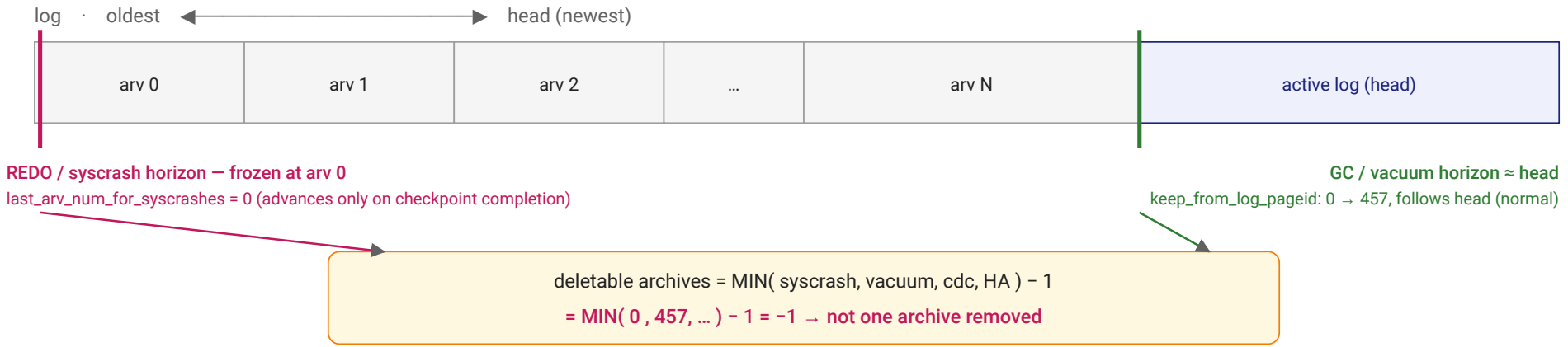
Neither the vacuum worker nor a flusher deletes archive files. Vacuum and checkpoint only **publish horizons**; the **log manager** performs the delete, gated by their MIN.

Actor	Role	Code
vacuum master / worker	advances `keep_from_log_pageid` (GC) — deletes nothing	`query/vacuum.c:5782`
checkpoint	advances `last_arv_num_for_syscrashes` (syscrash)	`log_page_buffer.c:7300`
log manager	the actual file delete, gated by `MIN(…)`	`logpb_remove_archive_logs_exceed_limit:5990`

- Triggered by log-side events — archiving the active log (`logpb_archive_active_log:5673`) or the remove-archive daemon (`log_manager.c:10270`), **not** a vacuum tick.
- Checkpoint does **not** call the deleter; it only **raises** the syscrash clamp the deleter reads. So both horizons are published, the log manager removes.

Where the horizons meet — truncation = MIN

Projected onto the log, the horizons are **positions**; the deletable bound is the **leftmost (oldest)** — their MIN.




```
// logpb_remove_archive_logs_exceed_limit — src/transaction/log_page_buffer.c:5990
last = MIN (nxarv-n, syscrash /*:6079*/, vacuum /*:6089*/, cdc, ha) - 1;
// syscrash frozen at 0  ⇒  MIN(0, …) - 1 = -1  ⇒  zero archives deleted
```

The incident — CBRD-26957

a hot page starves the checkpoint's flush

CUBRID

The hot page — db_serial under AUTO_INCREMENT

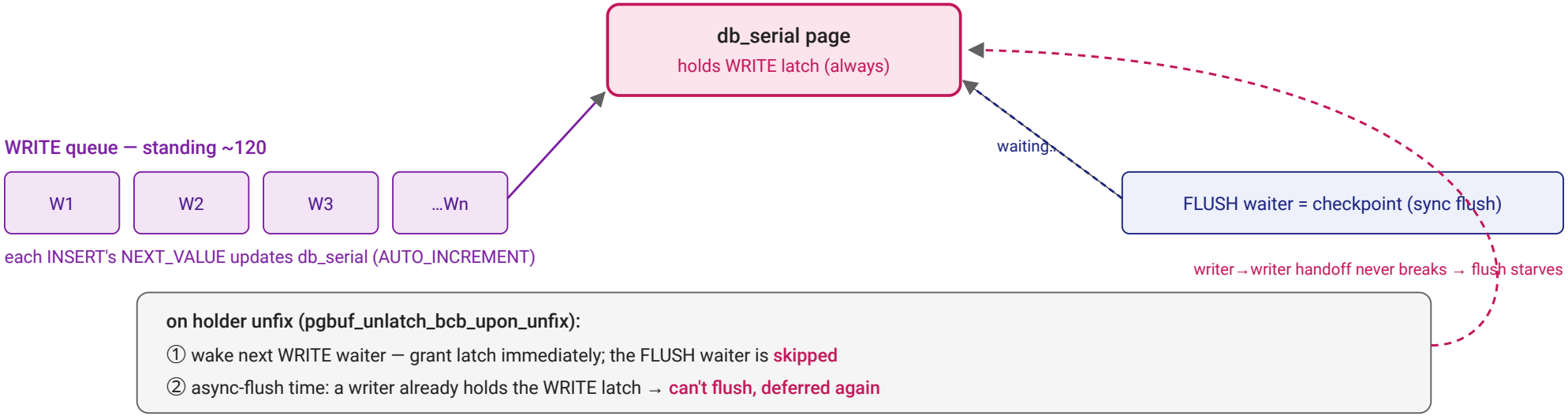
```
checkpoint thread (pstack, ×11 snapshots):
  pgbuf_block_bcb(<db_serial pg>,
              PGBUF_LATCH_FLUSH)  ← waits
   ← pgbuf_bcb_safe_flush_internal(sync=true)
   ← pgbuf_flush_checkpoint
   ← logpb_checkpoint
~120 workers:
  xserial_get_next_value
   → qexec_execute_insert (write-latch)
```

