



Kladde

Specification and design documentation

Built from commit [b3c582b](#) of 3 October 2026

Contents

1	kladde	14
1.1	Project status	14
1.2	The ambition	15
1.3	Language-independent tooling	15
1.4	The sections	15
1.5	License	16
2	Specification	17
2.1	The layers	17
2.2	The compatibility contract	17
2.3	What is fixed and what is free	18
2.4	Guarantees an implementation must provide	18
2.5	Versioning	19
2.6	Status	19
3	File format	20
3.1	Integer encodings and CRCs	20
3.2	Pages	20
3.3	Page framing	20
3.3.1	How much of the framing is load-bearing	21
3.4	Header pages	21
3.4.1	The journal pointer	22
3.4.2	The consolidator state	23
3.4.3	The magic	23
3.5	Epochs	24
3.6	Versioning	24
3.7	Concurrency	24
4	Durability	25
4.1	What is guaranteed	25
4.2	Page states	25
4.2.1	The reuse rule	26
4.2.2	The truncation rule	27
4.2.3	Deallocating reusable pages	27
4.3	The flush protocol	27
4.3.1	The two invariants	28
4.3.2	Why one fsync suffices	28
4.4	Recovery / loading a kladde file	28
4.4.1	Case analysis for a power cut during flush E	29
4.5	fsync failure is fatal	29
4.6	Headroom	29
4.6.1	Detecting that the disk is full	30
4.7	What this costs	30
5	Address table	31
5.1	The idea	31

5.2	Logical level	31
5.2.1	Statement types	31
5.2.2	No conflicts within each epoch	33
5.2.3	Conflict resolution across epochs	34
5.3	Physical format	35
5.3.1	Delta encoding	35
5.3.2	Notes on the physical format	36
5.4	Bounds	36
5.5	Design directions	37
6	Allocations	38
6.1	Allocations	38
6.2	Stable ids, and why pointers are not addresses	38
6.2.1	Id recycling	39
6.2.2	Pointer encoding	39
6.3	Ownership	39
6.4	Content semantics	39
6.5	Free space	40
6.6	Reclamation is not part of this specification	40
7	Journal	41
7.1	What the journal is	41
7.2	Record kinds	41
7.2.1	Every record but Free brings its allocation into existence	42
7.2.2	Records that read	42
7.2.3	Move	42
7.3	Transactions	42
7.4	Ordering	43
7.5	Encoding	43
7.5.1	Records	43
7.5.2	Framing	43
7.5.3	Pages	44
7.5.4	The CRC chain	44
7.6	Recovery	44
7.7	The start of a session	45
7.8	Folding	45
7.8.1	Checkpoint versus commit	45
8	Schema	46
8.1	Why a file carries its schema	46
8.2	The documents	46
8.3	The guiding principle	46
8.4	Scope	47
9	Type descriptors	48
9.1	Terminology	48
9.2	The kinds	48
9.2.1	Primitive	48
9.2.2	Struct	49
9.2.3	Enum	49
9.2.4	Opaque	49
9.2.5	Array (reserved)	50
9.2.6	Pointer (reserved)	50
9.3	References and the root	50
9.4	Inline size	50
10	Canonical encoding	51
10.1	Encoding primitives	51
10.2	Descriptor encoding	51

