POD 0009: The Data-Oriented Ledger

COMMITTED

CATEGORY	**STATUS**
Design Record	Committed
AUTHORS	**CREATED**
Vikrant Rathore <vikrant@insan.ai>, Paxos Odin Contributors	2026-09-16
LAST UPDATED	**LABELS**
2026-09-17	design, performance, memory, odin

Abstract

This record explains the layout of the `0.2.0` core: a `Ledger` that stores Lamport's acceptor variables as columns over a power-of-two window, two bitmaps that make every scan a word walk, a slot tag per cell, and messages, records, and released entries that reference a value inside that ledger instead of copying it. It states the pointer contract, gives the memory formulas per configuration, says what the change made faster and what it did not, lists the constraints the layout imposes, and records the alternatives that were rejected.

Status and Implementation Boundary

Recovery reserves one chunk of values and metadata, rather than one complete ledger window. Its indexes are relative to the active recovery base; per-peer report bitmaps use the same offsets. On chunk rollover metadata is reset while payload bytes remain uninitialised until a valid report stores them. A read-quorum selection is frozen before phase two starts, including when the window forces a retry. See the chunk-local recovery argument in the proof chapter.

The public procedures and durable/wire formats are unchanged. The size and layout of `Node` change; consumers recompile, and code inspecting its recovery arrays must use chunk-relative indexes. Raw node images are not a supported journal format.

The matched benchmark and Valgrind drivers live outside the library. They introduce no runtime dependencies, storage adapters, threads, clocks, or network services into the core. Measurements distinguish static capacity, allocated heap, resident memory, and elapsed time. Callgrind instruction counts guide investigation; they are not substitutes for uninstrumented performance measurements.

Problem Statement

In `0.1.0` a value lived inline in `Accept_Message`, `Commit_Message`, `Promise_Message`, the durable records, and `Committed`. `Message(Value)` was a union, so every envelope was as large as the largest variant plus the value, and a value was copied at each of these points: into the leader's proposal cell, into the accept envelope for each peer, into each acceptor's cell, into the acceptor's write record, into the commit envelope for each peer, into each learner's cell, and into the committed entry. For an 8-byte value the copies were noise; for a 1 KiB value they were most of the work. A profile of the in-memory benchmark on the benchmark host (POD 0007, third pass) showed the fat message union and the value copies

dominating the 1 KiB workload. The per-slot `Durable_Cell{slot, accepted: Maybe(Accepted), committed: Maybe(Value)}` also meant that a phase-one scan or a retransmission scan loaded whole cells, values included, to read a ballot.

The Layout

Columns

`Ledger(Value, WINDOW)` in `src/ledger.odin` is struct-of-arrays:

Field	Type	Lamport variable
`promised`	`Ballot`	`maxBal` for every decree at or above the recovery base of the promise.
`anchor`	`Trim_Anchor`	The certified prefix (POD 0003).
`slot`	`[WINDOW]Slot`	The tag: which slot owns the cell; zero is empty.
`promised_at`	`[WINDOW]Ballot`	A per-decree promise; the effective promise is `max(promised, promised_at[c])`.
`vote_ballot`	`[WINDOW]Ballot`	`maxVBal`.
`state`	`[WINDOW]Cell_State`	`Empty`, `Voted`, or `Chosen`.
`value`	`[WINDOW]Value`	`maxVal`, or the chosen value.
`used`, `chosen`	`Bit_Set(WINDOW)`	Bitmaps over cells: has a vote or decision; has a decision.

A cell's index is `cell_of(slot, WINDOW) = (slot - 1) & (WINDOW - 1)`. `ledger_open` retags a cell for a new slot and clears its ballots, state, and bitmap bits, leaving the value storage to be overwritten by the next vote. `ledger_record_vote` and `ledger_record_chosen` are the only writers of `state` and the bitmaps, so the invariant "`used` iff `state != .Empty`, `chosen` iff `state == .Chosen`" is local to two `#force_inline` procedures and is checked by `node_assert_valid` when `INVARIANT_CHECKS` is on.

