CUBRID Storage Engine

Disks, Pages, Heaps, B+Trees, Hash, Overflow

2026-05 · Code Analysis Seminar

Agenda

1. **Why a storage engine** — what the layer must guarantee
2. **The layered stack** — one diagram of the whole thing
3. **From the bottom up** — `disk_manager` → `page_buffer` + DWB → heap → btree → ehash → overflow
4. **Crossing concerns** — TDE, the WAL contract, recovery's view
5. **Where to read next** — symbol-level pointers

> Closing: **Beyond CUBRID** — InnoDB, PostgreSQL, RocksDB, Oracle.

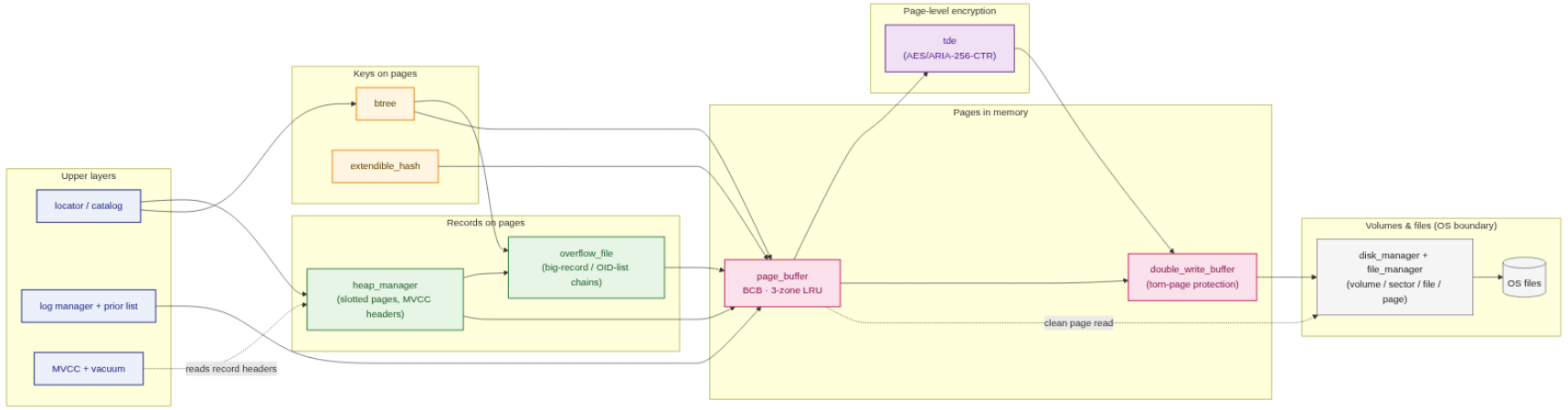
Why a storage engine, what it must guarantee

The storage engine is everything **below** the catalog and the locator and **above** the operating system's filesystem. It takes byte streams from the OS and produces fixed-size pages, records, indexes — under four hard guarantees:

- **Durability.** A committed write must survive a crash. Implemented by Write-Ahead Logging (WAL) — log records are forced to disk **before** the dirty page they describe.
- **Atomicity at page grain.** A single page write is either fully old or fully new on disk — never a half-old / half-new "torn" page. CUBRID buys this with the **Double Write Buffer**.
- **Fast point reads + range scans.** Heap pages give random access by OID; B+Tree leaves give ordered scans by key.
- **Concurrent access without corruption.** Multiple workers may read or mutate the same page; the page buffer hands out a custom **read / write / flush latch** on each in-memory frame.

> Everything in the rest of the talk is one of these four guarantees, paid for in code.

The layered stack



- **Strict layering.** Each layer knows only the one below it.
- **Page buffer is the choke point.** Every record / key layer talks to it.

Every record layer talks to the page buffer, not the disk

The single structural rule that makes WAL ordering enforceable.