10.3 Canonical order	51
10.4 Table encoding	52
10.5 Worked example	52
11 Fingerprints	53
11.1 The value	53
11.2 Hash function	53
11.3 What is and is not fingerprinted	53
11.4 Cycles	54
11.5 Determinism	54
11.6 As a fast path	54
12 Evolution	55
12.1 The levels	55
12.2 What is settled	55
12.3 What is TBD	56
12.4 What was considered and declined	56
12.5 Prior art	57
13 Tooling	58
13.1 The premise	58
13.2 Level 1 — storage analysis	58
13.3 Level 2 — schema inspection	58
13.4 Level 3 — structural export	59
13.5 The container problem	59
13.6 Cross-implementation testing	59
14 Conformance	60
14.1 What conformance means	60
14.2 Must	60
14.3 May	61
14.4 Must not	61
14.5 The suite	61
14.6 Open questions	62
15 Implementation	63
15.1 The documents	63
15.2 The shape of an implementation	63
15.3 The four structures	64
15.3.1 The mirror	64
15.3.2 The indexes	65
16 In-memory state	66
16.1 1. The fragment map	66
16.1.1 The fragments of an existing id exactly partition [0, size)	67
16.1.2 Where ZeroByDefault comes from	67
16.1.3 Consequences the implementation must honour	67
16.2 2. The statement slab	68
16.3 3. The allocation map	68
16.3.1 A tombstoned id keeps no allocation-map entry	69
16.4 4. The page table	70
16.5 Derived structures	70
16.6 During a flush	70
16.7 Cost	71
17 Address-table operations	72
17.1 Reading	72
17.1.1 resolve(id, probe) -> Fragment	72
17.1.2 read(id, offset, len) -> bytes	72

17.1.3	size(id) and exists(id)	73
17.2	Opening a file	73
17.2.1	Three facts about the physical format that decide the algorithm	74
17.2.2	The pipeline	74
17.2.3	Resolving one id	75
17.2.4	What the sweep leaves behind	76
17.2.5	Why run_start is the size	77
17.2.6	Why Grow needs its own rule	77
17.2.7	Four details that carry the correctness	77
17.2.8	Why a sweep, and not an intermediate tree	78
17.2.9	What falls out of emission	78
17.2.10	Cost	79
17.3	Fragment-map primitives	79
17.3.1	split(id, at)	79
17.3.2	overwrite(id, range, new: Fragment)	79
17.3.3	coalesce_around(id, range)	80
17.3.4	pin / unpin	80
17.4	Taking and stating	80
17.4.1	take(id, range, pending, dirty)	80
17.4.2	bind(stmt, page, run)	81
17.4.3	grow_size_to(id, n) and shrink_size_to(id, n)	81
17.4.4	set_anchor(id, new)	82
17.5	Mutation, at flush time	82
17.5.1	write_bytes(id, offset, len, origin)	82
17.5.2	resize(id, n), allocate(id, n), and free(id)	82
17.5.3	What the cut states for a touched id	83
17.5.4	drop_physically(stmt)	83
17.6	Page rewrites	83
17.6.1	rewrite_page(victim, dirty)	84
17.6.2	rewrite_key_range(lo, hi, dirty)	84
17.6.3	owned_fragments(stmt)	85
17.6.4	replacement_anchor(id) — the one correctness obligation	85
17.6.5	The header	85
17.6.6	Consolidating a data page	86
17.7	Worked trace	86
17.7.1	Re-allocation	87
18	Liveness accounting	88
18.1	Coverage	88
18.2	Pins	88
18.3	The droppability rule	89
18.4	Tombstones	89
18.4.1	The last tombstone	90
18.4.2	The lemma	90
18.4.3	The cascade	90
18.5	Grow is locally decidable, and Shrink is not	90
18.5.1	The anchor pin is not substitutable	91
18.5.2	A Shrink can raise the size, not only lower it	91
18.6	Emission: when a resize must write a statement	91
18.7	The residual	92
18.8	A possible extension, not adopted	92
18.9	Confirmed sound	93
18.10	Ranking	93
19	Id recycling	94
19.1	The policy	94
19.2	The refinements	94
19.3	What an earlier draft got wrong	94

