CUBRID MVCC

Snapshot Construction, Active-MVCCID Tracking, Vacuum Coordination

2026-05 · Code Analysis Seminar

© 2026 CUBRID Corporation. All rights reserved.

Agenda

0. **The problem** — what an MVCC engine is for
1. **Overview** — one diagram of the whole flow
2. **Theory** — snapshot isolation, write skew, the choice on serializable
3. **Common patterns** — what every SI engine looks like
4. **CUBRID specifics** — `mvcctable`, `mvcc_active_tran`, the bit array
5. **Hot paths** — snapshot build, visibility check, commit, vacuum

> Closing: **Beyond CUBRID** — comparative designs.

© 2026 CUBRID Corporation. All rights reserved.

The problem — what snapshot isolation fixes

Without coordination, concurrent transactions see each other's in-progress work. SI prevents the first four; **write skew** is the residual it admits.

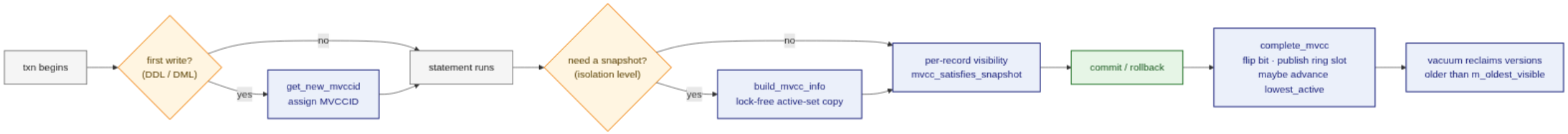
Anomaly	Scenario	What goes wrong
Dirty read	T_1 reads T_2's uncommitted write	T_2 rolls back → T_1 saw a value that "never existed"
Lost update	T_1 and T_2 both read `n`, both write `n+1`	one of the increments is silently overwritten
Non-repeatable read	T_1 reads R twice; T_2 updates R between them	second read disagrees with the first
Phantom	T_1 runs `WHERE color='red'` twice; T_2 inserts a red row	new row appears in the second read
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**: each transaction sees a consistent **snapshot** of the database, while readers never block writers and writers never block readers.
- **MVCC's role**: keep multiple timestamped versions of each row; a **visibility predicate** answers "is this version in your snapshot?" without taking row locks.
- **Snapshot isolation** (SI) is the dominant isolation level built on top of MVCC. CUBRID maps READ COMMITTED, REPEATABLE READ, and SERIALIZABLE onto SI by varying **when** the snapshot is taken.

© 2026 CUBRID Corporation. All rights reserved.

CUBRID

Overview — how an MVCC operation flows



- **Lazy MVCCID issuance.** Read-only transactions never consume an ID.
- **Snapshot acquisition timing is the isolation knob** — same machinery, different cadence.
- **The watermark is the rendezvous between read traffic and vacuum** — one atomic value gates reclamation.

> The rest of the talk zooms into each box, names the data structure behind it, and shows the actual code.

© 2026 CUBRID Corporation. All rights reserved.

MVCC vs 2PL vs OCC

Three families of concurrency control. **Database Internals** ch. 5 frames them by what they push the coordination through.

Family	Coordinates through	Reads block writes?	Writes block reads?	Where the cost lands
2PL (pessimistic)	lock acquisition order	yes	yes	high contention; deadlocks
OCC	commit-time validation	no	no	abort-rate cost; revalidation
MVCC + SI	per-version visibility	**no**	**no**	space (old versions) + vacuum

- The MVCC promise is **non-blocking reads** — every reader walks its own snapshot. That moves coordination off the row latch and onto a tiny per-record check.
- The price is a **bookkeeping debt**: who's active, what's the oldest visible version, when can each dead version be reclaimed. The rest of this deck is that bookkeeping.

> CUBRID layers MVCC for snapshot reads on top of the lock manager for write/write conflicts — the two systems share one language (the MVCCID) and one watermark.

© 2026 CUBRID Corporation. All rights reserved.