- **The page buffer is the only module allowed to call `pread` / `pwrite`** against a CUBRID data volume (via `disk_manager` + `file_io`). The heap, the B+Tree, the catalog, ehash — none of them ever touch a file descriptor directly.
- **Mutation order.** A heap insert / B+Tree split / ehash bucket split (1) acquires the page's `PGBUF_LATCH` in WRITE, (2) mutates the in-memory frame, (3) produces a log record routed through the prior list to the log manager, then (4) marks the page dirty and releases the latch.
- **Flush order.** The page buffer's flush path refuses to write a dirty page until the log manager confirms the page's last-modifying LSA is durable. ARIES's **log first, then page** rule, made structural.
- **Page latch ≠ transactional lock.** The latch is short, embedded in the BCB, microsecond-scale. Transactional locks live in the lock manager (see the **Lock Manager** deck).

> If a record layer ever bypassed the page buffer, WAL ordering would collapse. The architecture forbids it by construction.

Part II

From the bottom up

CUBRID

`disk_manager` — volumes, sectors, files, pages

The OS-file boundary. Four levels:

Level	What it is	CUBRID name
Volume	one OS file	`VOLID`
Sector	64 contiguous pages = disk-manager allocation unit (default `1 MiB`)	`SECTID`, `nsect_total`
File	a bundle of sectors owned by one relation	`VFID`
Page	the I/O unit; named by `(volid, pageid)`	`VPID`

- **Permanent vs temporary purpose split.** Temp files (sort spills, hash partitions) live on a separate volume class and bypass WAL — they never need to survive a crash.
- **Two-step sector reservation.** A growing file (1) reserves N sectors from the in-memory **disk cache**, then (2) commits the reservation into the on-disk **sector allocation table (STAB)** under an ARIES nested top-action. Concurrent writers cooperate without serializing on the STAB.
- **Adaptive volume extension.** When the cache runs short, `disk_volume_expand` / `disk_add_volume` grow the current volume or add a new one — same nested top-action discipline.

> Vocabulary: `VPID`, sector, file, volume. Every other doc in the section reuses it.

`page_buffer` — BCB array + three-zone LRU

Most of the storage-engine engineering effort lives here.

- **Page table** — fixed 1,048,576-bucket VPID hash. `VPID → PGBUF_BCB → in-memory frame`.
- **BCB (`PGBUF_BCB`)** — one per cached page: `vpid`, `fcnt`, latch state, dirty flag, LRU pointers, packed flags word (`zone | LRU_index | bits`), `oldest_unflush_lsa`.
- **Three-zone LRU** — `LRU_1_ZONE` (cold, new arrivals) → `LRU_2_ZONE` (warm) → `LRU_3_ZONE` (hot). Promotion on second touch; demotion on age.
- **Per-thread private LRU lists + shared list, with adjustable quotas.** Per-worker private lists defend against scan flooding (a sequential scan can only evict from its own list until quota); the shared list absorbs spillover.
- **Direct victim hand-off via lock-free queues (LFCQ).** A thread looking for a free slot sleeps on a high- or low-priority queue; the next flusher hands it the BCB directly — no LRU walk on the hot path.
- **Custom RW / flush latch on each BCB** — `PGBUF_LATCH_READ` / `_WRITE` / `_FLUSH`. Three-state, not POSIX rwlock — flush is a third state so reads and the flusher can coexist while writers are excluded.
- **Ordered fix** — multi-page operations follow a fixed acquisition order to prevent latch deadlock.

> One BCB · one page · one latch. The buffer pool's most important invariant.

`double_write_buffer` — torn-write protection

A torn page is half-old / half-new on disk after a crash mid-write. WAL can recover **a coherent before-state**, not a Frankenstein. The DWB defeats this.

- **Volume.** A separate permanent file (`<db>_dwb`), 512 KiB – 32 MiB, split into 1–32 power-of-two blocks. `LOG_DBDWB_VOLID` is reserved for it.
- **Producer side.** A position-with-flags 64-bit atomic word coordinates concurrent producers. A filled block goes to the flush daemon.
- **Recovery side.** At restart, `dwb_load_and_recover_pages` reads each staged copy, checks its LSA, and if the home page is torn — fails its checksum — replaces the home page from the DWB copy. **All before log replay starts.**

`heap_manager` — slotted pages + MVCC headers

Variable-length user records on fixed-size pages, with MVCC visibility decided **without** leaving the heap page.