20	Write-phase state	96
20.1	The log is the sole authority	96
20.2	Two derived structures	96
20.3	One id, one lifetime	97
20.4	The geometry state machine	97
20.4.1	Transitions	97
20.4.2	Repopulation after a flush	98
21	The flush	99
21.1	Phase A — the fold	99
21.1.1	Per-record rules	99
21.1.2	What falls out unasked	100
21.1.3	Representing a piece table cheaply	100
21.2	Phase B — placement	100
21.2.1	Cutting content across pages	100
21.2.2	Page contiguity buys nothing	101
21.2.3	Where the pages come from	101
21.2.4	Cutting and packing, concretely	101
21.2.5	The InLine threshold	102
21.3	Phase C — ordering	102
21.3.1	Edges	102
21.3.2	Frees-first is a preference, not an edge	103
21.3.3	Why the graph may be unnecessary	103
21.3.4	The content-blind fast path	103
21.4	The header as write buffer	104
21.5	The shape of the tree	104
21.6	Correctness: the differential oracle	105
21.7	Implementation order	105
21.8	Walk-through of an epoch	105
22	Consolidation	107
22.1	The high-level picture	107
22.1.1	Four mechanisms	107
22.1.2	Why address-table pages have two mechanisms	108
22.1.3	Description defragmentation rides the rotating window	109
22.1.4	The envelope is a maximum-subarray problem	109
22.1.5	Where it is called, and how it is executed	110
22.1.6	How a flush and consolidation compose	111
22.2	The budget	116
22.2.1	The structures behind the ranking	117
22.2.2	The budget loop	117
22.2.3	The complexity this incurs	119
22.3	Two forms of consolidation	119
22.4	Victim selection for data pages	120
22.4.1	Scoring a data page	120
22.4.2	Victims are pulled one at a time	120
22.5	Victim selection for address-table pages	121
22.5.1	The page rewrite	121
22.5.2	The rotating window	122
22.5.3	Unlinking an emptied page	122
22.6	Finding the referrers of a data page	123
22.6.1	The shape of the index	124
22.7	Outer fragmentation	124
22.7.1	Lowest page first	124
22.7.2	Truncation	124
22.7.3	Compaction mode	125
22.8	Pacing	125
22.9	The bound this buys	126
22.10	Constants still to be chosen	126

23 Consolidator state	127
23.1 Why it exists	127
23.2 What it holds	127
23.2.1 Content ages	128
23.2.2 The rotating window's position	128
23.2.3 The budget	129
23.2.4 What stays out	129
23.3 Writing it	129
23.4 Checking it	129
23.5 What the ripeness draft would add	129
24 Transactions and batches	130
24.1 Three mechanisms over one journal	130
24.1.1 Recommendations	131
24.2 Nesting	131
24.3 The buffering state machine	131
24.3.1 Transitions in detail	132
24.4 Flush triggers	133
24.4.1 Consequence for write-phase geometry	133
24.5 Checkpoint versus commit	133
24.6 Open questions	134
25 Related work	135
26 kladde-rs	137
26.1 Two audiences	137
26.2 What it is	137
26.3 The central problem	138
26.4 The documents	138
26.5 Status	138
27 Tutorial	139
27.1 Is kladde the right tool?	139
27.2 Status warning	139
28 Getting started	140
28.1 The mental model	140
28.2 Your first durable value	140
28.3 Why the guard exists	141
28.4 Nesting	141
29 The built-in containers	142
29.1 A default library, not a privileged one	142
29.2 <code>PersistableVec<T></code>	142
29.3 <code>PersistableString</code>	142
29.4 <code>PersistableHashMap<K, V></code>	143
29.5 <code>PersistableBlob<T></code>	143
29.6 What is missing	143
29.7 Choosing	144
30 Deriving your own types	145
30.1 What to import	145
30.2 Structs	146
30.3 Newtypes	146
30.4 Enums	146
30.5 Generics	147
30.6 What the derive needs	147
30.7 Non-persisted fields	148
30.8 Layout stability	148

