CUBRID Locator

OID Workspace, Bulk Fetch/Flush, Server-Side Insert/Update/Delete Bridge

2026-05 · Code Analysis Seminar

Agenda

0. **The problem** — bridging in-memory objects and on-disk OIDs
1. **Theory** — object identity, the workspace pattern, bulk vs per-row
2. **Common patterns** — what every object-aware engine looks like
3. **CUBRID client side** — `MOP`, workspace, `LC_COPYAREA`, bulk fetch / flush
4. **CUBRID server side** — the `locator_*_force` family and the canonical pipeline
5. **Lifecycle and integrations** — transient OIDs, triggers, FK, replication

> Closing: **Beyond CUBRID** — ZODB, ORM sessions, PostgreSQL's inline executor.

Why a locator at all

CUBRID has three layers that need to agree on what **a row** is, and they speak three different vocabularies:

Layer	Vocabulary	Examples
Object / row	decoded values	`DB_VALUE`, `PT_NODE`, `RECDES` with offsets parsed
Storage	raw bytes + OIDs	heap, btree, page buffer
Cross-cutting	OIDs + class metadata	lock manager, MVCC, log, vacuum, HA replication

- An `INSERT` touches **ten** subsystems in a fixed order — find page, allocate OID, X-lock, write row, update every B-tree, unique-check, FK-check, log per page, replicate, bump catalog stats.
- None of those layers should know how to do all the others' jobs. They need a **conductor**.
- That conductor is the **locator** — `locator_cl.c` on the client, `locator_sr.c` on the server, `locator.{h,c}` as the protocol in between.

Theoretical background

- **Object identity** (Petrov **Database Internals**, ch. 3–4): name a row so the name survives page compaction and B-tree leaf moves. The OID is the artifact; the locator creates, resolves and mutates rows at that OID.
- **Workspace pattern** (Garcia-Molina, Ullman, Widom §10.6): the in-memory cache of objects the application has touched. Reads pull in, writes mark dirty, **commit** flushes the dirty set in **one batch**.
- **Bulk fetch / flush vs per-row**: N rows × N RTTs becomes one buffer per transaction. The same shape lives in XA, in PostgreSQL `COPY`, in MySQL `LOAD DATA`, and in every ORM `session.flush()`.

CUBRID

Five common patterns

Every object-aware engine — ZODB, ObjectStore, GemStone, ORMs, **CUBRID** — sets the same five dials, just to different values:

1. **Identity table.** Hash from in-memory pointer → persistent identifier. Survives across statements. CUBRID's MOP↔OID hash; ZODB's `_p_oid`.
2. **Dirty-bit batching.** The workspace tracks a **dirty list**; commit/flush packs every dirty object into one buffer in one pass.
3. **Copy-area marshaling.** A single bidirectional buffer encodes both **descriptors** (operation, class, OID, length) and **row bodies**. One alloc, one wire send.
4. **Canonical server entry.** One function per DML operation, called from every code path that mutates data. Cross-cutting work (lock, log, FK, replication) lives in exactly one place.
5. **FK + replication fan-out** ride that single entry. Coverage is complete by construction — there is no "I forgot to replicate this code path".

CUBRID is **one point** in this dial-space — workspace-explicit, bulk-flush per transaction, one `locator_attribute_info_force`.