- **Slotted page.** Header + slot directory (page tail) + record bodies (page front); free space in between. Slots have **stable identifiers** — OIDs never shift on compaction.
- **OID = `(volid, pageid, slotid)`.** The on-disk record identifier. Every index entry, every log record, every visibility check names rows by OID.
- **Record types** packed into a 4-bit `record_type` field:

Type	Meaning
`REC_HOME`	Normal record on its home page
`REC_RELOCATION` → `REC_NEWHOME`	Forwarding: UPDATE outgrew slot but fits another heap page
`REC_BIGONE`	Forwarding to overflow file (record too big for any heap page)
`REC_MARKDELETED` / `REC_DELETED_WILL_REUSE`	Tombstones — kept forever, or reusable after vacuum

- **MVCC header inline in the record.** `insert_MVCCID`, `delete_MVCCID`, `prev_version_lsa` — visibility predicate answered without touching the log or an undo segment.

CUBRID

`btree` — latch-coupled B+Tree

CUBRID's primary access path. Slotted-page nodes; three node types — `LEAF`, `NON_LEAF`, `OVERFLOW`.

```
// non_leaf_rec / leaf_rec — src/storage/btree.h
struct non_leaf_rec { VPID pnt;  short key_len; };   // [fixed][key bytes]
struct leaf_rec    { VPID ovfl; short key_len; };    // [fixed][key bytes][OID list]
```

- **Key‖OID concatenation in non-unique leaves.** Duplicate keys exist; OID is the tie-breaker, so the comparison key is the full pair. Search is one binary search per node.
- **Unique enforcement at the OID-suffix level.** Unique-index leaf entries store pure key + adjacent OID; the unique check runs **under the leaf's X latch** so two concurrent inserters cannot both observe "no duplicate".
- **OID-list overflow.** When a non-unique key accumulates more OIDs than the leaf record can hold, the surplus spills to a per-key overflow chain (managed via the overflow file's symbol-level module).
- **Latch coupling on descent.** Acquire READ on each non-leaf, release the parent once the child is fixed. The leaf is upgraded to WRITE on the write path; if a split has invalidated the descent, restart from the root.
- **Splits & merges.** A split is logged as a **logical undo** (`LOG_SYSOP_END_LOGICAL_UNDO`) — replaying physical undo of a multi-page split would be brittle.

CUBRID

`extendible_hash` — Fagin-style directory + buckets

The **second** on-page organisation. Used only by a handful of internal callers.

- **Theory** — Fagin et al. 1979. A pseudo-key from a uniform hash function indexes a **directory of `2^d` pointers** by its leftmost `d` bits (`d` = global depth). Each directory entry points at a **bucket** with its own **local depth** `d_b ≤ d`.
- **Split policy.** When a bucket overflows: if `d_b < d` (shared bucket), allocate a sibling and re-distribute by the next bit — directory size unchanged. If `d_b = d`, **double the directory first**, then split.
- **CUBRID-specific shape.** EHID-rooted directory file + slotted bucket pages with binary search within each bucket. Split / merge run inside a system op (`LOG_SYSOP_END_LOGICAL_UNDO`); `RVEH_*` log records carry the WAL.
- **Four callers, all internal:** class-name → OID lookup (`classname_table`), catalog repr-id directory, UPDATE/DELETE OID dedup (`is_tmp = true` suppresses logging), hash-list/hash-set scans (`query_hash_scan`).

> **User-defined indexes never use ehash.** Only B+Tree.

`overflow_file` — big-record + big-OID-list spill

One symbol-level module (`src/storage/overflow_file.c`) serves three different spillover needs.