Snapshot Isolation and write skew

SI prevents dirty / non-repeatable / phantom / lost-update at **snapshot acquisition time**. The remaining hazard is **write skew**: two transactions read overlapping data, each writes a **different** row, both updates pass their local predicate, the global invariant breaks.

Two production responses, both well-studied:

Approach	Engine(s)	Mechanism	Cost
SSI (Serializable Snapshot Isolation)	PostgreSQL	Predicate locking + dependency-graph cycle detection at commit	extra bookkeeping per read; aborts at commit time
Lock fallback on writes	**CUBRID**, InnoDB	SI for reads; on `SERIALIZABLE`, fall back to lock-based serialization on the write path	the lock manager pays the cost (see companion deck)

- CUBRID's choice is the **lock-fallback** path. The MVCC layer itself does not detect write skew; the lock manager covers `SERIALIZABLE` writes.
- Source verification (open question #3): the exact write path through `lock_object` under `SERIALIZABLE` is owed a separate trace.

> SI by itself is not serializable. Every production MVCC engine pays for that gap somewhere.

© 2026 CUBRID Corporation. All rights reserved.

Five common patterns

Every SI/MVCC engine — PostgreSQL, Oracle, InnoDB, SQL Server, CUBRID — sets the same five dials, just to different values:

1. **Per-record version stamp + visibility predicate.** Minimum `(ins_id, del_id)` plus a prev-version pointer.
2. **Active set representation.** Sorted list, hash, or bit-array-with-overflow. The window size is the central knob.
3. **In-place vs out-of-place old versions.** PostgreSQL keeps prior versions inline on the heap (HOT updates, bloat, scan-everything vacuum). Oracle, InnoDB, **CUBRID** push prior versions into undo / log space.
4. **Snapshot acquisition timing is the isolation knob.** RC = per-statement, RR/SR = once at txn start. Same machinery, different cadence.
5. **Vacuum + watermark.** One atomic "oldest visible MVCCID" — the rendezvous point between live snapshots and the reclamation process.

> CUBRID is **one point** in this dial-space — out-of-place versions, bit-array active set, lazy MVCCIDs.

© 2026 CUBRID Corporation. All rights reserved.

Part II

How does CUBRID turn the dials?

© 2026 CUBRID Corporation. All rights reserved.