CUBRID

Inside CUBRID

How the bridge is realized

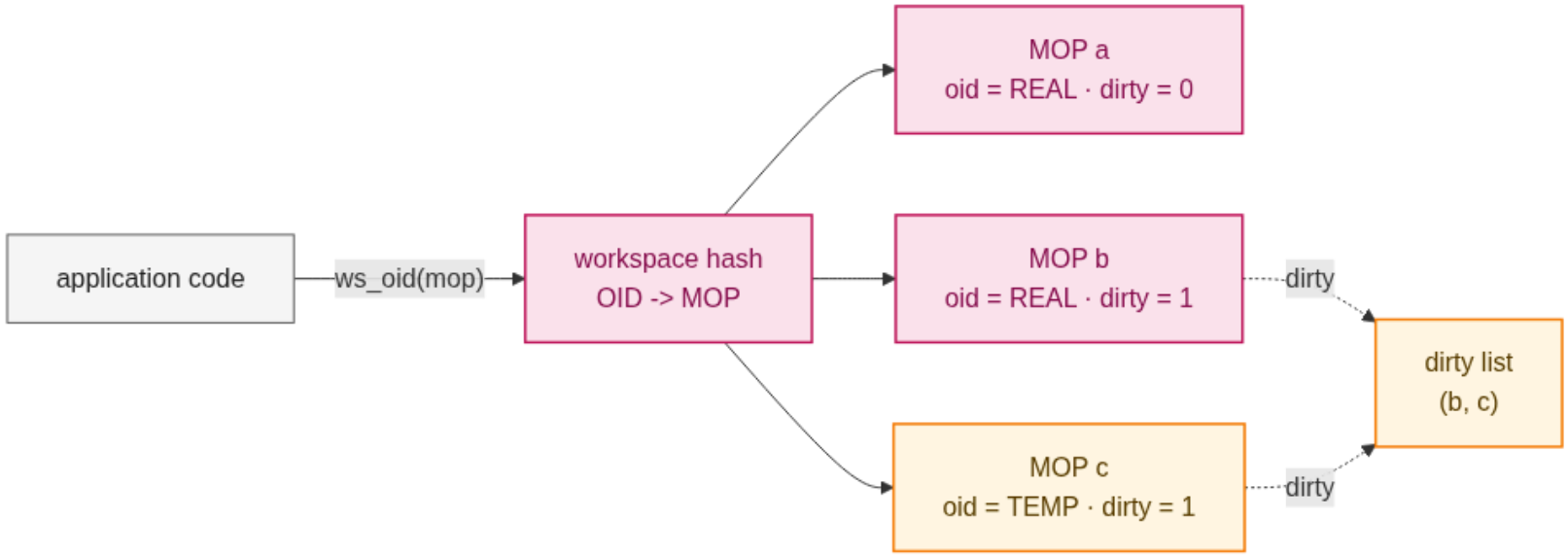
CUBRID

The MOP — client-side identity

```
// db_object (typedef MOP) — src/object/work_space.h
struct db_object
{
  OID oid;                  /* server OID; OID_ISTEMP until flushed */
  MOP class_mop;            /* MOP of the class object */
  void *object;             /* in-memory decoded object (MOBJ) */
  unsigned dirty:1;         /* needs flush */
  unsigned deleted:1;       /* logical delete */
  unsigned no_objects:1;    /* class with no instances cached */
};
```

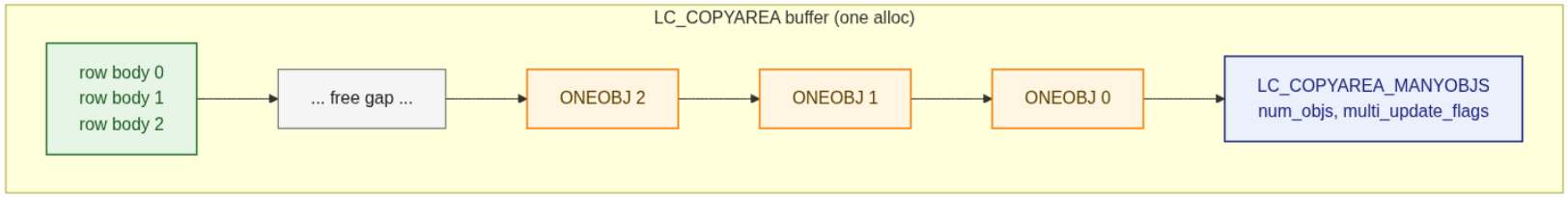
- A `MOP` is the application's **long-lived handle** — held across statements, returned from queries, used to navigate between rows.
- `oid` carries either a real `(volid, pageid, slotid)` or a temp sentinel (`OID_ISTEMP`) for not-yet-flushed instances.
- The three bits (`dirty`, `deleted`, `no_objects`) are the entire workspace lifecycle.