31 Compared to serde	149
31.1 The core difference	149
31.2 What serde does better	149
31.3 What kladde does better	149
31.4 They are not exclusive	150
31.5 Choosing	150
31.6 An honest caveat	150
32 Durability and flushing	151
32.1 The two-stage model	151
32.2 What a crash costs	151
32.3 Flushing	151
32.4 When the disk fails	152
32.5 Transactions	152
32.6 Long runs of mutations	152
33 Writing a custom Persistable	154
33.1 When you need one	154
33.2 The trait	154
33.3 What to import	155
33.4 A complete example	155
33.5 Owning an allocation	157
33.6 Your descriptor	157
33.7 The obligations you are taking on	157
34 Crate layout	158
34.1 The crates	158
34.2 The parts	160
34.2.1 kladde-store — the storage layer	160
34.2.2 kladde-persist — the serialization layer	160
34.2.3 kladde-schema — type descriptors	160
34.2.4 kladde-types and kladde-derive — the type vocabulary	160
34.2.5 kladde — the entry point	160
34.3 The seam	160
34.4 Cross-cutting concerns	160
35 Pointers	162
35.1 Two types, one idea	162
35.2 Why exactly one copyable type	162
35.3 Ownership	162
35.3.1 The one hole	163
35.4 Width	163
36 Persistable and guards	164
36.1 Persistable	164
36.1.1 <code>INLINE_SIZE</code>	164
36.1.2 Location	165
36.1.3 Why store takes <code>&mut self</code>	165
36.1.4 Why load takes <code>&mut B</code>	165
36.1.5 free	165
36.2 Guards	165
36.2.1 Mutations return a <code>Result</code>	165
36.2.2 Guards are generated per type	166
36.2.3 The write/read asymmetry	166
36.3 Why an automatic flush cannot tear a mutation	166
36.4 The chain, end to end	166
36.5 What this costs	166
37 Containers	167

37.1	PersistableVec<T>	167
37.2	PersistableString	167
37.3	PersistableHashMap<K, V>	167
37.4	PersistableBlob<T>	168
37.5	What is missing	168
38	The derive macro	169
38.1	What it generates	169
38.2	The aggregate records nothing of its own	169
38.3	Primitives are Persistable too	169
38.4	Enums	170
38.5	Generics and transparency	170
38.6	Path resolution	170
38.7	What is missing	170
38.8	Testing compile failures	171
39	Freeing	172
39.1	The problem	172
39.2	The mechanism	172
39.2.1	Ordering	172
39.3	Where ownership ends	172
39.4	Partial initialization	173
39.5	The schema-level alternative	173
40	Transactions	174
40.1	RAII types	174
40.2	Behaviour on unwind	174
40.3	Open questions	175
41	Schema binding	176
41.1	Two layers	176
41.2	The binding	176
41.3	What a type declares	176
41.4	At create and at open	177
41.5	Design invariants	177
42	The store	178
42.1	Interior mutability	178
42.2	Storage	178
42.3	The mirror	178
42.4	The in-memory structures	179
42.4.1	Keys	179
42.4.2	Fragments	179
42.4.3	The statement slab	179
42.4.4	Maps keyed by id or page	179
42.5	Where the integer widths come from	179
43	Evaluation	180
43.1	The evaluations	180
44	Consolidation under load	181
44.1	What was measured	181
44.2	Space	182
44.3	The churn floor sets the fill, not the target	183
44.4	Writes	184
44.5	Mass frees	185
44.6	The description	186
44.7	Time	186
44.8	The typed layers	188