Theory ↔ CUBRID mapping

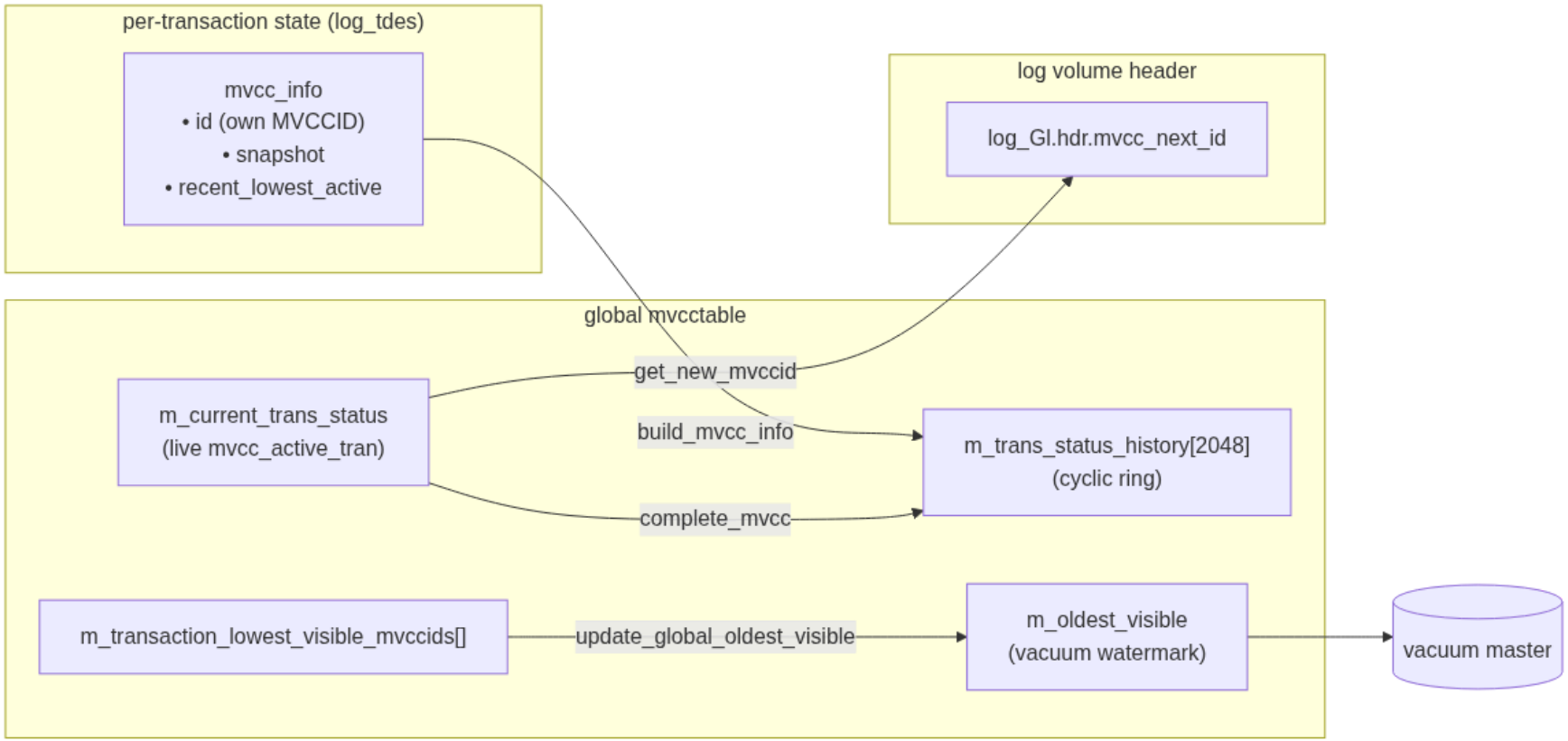
The talk's spine. Every left-column concept maps to a named entity in the source tree.

Theory	CUBRID name
Per-version timestamp	`MVCCID` — 64-bit counter, **lazily** issued on first write
Inserter / deleter stamps in record	`mvcc_rec_header.mvcc_ins_id` , `mvcc_del_id`
Old-version chain (out-of-place)	`mvcc_rec_header.prev_version_lsa` → log-resident copy
"Active at snapshot time" set	`mvcc_active_tran` — bit array + long-tran overflow array
Per-transaction snapshot	`mvcc_snapshot` — active set + low/high MVCCID scalars
Visibility predicate	`mvcc_satisfies_snapshot` — 3-valued result
Global registry	`mvcctable` + `m_trans_status_history[2048]` ring
Oldest visible watermark	`mvcctable::m_oldest_visible` (atomic)

> The next slides walk this table top-to-bottom. Memorize the right-hand column.

© 2026 CUBRID Corporation. All rights reserved.

Overall structure



- **Left**: each in-flight transaction's `mvcc_info` — its MVCCID (if any) and its snapshot.
- **Middle**: the global `mvcctable` — live state, 2048-slot history ring, watermark.
- **Right**: the log volume header owns the MVCCID counter; vacuum master reads the watermark.

© 2026 CUBRID Corporation. All rights reserved.

MVCCID assignment — lazy issuance

```
// mvcctable::get_new_mvccid — src/transaction/mvcc_table.cpp
MVCCID
mvcctable::get_new_mvccid ()
{
  MVCCID id;
  m_new_mvccid_lock.lock ();
  id = log_Gl.hdr.mvcc_next_id;
  MVCCID_FORWARD (log_Gl.hdr.mvcc_next_id);
  m_new_mvccid_lock.unlock ();
  return id;
}
```