Workspace mechanics



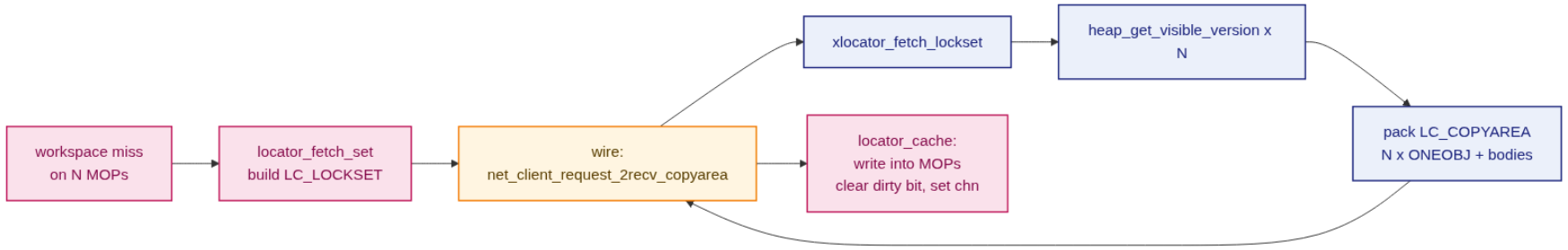
- New `db_create` mints a **temp OID** locally — no server contact. The MOP enters the dirty list with `LC_FLUSH_INSERT`.
- Existing MOPs flip `dirty` on `locator_update_instance` and on attribute writes; the dirty list is what `locator_mflush` scans at flush time.

`LC_COPYAREA` layout — bidirectional packing



- **Row bodies grow forward** from the front; `LC_COPYAREA_ONEOBJ` **descriptors grow backward** from the end, anchored at `LC_COPYAREA_MANYOBJS`.
- Each descriptor carries: operation, flag, hfid, class_oid, oid, length, offset (4 ints + 1 HFID + 2 OIDs).
- Packing is bounded by the watermark where bodies meet descriptors. One alloc, one wire send.

Bulk fetch — `locator_fetch_*`



- Six client entries — `_object`, `_class`, `_class_of_instance`, `_instance`, `_set`, `_nested` — differ in **scope** but all converge on `locator_lock` / `locator_lock_set` and round-trip to the server.
- `_set` is the prefetch path: N MOPs in one buffer instead of N × RTT.
- **Version policy** is the `LC_FETCH_VERSION_TYPE` knob — `MVCC` (snapshot, no lock), `DIRTY` (S-lock, latest committed), `CURRENT` (caller holds X-lock).

Bulk flush — client-side `locator_mflush_cache`

```
// locator_mflush_cache — src/transaction/locator_cl.c (members)
struct locator_mflush_cache {
  LC_COPYAREA          *copy_area;    /* staging buffer */
  LC_COPYAREA_MANYOBJS *mobjs;        /* N-objects descriptor */
  LC_COPYAREA_ONEOBJ   *obj;          /* current ONEOBJ slot */
  LOCATOR_MFLUSH_TEMP_OID *mop_toids; /* temp-OID MOPs to patch */
  RECDES recdes;                      /* current record body */
  /* + class_mop / hfid (last-class cache), flags */
};
```

- `ws_map_dirty` walks the dirty list; for each MOP, `locator_mflush` encodes `RECDES` and appends one `LC_COPYAREA_ONEOBJ`.
- **Classes flushed before instances**; FK-aware ordering inside the dirty list keeps unique / FK probes sane.
- If the buffer overflows, `locator_mflush_force` drains it now, resets, and continues with the overflowing object.

Server-side bridge — the `locator_*_force` family

The server has **one canonical entry** that every DML eventually calls. Three flavors fan out from it.