- All 140 tables' AUTO_INCREMENT values sit on **one or two `db_serial` pages**; `NEXT_VALUE` write-latches one → a **standing queue** (~120). AUTO_INCREMENT can't cache.
- The page is **always** write-latched → the checkpoint's synchronous flush never acquires it.
- Not a deadlock — a **starvation**: the holder just rotates forever.

CUBRID

Checkpoint flush starvation

Synchronous flush blocks on the write-latched page and waits for a handoff that, under a standing queue, never comes:



- So `Checkpoint finished = 0` → syscrash frozen at 0 → `MIN − 1 = −1` → disk full. The pinned MIN term is syscrash `:6079`, **not** vacuum `:6089`.
- Starvation, not deadlock: ease the contention and the checkpoint completes, the horizon jumps to head, the backlog is reclaimed at once.

Why checkpoint froze but vacuum didn't

Same hot-page contention, opposite fate — the difference is **how each writes its pages**.

	Checkpoint	Vacuum
Flush mode	**synchronous** `pgbuf_flush_checkpoint`	**non-blocking** `pgbuf_flush_if_requested`
On a write-latched page	blocks (`PGBUF_LATCH_FLUSH`)	skips — flushes only if asked, page held latched
Incident outcome	starves → never completes	follows the log head normally

- Measured: vacuum's `keep_from_log_pageid` advanced **0 → head** (`cubrid vacuumdb`); the checkpoint's syscrash horizon stayed **0**.
- So the MIN was pinned by checkpoint, and vacuum was a **red herring**. The deep reason the two diverge under identical contention is the sync-vs-async flush asymmetry.

The fix

Engine (root cause) — Option A, flush fairness. Make the synchronous-flush handoff race-free: flush the page **before** releasing the latch on unfix, so the checkpoint's per-page wait always completes in bounded time.

- `pgbuf_unlatch_bcb_upon_unfix` — flush while still holding the latch; +17 lines, verified (checkpoint completes, archives purge).
- Complement (long-term): non-blocking checkpoint flush with bounded revisit (Option D) — reduces how often the checkpoint relies on the per-page wait at all.

Workload (mitigation). AUTO_INCREMENT cannot use a serial cache, so the user levers are: reduce concurrency / table count (verified workaround), or switch the app to an explicit `CREATE SERIAL … CACHE n` + `NEXT_VALUE`.

CUBRID

Source walkthrough — where to read next

Symbol names are the stable handle; line numbers drift. `git grep -n '<symbol>' src/`.

Topic	Symbol	File
Fuzzy checkpoint	`logpb_checkpoint`	`transaction/log_page_buffer.c:6877`
Redo point	`pgbuf_flush_seq_list`	`storage/page_buffer.c:4370`
Checkpoint record (no MVCC)	`log_rec_chkpt`	`transaction/log_record.hpp:345`
Vacuum gate	`is_cursor_entry_ready_to_vacuum`	`query/vacuum.c:3105`
Archive floor	`vacuum_update_keep_from_log_pageid`	`query/vacuum.c:5782`
Archive remover (the deleter)	`logpb_remove_archive_logs_exceed_limit`	`transaction/log_page_buffer.c:5990`
Recovery resume	`vacuum_notify_server_crashed`	`transaction/log_recovery.c:812`

CUBRID

Thank you

Q & A

- Issue · analysis · fix: **CBRD-26957** (three retention horizons · sync-vs-async flush · flush-fairness Option A)
- Observe: `cubrid vacuumdb` (`keep_from_log_pageid`) · `er_log_vacuum=yes` · `Checkpoint finished` in the server log