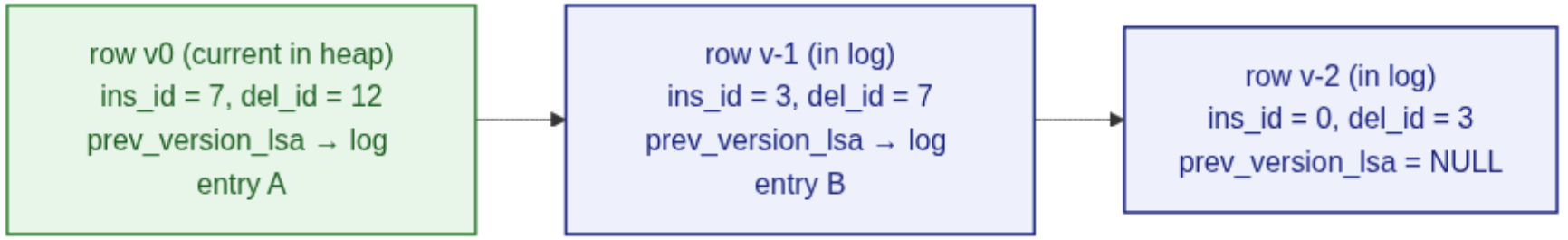
- An MVCCID is allocated **lazily**, on the transaction's first write (DDL or DML). Read-only transactions never get one and never consume active-set capacity.
- Exactly **one** MVCCID per write transaction; sub-transactions get their own.
- The counter itself lives in the active log volume header (`log_Gl.hdr.mvcc_next_id`), not in the MVCC table.
- A dedicated `m_new_mvccid_lock` keeps issuance off the hot `m_active_trans_mutex` path. The `mvcc_table.hpp` comment notes this could in principle become an atomic.

© 2026 CUBRID Corporation. All rights reserved.

CUBRID

Per-record header — `mvcc_rec_header`

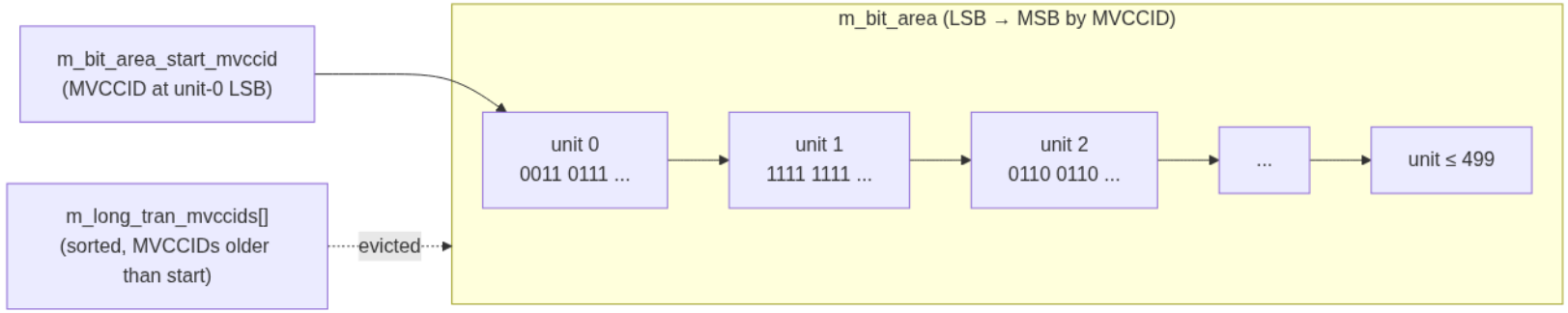
```
// mvcc_rec_header — src/transaction/mvcc.h
struct mvcc_rec_header
{
  INT32 mvcc_flag:8;            /* MVCC flags */
  INT32 repid:24;               /* representation id */
  int chn;                      /* cache coherency number */
  MVCCID mvcc_ins_id;           /* MVCC insert id */
  MVCCID mvcc_del_id;           /* MVCC delete id */
  LOG_LSA prev_version_lsa;     /* log address of previous version */
};
```



Older versions are **out-of-place** — reachable via `prev_version_lsa` chained back through the log. The flag byte controls which optional fields are physically present, so unused slots cost zero bytes.

© 2026 CUBRID Corporation. All rights reserved.