44.9	Defects the benchmarks found	188
44.10	What to change	189
44.11	Reproducing	189
45	Cleaning by ripeness	190
45.1	What was measured	190
45.2	The trade-off	191
45.3	The controller	192
45.4	At the default settings	193
45.5	Hot, cool, and cold allocations on shared pages	195
45.6	Without the logarithm	195
45.7	What it costs	197
45.8	What to change in the draft	198
45.9	Reproducing	198
46	Ripeness with a static share	200
46.1	What was measured	200
46.2	How often pages earn a static share	200
46.3	Cleaning	201
46.4	What it costs	203
46.5	What to change in the draft	204
46.6	Reproducing	204
47	Bayesian ripeness	205
47.1	What was measured	205
47.2	Cleaning	206
47.3	What the posterior calls static	207
47.4	The empirical prior	208
47.5	What it costs	210
47.6	What to change in the draft	211
47.7	Reproducing	212
48	Superseded	213
48.1	What was replaced, and by what	213
48.2	A note on measurement	213
49	The relocatable heap	215
49.1	What it was	215
49.2	The trait	215
49.2.1	The heap owned the id table	215
49.2.2	Ids came from outside	216
49.2.3	A step was a byte range, not an allocation	216
49.2.4	budget was a ranking input, not a cap	216
49.3	Sizedness	216
49.4	Errors and composition	217
49.5	Placement	217
49.6	resize could ask for two moves	217
50	Incremental compaction	218
50.1	The constraints	218
50.2	The potential	218
50.3	The candidate shapes	219
50.4	The evacuation index	219
50.4.1	The lesson worth keeping	220
50.5	Pathologies	220
50.5.1	The stance	220
50.5.2	The stress test	220
50.5.3	Remedy 1: weighting by class	220
50.5.4	Remedy 2: the lift	221

50.6	Measurement methodology	221
50.7	What was still open when it was retired	221
51	The in-place flush	223
51.1	What it was, and what it needed	223
51.2	Why it went	223
51.3	What kladde does differently from those systems	224
51.4	What was given up	224
52	Whole-allocation table entries	225
52.1	What it was	225
52.2	Why per-chunk statements were rejected first	225
52.3	The mechanisms it needed, and no longer does	226
52.3.1	Extents	226
52.3.2	Spilling	226
52.3.3	A separate index	226
52.3.4	Tombstone lifetime	226
52.4	What else survived	227
53	Drafts	228
53.1	Current drafts	228
54	Open issues	229
54.1	Contradictions	229
54.1.1	1. The journal is two different things — resolved	229
54.1.2	2. Journal pages cannot satisfy the page framing — resolved	229
54.1.3	3. The durability promise is stated unqualified where users read it	229
54.1.4	4. Framing is per-record in one place and per-transaction in another	230
54.1.5	5. mentions is claimed to have one reader and has at least three	230
54.1.6	6. Reclamation is optional and also required	230
54.1.7	7. Conversion is gone, and the flush still reasons about it	230
54.1.8	8. “Five layers”, six rows	230
54.1.9	9. Which “read” the complexity bound governs — narrowed, not closed	231
54.1.10	10. Grouping is “free” but constrained	231
54.2	Blockers for a first prototype	231
54.2.1	1. The journal’s on-file representation — resolved	231
54.2.2	2. The byte encoding of journal records — resolved	231
54.2.3	3. Bootstrapping — partly described	231
54.2.4	4. File extension is not covered by the durability argument	231
54.2.5	5. The header’s byte layout	232
54.2.6	6. The pointer encoding inside allocations	232
54.2.7	7. The in-memory address table	232
54.2.8	8. The fold, plus its differential oracle	232
54.2.9	9. Recording every mutation	232
54.2.10	10. Freeing	232
54.2.11	11. Choosing values for things that only need <i>a</i> value	232
54.2.12	Deferrable, and why	233
55	The Move operation	234
55.1	It needs no new address-table statement	234
55.2	It keeps the byte-referenced-once invariant	234
55.3	What it buys	234
55.4	Settled by the record’s semantics	235
55.5	Open questions	235
56	Segments	236
56.1	Why it is reachable at all	236
56.2	What has already been decided with segments in mind	236
56.3	What is not decided	236