Caller	What spills	File type
`heap_manager`	a record too big for any heap page	`FILE_MULTIPAGE_OBJECT_HEAP`
`btree` overflow **key**	a key too big for a leaf record	`FILE_BTREE_OVERFLOW_KEY`
`btree` overflow **OID list**	too many duplicate-key OIDs for one leaf	per-tree OID-overflow chain

- **One overflow file per containing object.** Pages allocated from it are owned by exactly that heap / B+Tree — no cross-tenant overflow.
- **Single-linked page chain per logical record.** Each page header carries a "next-page" VPID; the first page also carries the record length.
- **Reference at the home slot.** Heap uses `REC_BIGONE` (one VPID); B+Tree uses the `leaf_rec.ovfl` pointer or an embedded VPID for overflow keys.
- **Vacuum-aware.** When MVCC vacuum reclaims a big record, the overflow chain is freed page-by-page under WAL.

TDE — encrypt-at-rest at the page boundary

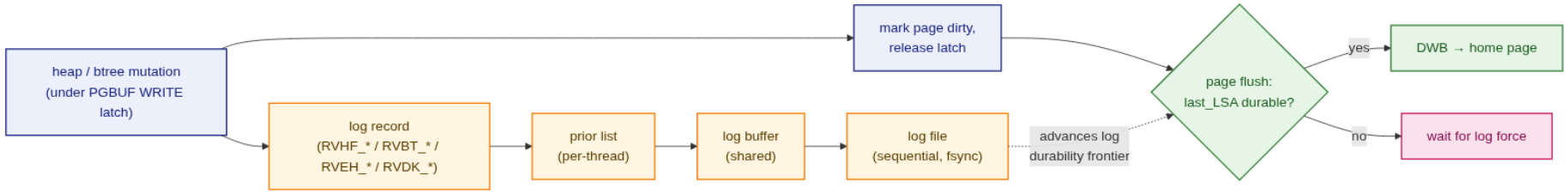
Transparent Data Encryption: per-page encrypt-on-flush / decrypt-on-read. Largely orthogonal to the rest of the engine.

- **Two-level key hierarchy.** A **Master Key (MK)**, supplied by the DBA at startup, lives outside the database in a `<db>_keys` file. The MK wraps three per-database **Data Encryption Keys (DEK)** — DATA, LOG, TEMP. Rotating the MK rewraps the DEKs; the data pages are untouched.
- **Cipher & mode.** AES-256-CTR or ARIA-256-CTR. CTR mode lets a page be encrypted in one parallel pass, with no length growth.
- **Per-page nonce.** LSA for permanent pages, an atomic counter for temp, logical pageid for log. The CTR invariant — `(key, nonce)` never repeats — falls out for free.
- **Encryption boundary.** Page header (`FILEIO_PAGE_RESERVED` with `pflag` , `tde_nonce`) stays **plaintext** — the buffer manager must read `pflag` to know **whether** to decrypt. The tail watermark also stays plaintext (torn-write check works on ciphertext too). Body in between is encrypted.
- **Per-file granularity.** TDE is a **tablespace-style dial** — each `FILE_HEADER` carries its own `TDE_ALGORITHM` ; `file_alloc` stamps the algorithm onto each new page's `pflag` .

CUBRID

The WAL contract — `page_buffer` ↔ `log_manager`

The single rule that makes ARIES recovery work.



- **Log first, then page.** The buffer manager refuses to flush a dirty page until the log manager confirms `oldest_unflush_lsa` is durable.
- **Every layer participates.** Disk manager logs sector / volume changes (`RVDK_*`); heap logs record mutations (`RVHF_*`); B+Tree logs node mutations (`RVBT_*`); ehash logs directory + bucket changes (`RVEH_*`); overflow file logs page-chain extensions.
- **The dependency is structural.** Without it, replaying redo against a page newer than the log would be impossible — the storage engine would be unrecoverable.

Recovery's view of the storage stack

Three passes (ARIES). The storage stack participates in all three.