Active-MVCCID tracking — the bit array




```
// mvcc_active_tran (private members) — src/transaction/mvcc_active_tran.hpp
static const size_t BITAREA_MAX_SIZE = 500;    // 500 * 64 = 32k MVCCIDs
static const unit_type ALL_ACTIVE    = 0;
static const unit_type ALL_COMMITTED = (unit_type) -1;
unit_type        *m_bit_area;
volatile MVCCID   m_bit_area_start_mvccid;  /* anchors unit-0 LSB */
volatile size_t   m_bit_area_length;        /* length in bits */
MVCCID           *m_long_tran_mvccids;       /* sorted overflow array */
volatile size_t   m_long_tran_mvccids_length;
```

- **Each bit = one MVCCID.** `0` = active, `1` = completed. Bit `i` → MVCCID `start + i`.
- **32 000 MVCCIDs** (500 × 64) fit in the bit area; older active IDs spill into `m_long_tran_mvccids`.

© 2026 CUBRID Corporation. All rights reserved.

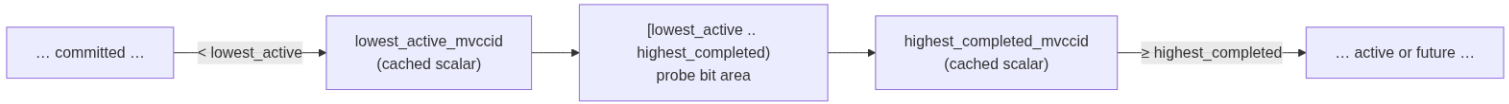
Bit-area lifecycle

When a transaction completes, the bit flips. Two cleanup events run on top:

1. **LTRIM.** If the bit-area prefix is fully `ALL_COMMITTED`, advance `m_bit_area_start_mvccid` and shrink the area.
2. **Long-tran eviction.** If after LTRIM the area still exceeds `LONG_TRAN_THRESHOLD`, residual still-active IDs migrate into the sorted overflow array.

Two cached scalars short-circuit common queries:

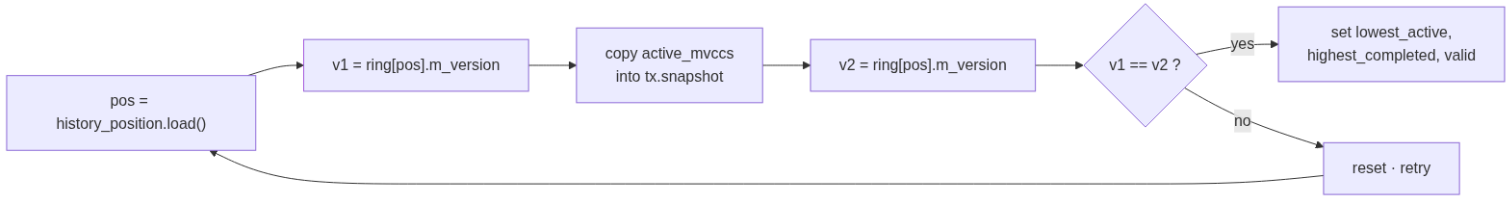
- **`lowest_active_mvccid`** — everything strictly below is **known-completed**. Probe-free fast path.
- **`highest_completed_mvccid`** — everything ≥ this is **certainly active or future**. Probe-free fast path.



© 2026 CUBRID Corporation. All rights reserved.

Snapshot construction — lock-free retry

```
// mvcctable::build_mvcc_info – src/transaction/mvcc_table.cpp (condensed)
while (true) {
    index = m_trans_status_history_position.load ();
    const mvcc_trans_status &ts = m_trans_status_history[index];
    trans_status_version = ts.m_version.load ();
    ts.m_active_mvccs.copy_to (tdes.mvccinfo.snapshot.m_active_mvccs,
        mvcc_active_tran::copy_safety::THREAD_UNSAFE);
    if (trans_status_version == ts.m_version.load ())
      break;                                    /* stable copy */
    tdes.mvccinfo.snapshot.m_active_mvccs.reset_active_transactions ();
}
```



- Lock-free: readers check `m_version` before and after copying the active set; retry if a writer changed it.

© 2026 CUBRID Corporation. All rights reserved.