Bitmaps and tags

`Bit_Set(N)` is an array of native `bit_set[0..<64]` words. `bit_set_next(bs, from)` masks the word that holds `from` and counts trailing zeros, then walks words; `bit_set_last` counts leading zeros from the top. Every window walk in the core (`on_prepare`, `on_learn`, `resend_to`, `ledger_highest_used`, `replicated_log_pending_stop_sign`, `replicated_log_observe_durable`) walks a bitmap and touches only the cells it selects, then reads only the columns it needs. The slot tag makes reuse safe: a cell that held slot 5 and now holds slot 261 answers `ledger_cell(l, 5)` with false, so a stale message about slot 5 cannot read slot 261's vote.

Ballots

`Ballot :: distinct u64` packs `round` (40 bits), `priority` (8 bits), and `node` (16 bits), most significant first, so the lexicographic order of "The Part-Time Parliament" is integer comparison and `max` over a column is a plain reduction (`ledger_highest_ballot`). Every ballot column is eight bytes per cell.

Volatile columns

`Node` keeps its phase-one evidence (`recovered_slot`, `recovered_ballot`, `recovered_state`, `recovered_value`) and its phase-two bookkeeping (`lead_slot`, `lead_ballot`, `acknowledgements`, `acknowledged`) as columns in separate index domains: phase one is relative to `recover_base` within one chunk; phase two uses the ledger window cell. The proposal a leader is driving is its own vote in the ledger; there is no second copy of the value.

The Pointer Contract

`Promise_Message`, `Accept_Message`, `Commit_Message`, `Write_Vote`, `Write_Chosen`, and `Committed` carry `value: ^Value`.

- **Outbound.** The pointer refers into the producing node's ledger (`&l.value[cell]`) or, for a decision released past the window edge, into `node.pass_through`. It is valid until that node's next transition, which is exactly the interval in which the host must persist the writes and serialise the messages under the durability rule of POD 0003.
- **Inbound.** The host points `message.value` at the decoded value for the duration of the `step` call; the node dereferences it and stores what it keeps (`on_accept`, `on_promise`, and `on_commit` all read `msg.value^` before the call returns).
- **In-process transports.** A queue that holds envelopes across transitions copies the value at enqueue time: `Packet{envelope, value}`, `packet_of` (using `message_value` to find the payload), and `packet_envelope` in `tests/harness.odin`, `examples/counter.odin`, `sim/simulation.odin`, and `bench/main.odin`. A journal kept in memory does the same with `Journal_Record` / `Sim_Record` and repoints the record at its copy before `ledger_replay_fold`.

The contract is stated on `Write`, on the message types, and on `Committed` in the source, and `tests/harness.odin` opens with it.

Capacity and Memory

The sizes below are `size_of` results from the current sources on a 64-bit build.

Per window cell

A ledger cell costs three 8-byte columns (`slot`, `promised_at`, `vote_ballot`), one byte of `state`, `size_of(Value)` of storage, and two bits of bitmaps: $25 + \text{size\_of}(\text{Value})$ bytes plus $\frac{1}{4}$ byte. The fixed part is `promised` (8) and `anchor` (16).

Configuration	Bytes per cell	`size_of(Ledger)`
`Ledger(u64, 256)`	33 + 1/4	8,536
`Ledger([128]u64, 256)` (1 KiB values)	1,049 + 1/4	268,632

`Node` adds phase-two columns per window cell: `lead_slot` (8 bytes), `lead_ballot` (8), `acknowledged` (4), and `acknowledgements` (8 bytes per 64 members, rounded up). Recovery adds `CHUNK_SLOTS * (17 + size_of(Value))` bytes of slot, ballot, state and value columns, plus `MAX_MEMBERS * ceil(CHUNK_SLOTS / 64) * 8` bytes for `promise_seen`. Alignment and scalar fields add overhead; use `tools/memory_report.odin` and the archived before/after CSVs for exact target-specific sizes.

Per message and record

`Message(Value)` is 64 bytes and `Envelope(Value)` 72 bytes for every `Value` (the largest variant is `Promise_Range_Message`); `Write(Value)` is 32 bytes; `Committed(Value)` is 16 bytes. `Effects` capacities are therefore independent of the payload type: `size_of(Effects(u64, 7, 256, 64))` is 41,840 bytes, and the benchmark's `Effects(u64, 3, 4096, 256)` is 137,904 bytes.

Historical Measurements

The in-memory benchmark (`bench/main.odin`, recorded by `make bench-compare` into `bench/results/latest.json`) measures the cost per committed value for three and five voters, 8-byte and 1 KiB values, synchronous, pipelined, and batched proposals, and the ownership mode.