57 Cleaning by ripeness	237
57.1 The problem	237
57.1.1 A mixed page drains, then freezes	237
57.1.2 Why the current design never cleans a frozen page	237
57.1.3 What LFS did about it	238
57.2 The proposal	238
57.2.1 When waiting no longer pays	238
57.2.2 A page holds a draining share and a static one	239
57.2.3 Estimating how fast a page still drains	239
57.2.4 Checking the fit in simulation	243
57.2.5 Ranking pages by ripeness	243
57.2.6 The budget, and the price	244
57.2.7 Where survivors go	245
57.3 What would change in Consolidation	247
57.4 Ablation: without the logarithm	247
57.5 Open questions	248
58 Bayesian ripeness	249
58.1 Executive summary	249
58.2 The decision under a posterior	250
58.2.1 The gain of cleaning now	250
58.2.2 Three ways to decide	250
58.3 One rate per page	250
58.3.1 Losses counted in bytes, weighed in loss events	250
58.3.2 The exact posterior is a Gamma distribution	251
58.3.3 A rate that drifts keeps the posterior exact	251
58.3.4 The single-rate draft is this posterior's mean	251
58.3.5 What a page keeps	252
58.3.6 Where a page's prior comes from	252
58.3.7 The floor	253
58.3.8 Deciding on one rate	253
58.3.9 Waiting teaches, and the drafts' rule does not see it	254
58.4 A draining share over a static one	255
58.4.1 The cure model, per chunk	255
58.4.2 The exact posterior is a mixture over the untouched chunks that drain	255
58.4.3 The prior replaces the test	256
58.4.4 A drifting rate	256
58.4.5 What a page keeps, and what a loss costs	256
58.4.6 Deciding on a mixture	256
58.4.7 How it relates to the draft's fit	257
58.5 Checking in simulation	257
58.5.1 Where a page's prior comes from, in simulation	258
58.6 What would change	259
58.7 Open questions	260
59 Sorting survivors by temperature	261
59.1 The problem	261
59.2 The proposal	261
59.2.1 Three classes	261
59.2.2 The evidence	262
59.2.3 Where each class goes	262
59.2.4 Within a class, key order	263
59.2.5 What it costs	263
59.2.6 What it gives the estimates	264
59.2.7 What to expect on kladde-bench	264
59.3 Alternatives	264
59.4 What would change	265
59.5 Open questions	265

60	Variable-size values	266
60.1	The problem	266
60.2	The proposal	267
60.2.1	Encodings: fixed and packed	267
60.2.2	Pointers, and what they point to	269
60.2.3	Which library a type belongs to, later	269
60.2.4	Small values	270
60.2.5	Who chooses, and how the choice nests	271
60.2.6	In the schema	271
60.3	What a mutation costs	272
60.3.1	The size stays the same	272
60.3.2	The size changes	272
60.3.3	In memory	273
60.4	Why it fits kladde	273
60.5	What else it touches	273
60.5.1	Containers	273
60.5.2	Evolution	274
60.5.3	Tooling	274
60.5.4	kladde-rs	274
60.6	Stages	275
60.7	Left for later: floats as decimals	275
60.7.1	What it would save	275
60.7.2	What others do	276
60.7.3	What adopting it would take	276
60.8	Alternatives	276
60.8.1	A durable Box for large variants	277
60.8.2	Struct of arrays, in the application	277
60.8.3	Compressing data pages	277
60.8.4	Zero statements for padding	277
60.9	Open questions	277

Chapter 1

kladde

Durable data structures: mutate in memory, and it's on disk. A cross-language file format and durability protocol with crash-safety guarantees, and libraries for building durable data types on top of it.

Founded and currently maintained by [Robert Bamler](#). Source on [GitHub](#); all of these pages also as [one PDF](#).

Kladde is becoming a system for *durable data structures*: containers (strings, vectors, hash maps, trees, ...) and user-defined types that behave like their ordinary in-memory counterparts, but whose every mutation is durably recorded to a file as it happens, in a way that is optimized for disk I/O. Unlike serialization formats like JSON or XML, there is no save step and no serialization pass. Unlike sqlite, there is no SQL and no object-relational layer. You open a kladde file containing some value (typically in some complex nested application-defined data type), you mutate some parts of the value in (almost) the same way you would mutate any other value, and each one of your changes is immediately and efficiently applied to the file.