Entry	Origin	What it does
`xlocator_force`	wire (client flush)	top-op; loops over `LC_COPYAREA_ONEOBJ`s
`locator_attribute_info_force`	executor / triggers / DDL / xlocator_force	switch on `LC_COPYAREA_OPERATION`, encode RECDES, dispatch
`locator_insert_force`	attribute-info dispatch	heap insert + indexes + FK + replication
`locator_update_force`	attribute-info dispatch	heap update + diff-driven indexes + FK + replication
`locator_delete_force`	attribute-info dispatch	heap delete + indexes + FK cascade + replication
`locator_force_for_multi_update`	UPDATE with triggers / cascade	multi-update path with `START_/END_MULTI_UPDATE` markers
`xlocator_force_repl_update`	HA applier	replication-side replay

Every row mutation in CUBRID — executor-driven, workspace-driven, trigger, cascade, HA — eventually passes through this family.

CUBRID

Canonical force path — `locator_attribute_info_force`

```
// locator_attribute_info_force — src/transaction/locator_sr.c
switch (operation) {
  case LC_FLUSH_UPDATE:                              /* read old row */
    locator_lock_and_get_object (..., X_LOCK, ...);  /* fallthrough */
  case LC_FLUSH_INSERT:
    locator_allocate_copy_area_by_attr_info (...);   /* encode */
    insert ? locator_insert_force (...) : locator_update_force (...);
    break;
  case LC_FLUSH_DELETE: locator_delete_force (...);
}
```

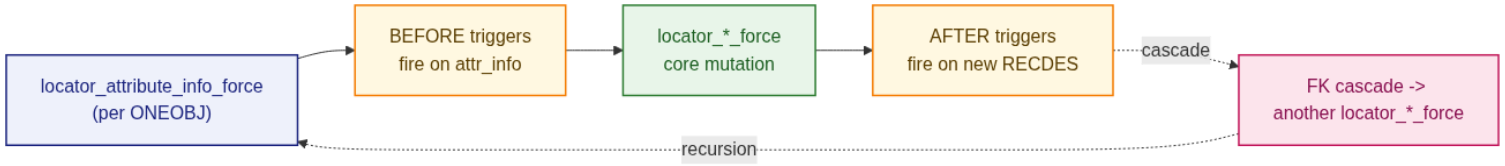
- **UPDATE falls through into INSERT.** Both share the encode step; only the read+lock prelude differs.
- **Locking happens here, not in the heap.** `need_locking` and `force_update_inplace` decide whether the force takes the lock or trusts the caller.
- **Snapshot consulted for UPDATE/DELETE, never INSERT** — inserts have no prior version to compare against.

CUBRID

The canonical pipeline — one row through the force path

- The order is **fixed by design**. Row first (heap stamps the MVCC header), indexes next (so unique-check sees the new key), FK third (parent must exist when child key probes), log and replication trail the successful mutation.
- This is the cashing of `cubrid-mvcc.md`'s claim that **MVCC headers are stamped by `locator_*` flows**: every heap call comes from a force function.

Trigger and integrity rules



- Triggers fire from inside `locator_attribute_info_force` — `BEFORE` on the encoded `attr_info`, `AFTER` on the new `RECDES`.
- FK cascades **re-enter** the force family on the cascading rows, packaged as fresh `LC_FLUSH_*` operations.
- The recursion bottoms out because cascade depth is bounded by the FK graph, and each level re-takes locks and rechecks FK.

FK enforcement — per-row probe

```
// locator_check_foreign_key — src/transaction/locator_sr.c
static int
locator_check_foreign_key (THREAD_ENTRY *thread_p, HFID *hfid,
                           OID *class_oid, OID *inst_oid,
                           RECDES *recdes, RECDES *new_recdes,
                           bool *is_cached,
                           LC_COPYAREA **cache_attr_copyarea);
```

- Walks the FK list on the class representation; extracts the referencing-column key from `recdes`.
- Probes the **parent class's PK B-tree** via `btree_keyoid_checks`. On miss → `ER_FK_INVALID`, the whole insert/update fails.
- Cascade actions (`ON DELETE CASCADE`, `ON UPDATE SET NULL`) re-enter the force family on the dependent rows; `not_check_fk` and `dont_check_fk` flags suppress redundant checks when the executor has already verified.
- `locator_check_unique_btree_entries` is the CHECKDB variant — same machinery, batch-mode integrity sweep.