- **1 KiB values.** The workload that copied the value at every hop now copies it once per acceptor (into the ledger) and once per in-process hop (into the packet). The benchmark measures the change; the recorded results file is the source of the figures, and POD 0007 quotes them once it is recorded.
- **Three voters, 8-byte values.** No change worth reporting is expected or claimed. With an 8-byte value the copy was already the size of the pointer that replaced it, and the per-value cost is dominated by the transition logic and the in-process queue, not by the layout. The benchmark measures this too; see the results file.
- **Scans.** Phase-one answers, catch-up answers, and retransmission now walk a bitmap and read ballot columns rather than whole cells. This initial run did not isolate them. The September 17 follow-up below adds recovery and retransmission profiles.

No benchmark number is typed into this record. The book and the README read their tables from the recorded file.

Where the remaining cost is

A callgrind run of the three-voter, 8-byte workload on the recording host (`bench/main.odin --only=u64-3n`, optimized build with `PAXOS_INVARIANT_CHECKS=false`) gives the instruction budget behind the 8-byte rows. One committed value is seven transitions: the leader's `node_propose`, two `on_accept`, two `on_accepted`, and two `on_commit`. Together they execute about 1,200 instructions, roughly 170 per transition; the in-process harness (`Packet` copies in and out of its queue, the queue itself, sampling) adds about a quarter of the program's instructions on top. The benchmark's nanoseconds per value therefore correspond to several instructions per cycle: the path is instruction-bound, not memory-bound, which is what the layout was meant to achieve.

Inside the library the instructions are spread thin rather than concentrated. The largest single items are the copies of `Envelope(Value)` into the effects buffer (72 bytes each, six per value), the union dispatch in `node_step` and `message_decided_through`, the per-cell ledger checks in `on_accept` and `record_commit`, and the one membership lookup per message. Two changes made after the profile removed a second lookup per message (`Node.self_index`, and the sender's index passed from `node_step` to the handlers that need it) and replaced the four two-branch `small_array.push_back` calls with a one-branch write into inline storage; both simplify the code and neither moved the instruction count by more than noise, which is the evidence that no single hot spot is left.

What would move it, and why it is not done here:

- **A smaller `Message` union.** `Promise_Range_Message` (56 bytes) sets the size of every envelope. Packing its slot range or trimming `Trim_Anchor` would take the union to 40 bytes and every envelope copy with it. It changes the wire types the host serialises, so it is a 0.3.0 decision, not a patch.
- **Inline values for small `Value` types.** The pointer contract is a clear win at 1 KiB and a small loss at 8 bytes, where the pointer is the size of the value and the host's copy at the queue is pure overhead. Odin has no conditional struct field, so a `Value`-size-dependent message layout would need two message families; the uniform contract was preferred (POD 0005).
- **The harness.** Its copies are the cost a real transport pays at serialisation and cannot be skipped without lying about pointer validity. A comparison harness that keeps values inline in its queue would flatter this library's 8-byte rows and nothing else.

The Odin features the layout does lean on are the ones the profile shows paying for themselves: struct-of-arrays columns and fixed `bit_set` words for the window (`bit_set_next` is a `count_trailing_zeros` loop), `distinct u64` ballots compared as integers, `#force_inline` on the cell and effect helpers, `#no_bounds_check` where the index was just checked, `@(cold)` on the failure path, `#config` for the invariant walks, and inline `small_array` storage so a transition touches no allocator.

Constraints the Layout Imposes

- **Power-of-two window.** `cell_of` is a mask, so `WINDOW_SLOTS` must be a power of two; `node_init` rejects anything else at compile time (`window_not_power_of_two` fixture). `0.1.0` accepted any positive window with a modulo.
- **Comparable, fixed-size values.** `Value` must satisfy `intrinsics.type_is_comparable` (the ledger compares values with `==` to detect `.Conflicting_Value` and `.Conflicting_Commit`) and is stored inline in `[WINDOW]Value`, so a variable-size payload must be a fixed-size type such as `[128]u64` or a struct with bounded fields. A comparable value that borrows host storage, such as a string, must keep that storage valid and immutable while the ledger references it. Slices do not satisfy this comparability requirement.
- **16-bit node ids.** `Node_Id :: u16` so the node field fits the packed ballot; `MAX_SUPPORTED_MEMBERS` is 65,535 and member positions use `u16`. Membership stores ids in ascending order and binary-searches that array; the former `by_id` array has been removed.
- **One transition between produce and consume.** The pointer contract holds only while the host runs one transition at a time per node and consumes or copies the batch before the next. This is the same discipline the durability rule already required.