Visibility predicate — `mvcc_satisfies_snapshot`

```
// mvcc_satisfies_snapshot — src/transaction/mvcc.c (skeleton)
if (!MVCC_IS_HEADER_DELID_VALID (rec_header)) {       /* not deleted */
    if (MVCC_IS_REC_INSERTED_BY_ME (...))       return SNAPSHOT_SATISFIED;
    if (MVCC_IS_REC_INSERTER_IN_SNAPSHOT (...)) return TOO_NEW_FOR_SNAPSHOT;
    return SNAPSHOT_SATISFIED;
} else {                                              /* deleted */
    if (MVCC_IS_REC_DELETED_BY_ME (...))        return TOO_OLD_FOR_SNAPSHOT;
    if (MVCC_IS_REC_INSERTER_IN_SNAPSHOT (...)) return TOO_NEW_FOR_SNAPSHOT;
    if (MVCC_IS_REC_DELETER_IN_SNAPSHOT  (...)) return SNAPSHOT_SATISFIED;
    return TOO_OLD_FOR_SNAPSHOT;
}
```

- Three-valued result. `TOO_NEW` → walk `prev_version_lsa`; `TOO_OLD` → stop. Fast path via cached scalars `lowest_active_mvccid` / `highest_completed_mvccid` skips the bit-area probe.

© 2026 CUBRID Corporation. All rights reserved.

Visibility worked example

Three concurrent readers — A: snapshot `{18,19,30}`; B: snapshot `{19,30,32}`, MVCCID 32; C: snapshot `{19,30,32,34}`, MVCCID 34.

Version `(ins, del)`	A `{18,19,30}`	B `{19,30,32}`, id=32	C `{19,30,32,34}`, id=34
`(3, 7)`	SATISFIED	SATISFIED	SATISFIED
`(7, 18)`	TOO_NEW → prev_lsa	SATISFIED	SATISFIED
`(18, 32)`	TOO_NEW → prev_lsa	DELETED_BY_ME → TOO_OLD	SATISFIED
`(32, –)`	TOO_NEW → prev_lsa	SATISFIED (my insert)	SATISFIED

- Asymmetry on `(ins=18, del=32)`: **A** finds inserter 18 in its active set, so the inserter check fires first → `TOO_NEW` → walks `prev_version_lsa`. **B** is the deleter (`DELETED_BY_ME` → `TOO_OLD`). **C** sees the deleter committed before its snapshot → not visible.

© 2026 CUBRID Corporation. All rights reserved.

CUBRID

Commit / rollback — `complete_mvcc`

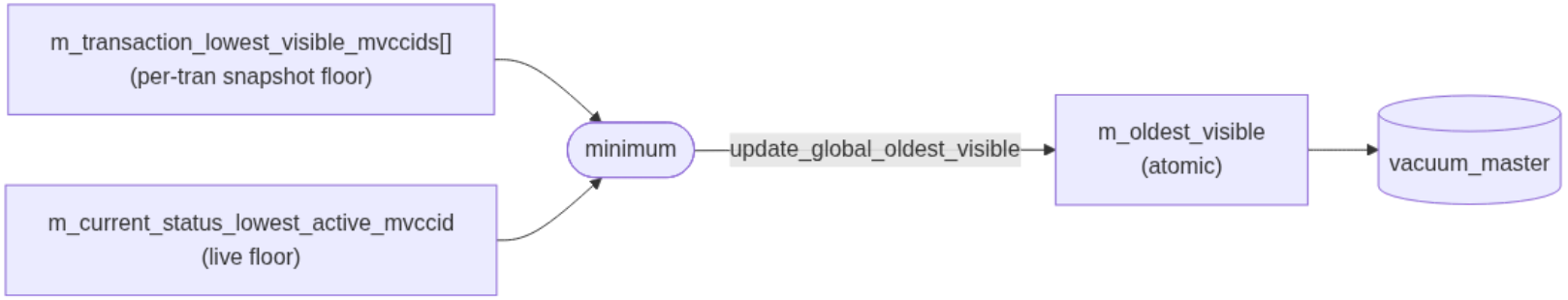
```
// mvcctable::complete_mvcc — src/transaction/mvcc_table.cpp (condensed)
std::unique_lock<std::mutex> ulock (m_active_trans_mutex);
mvcc_trans_status &next_status =
    next_trans_status_start (next_version, next_index);   /* reserve slot N+1 */
m_current_trans_status.m_active_mvccs.set_inactive_mvccid (mvccid);   /* flip bit */
m_current_trans_status.m_last_completed_mvccid = mvccid;
m_current_trans_status.m_event_type = committed ? COMMIT : ROLLBACK;
next_tran_status_finish (next_status, next_index);        /* publish ring slot */
ulock.unlock ();
if (global_lowest_active == mvccid)
  advance_oldest_active (next_status.m_active_mvccs.compute_lowest_active_mvccid ());
```