Pass	What recovery does	Storage-engine role
Analysis	walk forward from the last `chkpt_lsa`; rebuild the dirty-page table and the active-transaction table	reads checkpoint records produced by the page buffer's flush path
Redo	re-apply every log record whose page LSA is older than the record	every `RVxx_*` handler in `disk_manager`, `heap_file`, `btree`, `extendible_hash`, `overflow_file` runs again
Undo	roll back losers using per-LSA undo records; B+Tree splits use `LOG_SYSOP_END_LOGICAL_UNDO`	logical undo lets a split unwind without page-grain reversibility

- **DWB handshake runs before analysis.** Any home page that fails its checksum is replaced from its DWB copy. Redo then sees a coherent before-state.
- **MVCC vacuum is a heap-page mutation in its own right.** It takes the WRITE latch and emits `RVHF_*` records, so vacuum survives crash recovery the same way as user DML.

Source walkthrough — where to read next

Symbol names are the stable handle; line numbers drift. Detail docs live in `knowledge/code-analysis/cubrid/`.

Module	Symbols	Files	Detail doc
Disk / file	`disk_reserve_sectors`, `file_alloc`, `file_create`	`disk_manager.{h,c}` · `file_manager.{h,c}`	`cubrid-disk-manager.md`
Page buffer · DWB	`pgbuf_fix`, `pgbuf_get_victim`, `dwb_add_page`, `dwb_flush_block`	`page_buffer.{h,c}` · `double_write_buffer.{hpp,cpp}`	`cubrid-page-buffer-manager.md` · `cubrid-double-write-buffer.md`
Heap	`heap_insert_logical`, `heap_update_logical`, `heap_get_visible_version`	`heap_file.{h,c}` · `slotted_page.{h,c}`	`cubrid-heap-manager.md`
B+Tree	`btree_insert`, `btree_delete`, `btree_range_scan`, `BTREE_NEED_UNIQUE_CHECK`	`btree.{h,c}` · `btree_load.c` · `btree_unique.{hpp,cpp}`	`cubrid-btree.md`
Ehash	`xehash_create`, `ehash_search`, `ehash_insert`, `ehash_delete`	`extendible_hash.{h,c}`	`cubrid-extendible-hash.md`
Overflow · TDE	`overflow_insert`, `overflow_get`; `tde_encrypt_data_page`, `file_set_tde_algorithm`	`overflow_file.{h,c}` · `tde.{h,c}`	`cubrid-overflow-file.md` · `cubrid-tde.md`

> `git grep -n '<symbol>' src/storage/` is your friend.

Beyond CUBRID — comparative storage designs

CUBRID's dials, set against four neighbouring engines.

Engine	Storage layout	Spill / overflow	Atomicity
InnoDB	Clustered B+Tree (data lives in the primary-key leaves) + compressed pages	Off-page columns chained from the row	Doublewrite buffer (same shape as CUBRID's DWB)
PostgreSQL	Heap files + separate B+Tree, GiST, GIN indexes	**TOAST** — per-column out-of-line table; large values compressed first	`full_page_writes = on` — first-after-checkpoint page logged in full
RocksDB (LSM contrast)	Log-Structured Merge: sorted runs + compaction, **no in-place page rewrites**	Wide-column values handled by separate value log (BlobDB)	No torn writes possible — every write is append-only
Oracle	Heap-organised tables + B+Tree indexes; **UNDO segments** store before-images for read-consistency, not inline MVCC	Row migration / row chaining within the same segment	Atomic block writes via the redo + UNDO pair

- **CUBRID's position.** Heap + B+Tree + per-page DWB + inline-MVCC headers. Same family as InnoDB / Oracle; the LSM contrast lives in RocksDB.

Thank you

Q & A

- Section overview: `knowledge/code-analysis/cubrid/cubrid-overview-storage-engine.md`
- Per-module detail docs: `cubrid-disk-manager.md` · `cubrid-page-buffer-manager.md` · `cubrid-double-write-buffer.md` · `cubrid-heap-manager.md` · `cubrid-btree.md` · `cubrid-extendible-hash.md` · `cubrid-overflow-file.md` · `cubrid-tde.md`
- Code: `src/storage/*.{h,c,hpp,cpp}`