Alternatives Considered

- **Type-erased value storage.** Storing values as `[]u8` with a size parameter would have allowed variable-size payloads and decoupled the ledger from `Value`, but it would have moved equality and copying into the host, made `Committed` and the records carry a length, and removed the compile-time check that every peer agrees on the payload type. The parametric `[WINDOW]Value` keeps `==`, `size_of`, and the type mismatch errors in the compiler.
- **A `Log_Effects` wrapper.** A separate effects type for `Replicated_Log_Node` that unwrapped `Entry` values into commands and stop signs for the host was considered again during the redesign, for the same reason as in POD 0005: a friendlier committed entry. It was rejected again because it would need a second copy of every released entry (the wrapper cannot point into the core's `Entry` and present a `Value`), which is the copy this record removes. The host reads `committed.value^` as an `Entry` and switches on it.
- **Inline values with a small-value fast path.** Keeping values inline in messages for small `Value` types and switching to pointers above a threshold would have made `Message(Value)` and the host's transport code depend on `size_of(Value)`. One representation, with the copy made explicit at the transport and the journal, was chosen so that the same `Packet` idiom serves every payload.
- **Array-of-structs with a `Maybe` per field.** The `0.1.0` `Durable_Cell` with `accepted: Maybe(Accepted(Value))` and `committed: Maybe(Value)` stored two values per cell and paid for both on every scan. Columns with one `state` byte store the value once and let a scan skip the value column entirely.

Validation and Acceptance Gates

Implementation and correctness

All recovery values, metadata, and per-peer report bitmaps now scale with the recovery chunk. Ledger and phase-two state remain window-sized. Chunk-relative indexing checks bounds before subtraction; chunk rollover resets metadata without clearing payload bytes. Read-quorum selection is frozen before issuing phase-two votes, including across window-limited retries. Public procedures and durable/wire formats are unchanged.

The new tests cover absolute-slot selection, chunk sizes 1/3/8, ring crossings, duplicate and stale reports, delayed manifests, blocked recovery, large values, sparse retransmission, and late higher votes after

a partial drive. The expanded simulator exposed two harness defects (duplicate-packet borrowing and premature quiescence) and the partial-drive selection defect; all were fixed.

Validation: 79 tests in debug and optimized builds; 600 default-capacity simulations (6 million steps), plus 120 small-window/chunk-3 simulations (1.2 million steps) with majority and both flexible-quorum extremes. Contract fixtures, style, vet, the example, and benchmark smoke checks pass. The book and ledger design record compile.

Static memory

For three members, 1 KiB values, window 256/chunk 64, node plus one effects buffer decreased from **633,120 to 433,176 bytes (31.6%)**. At window 4,096/chunk 256 it decreased from **9,080,448 to 5,081,568 bytes (44.0%)**. Chunk equal to window retains the original node size. See the before/after CSV files. These figures exclude queues, application state and runtime overhead.

Matched timings

The JSON contains 90 rows: four libraries plus the preserved Odin baseline, 18 workloads, nine samples per row. Every row validates all 4,096 ordered payloads per epoch at every learner. The paired 5% regression gate passed; this does not mean every workload became faster.

Nanoseconds per completed value (median), finite in-process workload:

Recorded 20260917T073751Z on AMD Ryzen 7 5800H with Radeon Graphics. Nine samples per row; median ns per completed value; lower is better.

N	Bytes	Depth	Odin before	Odin	Zig	OmniPaxos	LibPaxos3
3	8	1	107.5	109.2	114.4	1062.1	2543.6
3	8	8	111.5	111	119	224.3	2548.5
3	8	64	108.1	111	118.3	88.1	2555
3	64	1	118.4	115.8	150.4	1228.7	2507.6
3	64	8	122.6	123.2	149.4	321.5	2586.7
3	64	64	119	121	152.4	96.3	2603
3	1024	1	357.1	300	1725.8	3251.8	3388
3	1024	8	430.3	338.1	1708.8	2246.7	3536.5
3	1024	64	469.9	349.3	1938	2450.5	3815.1
5	8	1	202.7	206.7	166	3077.3	3335.9
5	8	8	206.3	211.3	173.8	511.9	3348.7
5	8	64	204.5	209.5	177.5	153.2	3431.9
5	64	1	218.9	224.3	245.6	3269.3	3325.1
5	64	8	221.2	221.4	252.5	672.1	3404.5
5	64	64	218.2	220.8	259.5	250.4	3557.6
5	1024	1	773.3	739.2	2834.8	6517.7	5226.9
5	1024	8	848.3	792.1	2834.7	3887.5	5341.6
5	1024	64	893.8	935.2	3350.4	3170.2	5923.4