Replication path — complete by construction

- The replication record is built **inside** `locator_insert_force` / `_update_force` / `_delete_force_internal`, **after** the heap and B-tree primitives succeed. The record always describes the post-state.
- `repl_info.repl_info_type` is the format knob: `REPL_INFO_TYPE_RBR_NORMAL` for row-based (default), `REPL_INFO_TYPE_STMT_NORMAL` for statement-based, `..._AT_LEAST_ONE_RECORD` for multi-row.
- Because **every** DML passes through this family, replication coverage is **complete by construction** — no "I forgot this code path" failure mode.

CUBRID

Transient OIDs and OID promotion

- Temp OIDs (`OID_ISTEMP`) never reach the server — they live only in the client workspace.
- The heap manager's slot assignment is what fixes the permanent OID; the reply buffer carries it back per `ONEOBJ`.
- For **self-referencing** rows (catalog entries that need their own OID before the body), `xlocator_assign_oid` pre-mints a permanent OID via `heap_assign_address` placing a `REC_ASSIGN_ADDRESS` placeholder.

Source walkthrough — where to read next

Symbol names are the stable handle. `locator_sr.c` is ~14 000 lines, `locator_cl.c` ~7 100 — `git grep -n '<symbol>' src/transaction/` is your friend.

Topic	Symbol(s)	File
Wire types	`enum LC_COPYAREA_OPERATION`, `LC_COPYAREA_ONEOBJ`, `LC_COPYAREA_MANYOBJS`, `LC_LOCKSET`, `LC_LOCKHINT`	`locator.h`
Wire packing	`locator_pack_copy_area_descriptor`, `locator_pack_lockset`, `locator_pack_lockhint`, `locator_pack_oid_set`	`locator.c`
Workspace + bulk fetch	`locator_fetch_object`, `_fetch_set`, `locator_lock`, `locator_cache`	`locator_cl.c`
Bulk flush	`locator_mflush_cache`, `locator_mflush`, `_mflush_force`, `locator_all_flush`, `locator_force`	`locator_cl.c`
Force family	`xlocator_force`, `locator_attribute_info_force`, `locator_{insert,update,delete}_force`	`locator_sr.c`
Constraints + reads	`locator_add_or_remove_index`, `locator_update_index`, `locator_check_foreign_key`, `locator_get_object`, `locator_lock_and_get_object`	`locator_sr.c`
OID lifecycle	`xlocator_assign_oid`, `xlocator_find_class_oid`, `locator_permoid_class_name`	`locator_sr.c`

CUBRID

Beyond CUBRID — comparative designs

Engine / pattern	Workspace?	Bulk shape	Server-side fan-in
PostgreSQL	none (no client workspace)	per-row Bind/Execute	`heap_insert` / `_update` / `_delete` called directly by executor
InnoDB (MySQL)	per-statement `Field*` decode	`ha_bulk_update_row` per table-handle	`handler::ha_write_row` etc., one per storage engine
Oracle	row sources + DBWn write-behind	per-row through row source; `FORALL BULK COLLECT` syntactic	`kdusru` / `kdusrf` from DML operator
ZODB / ObjectStore / GemStone	explicit MOP-like handle, long-lived	`transaction.commit()` flushes the conflict set	object manager dispatches per OID
ORMs (Hibernate / SQLAlchemy)	`Session` / `unit-of-work`	`session.flush()` orders by FK	per-mapper insert/update/delete SQL
CUBRID	explicit `MOP` workspace, survives transactions	`LC_COPYAREA` per flush	one `locator_attribute_info_force` switches on operation

Frontiers: **gRPC streaming bulk fetch** (modern columnar OLAP), **CDC log shipping** as the new "session.flush()", **separation-of-storage** designs that push the workspace into the proxy tier.

Thank you

Q & A

- Analysis: `knowledge/code-analysis/cubrid/cubrid-locator.md`
- Code: `src/transaction/locator.{h,c}` · `locator_cl.{h,c}` · `locator_sr.{h,c}`
- Companion decks: `cubrid-heap-manager`, `cubrid-btree`, `cubrid-lock-manager`, `cubrid-mvcc`, `cubrid-ha-replication`