- **Under the mutex**: flip the active-set bit, reserve and publish the next history-ring slot, bump the slot's atomic `m_version` — what snapshot readers' retry loop trips on. The ring publish (`m_trans_status_history_position` bumped last) makes new state visible lock-free.
- **Outside the mutex**: if this MVCCID was the global lowest active, compute and advance `lowest_active`. Cheap when it doesn't trigger; correct when it does.

© 2026 CUBRID Corporation. All rights reserved.

CUBRID

Vacuum coordination — the watermark



- The watermark is `mvcctable::m_oldest_visible` — a single atomic, the source of truth for "what's reclaimable".
- `update_global_oldest_visible` (vacuum master) sweeps `m_transaction_lowest_visible_mvccids[]` plus the live `m_current_status_lowest_active_mvccid`, takes the minimum, publishes it.
- `mvcc_satisfies_vacuum` reads this single atomic to decide whether a version is reachable.
- **The cost of MVCC**: one long-running write txn with a small MVCCID pins `m_oldest_visible` and blocks reclamation of everything newer.

© 2026 CUBRID Corporation. All rights reserved.

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
Headers — record, snapshot, enum	`struct mvcc_rec_header`, `struct mvcc_snapshot`, `struct mvcc_info`, `enum mvcc_satisfies_snapshot_result`	`mvcc.h`
Active-set primitive	`struct mvcc_active_tran`	`mvcc_active_tran.hpp`
Global table — class + trans status	`class mvcctable`, `struct mvcc_trans_status`	`mvcc_table.hpp`
Hot paths — ID issuance, snapshot build, active-set probe	`get_new_mvccid`, `build_mvcc_info`, `is_active`	`mvcc_table.cpp`
Hot paths — commit/rollback, vacuum watermark	`complete_mvcc`, `update_global_oldest_visible`, `compute_oldest_visible_mvccid`	`mvcc_table.cpp`
Read path — fast check + predicate	`mvcc_is_id_in_snapshot`, `mvcc_satisfies_snapshot`, `mvcc_satisfies_vacuum`	`mvcc.c`

© 2026 CUBRID Corporation. All rights reserved.

CUBRID

Beyond CUBRID — comparative designs

Engine / approach	Versions	Active set	Reclamation	Note
PostgreSQL	in-place on heap (HOT)	sorted XID list per snapshot	autovacuum scans heap	SSI for serializable; bloat is the failure mode
Oracle	undo segments	latch + active-tx list	undo retention; UNDO_RETENTION param	"ORA-01555 snapshot too old" is the watermark complaint
InnoDB	undo log (rollback segments)	read view = transaction ID list	purge thread	very close to CUBRID's shape
Hekaton / Silo / Cicada	in-memory MVOCC	cache-aware versioned slots; no central registry	epoch-based GC	redesigned for many cores; orthogonal to disk-resident
CUBRID	log-resident (`prev_version_lsa`)	bit array + long-tran array	`m_oldest_visible` + vacuum master	the design we just walked through

Frontiers: **SSI** (Cahill 2008), **in-memory MVCC survey** (Wu VLDB 2017), **CC at 1000 cores** (Yu VLDB 2015).

© 2026 CUBRID Corporation. All rights reserved.

Thank you

Q & A

- Analysis: `knowledge/code-analysis/cubrid/cubrid-mvcc.md`
- Code: `src/transaction/mvcc.{h,c}` · `mvcc_table.{hpp,cpp}` · `mvcc_active_tran.{hpp,cpp}`

© 2026 CUBRID Corporation. All rights reserved.