The paired median ratios show 15.8-26.3% lower cost for the three-node 1 KiB workloads. Several small-payload rows are approximately 1-3% slower; the five-node 1 KiB/depth-64 paired ratio is 1.047 with a

95% interval of 0.928-1.089. That row is not an established improvement. Zig and OmniPaxos still lead some workload categories.

These numbers are not directly interchangeable with the historical README table: the common drivers remove the journal replay mirror, use equal command counts and payloads, retain complete finite logs, and time completion. LibPaxos retains its native preexecution work; OmniPaxos retains native coalescing. No language-wide or production-service superiority is established.

Profile-guided decision

The first retry-scan experiment counted occupied cells before scanning. Callgrind instructions increased from 12,591,988 to 17,833,184 in the dedicated retransmission workload, so that implementation was discarded.

The retained one-wrap scan reduces those instructions to 10,139,090 (19.5% fewer). A paired native timing experiment reported a median ratio of 0.877, with a 95% interval of 0.820-0.905. The sparse-retry regression test verifies that a single used slot produces one retry, not repeated duplicates. The aggregate matched gate additionally checks unchanged steady-state paths.

No speculative wire batching, protocol mode, storage adapter, networking, or threading was added. Further candidates were not implemented without measured benefit.

Final memory profiles

Sampled post-exec peak resident bytes, portable profiling builds (eight epochs):

Workload	Odin	Zig	OmniPaxos	LibPaxos
3 voters, 8 B, depth 1	3,067,904	2,400,256	2,109,440	2,584,576
5 voters, 8 B, depth 1	3,657,728	3,592,192	2,162,688	3,555,328
3 voters, 1024 B, depth 64	25,354,240	52,789,248	32,935,936	27,901,952

Odin is not the minimum-RSS implementation in every row. For the large-payload row, its static node is 4,943,664 bytes versus Zig's 17,173,488 bytes; effects are 137,904 versus 5,606,920 bytes. Both use the same configured window/chunk and fixed payload. The driver capacities and native algorithm differences remain part of the comparison.

Massif separately reports allocated capacity: it excludes static/BSS storage and can exceed RSS where allocated pages remain untouched. The JSON retains both metrics; they must not be summed or used interchangeably. The compressed archive includes annotated Callgrind traces, Massif snapshots, logs, and the accepted/rejected retry experiments. Profiling elapsed times are not used as performance measurements.

Reproduction and limits

The following evidence files are under `bench/results/`:

- `recovery-matched-20260917.json`: raw samples, commands, configurations, toolchains, source/binary hashes, and paired intervals.
- `recovery-baseline.json` and `recovery-baseline.patch`: reconstructible pre-refactor source.
- `recovery-memory-before.csv` and `recovery-memory-after.csv`: static memory measurements.
- `recovery-profiles-20260917.json` and `.tar.gz`: final CPU/memory summaries and raw profiles.

The book chapter "Reproducing Measurements" (`docs/book/06_measurement_methods.typ`) gives the workload contract and Callgrind/Massif commands.

The paper argument and these tests establish reviewable evidence, not machine-checked implementation correctness. Memory comparisons must distinguish inline storage, heap allocations, and RSS. Timing measurements are host-specific and finite-horizon; they do not include storage, serialization, network delay, or application work.

References

- Lamport, Leslie. "The Part-Time Parliament." ACM TOCS, 1998.
- POD 0002: Paxos-Odin: Architecture and Pure State Machine Design.
- POD 0003: Durability Contracts, Window Reuse, and Trim Anchors.
- POD 0005: The Idiomatic Odin API Surface.
- POD 0007: Review Findings and Verification Evidence, section "Third pass".
- `bench/results/latest.json`, recorded by `make bench-compare`.