The idea is known as *orthogonal persistence*: a program works with persistent data exactly as it works with transient data, and data persists by being reachable from a persistent root. Kladde brings it to existing languages as libraries, as far as each language allows.

Reads never touch the file. A durable data structure keeps a full, natively-typed in-memory representation, so reading a field or iterating a map costs what it costs in the host language, not what it costs in a database. Writes go two places at once: into that in-memory representation, and into an append-only on-disk journal that is durable by the time the mutating call returns. Whenever the journal reaches a threshold size, it is folded into the bulk part of the file, in a way that minimizes I/O and guarantees [durability under application crashes and consistency even under power outages](#).

The name is German: a *Kladde* is a merchant's rough day-book, where transactions are scribbled down in order as they happen and later transcribed into the clean *Hauptbuch*. The file format has exactly that shape.

1.1 Project status

The project is still very young and under active development. At the current stage, the **main products** are

- a [detailed file format specification](#) and
- a [documentation of a reference implementation](#).

The reference implementation itself is *not production ready*. Its purpose at the moment is to allow rapid experimentation on new *policies* (non-normative decisions that an implementation may make without breaking interoperability). This is why the reference implementation is currently limited to a single language (Rust) and why it is currently fully auto-generated from the specification.

Contributions are welcome. While this includes code contributions in principle, at the current stage, **the most valuable contributions are not in the form of code** but instead in the form of well-motivated

opinions and proposals of what should be added or changed in the specification. If you have a potential use case for a system like kladde and you think that something in its currently stated design would hold you back, then I want to hear your opinion. Please feel free to describe your situation in a [new issue](#).

1.2 The ambition

Kladde is meant to be a **cross-language system**, not a library for one language.

At the centre sits a language-independent specification of the file format and the schema description — everything all implementations must agree on. Around it sit implementations for individual languages: [kladde-rs](#) first, with `kladde-cpp`, `kladde-python`, `kladde-java` and others intended to follow.

The contract between them is deliberately narrow and deliberately strict:

A file written by any conforming implementation can be opened by any other conforming implementation, with no conversion step and no special treatment, as long as the opening application implements equivalent data structures.

Everything the specification does not pin down is free to vary, and implementations are expected to differ substantially beyond the normative part (e.g., how aggressively they consolidate free space). Each language should get an API that is idiomatic for that language rather than a transliteration of the Rust one. For example, the Rust implementation of kladde needs an explicit `Guard` type to intercept field assignments; by contrast, a Python implementation of kladde should intercept field assignments via property hooks so that `doc.title = "..."` simply persists.

1.3 Language-independent tooling

A file format that several languages can read is also a file format that *tools* can read, without knowing anything about the application that wrote it. A generic tool can enumerate allocations and measure fragmentation without interpreting a byte of application data, read the file's embedded type descriptors to report what types it contains, and walk the value graph from the root to render it as text. See [Tooling](#) for how far each of those goes.

1.4 The sections

section	what it holds	who it is for
Specification	the normative, language-independent description: what a kladde file is, what invariants it must uphold, and what guarantees an implementation must provide	implementers porting kladde to a new language; tool authors
Implementation	the reference algorithms and data structures — what must be maintained in memory to satisfy the spec, and how — stated in language-agnostic pseudocode	anyone building an implementation in any language
kladde-rs	everything specific to the Rust implementation: crate layout, traits, guards, macros, and the Rust-specific half of the algorithms	Rust users, and Rust contributors
Evaluation	measurements of kladde-rs on realistic workloads, and what they say about the design	anyone judging the design or choosing its constants
Superseded	designs that were worked out and then rejected, kept for the reasoning	anyone tempted to re-propose one of them
Drafts	half-baked ideas in progress, not yet held to the separation the other sections observe	the authors

The three-way split between **spec**, **impl**, and **rust** is the organising idea, and the test for which one a statement belongs in is:

- Could two implementations disagree about it and still read each other's files? If no, it is **spec**.
- Would a Python or C++ implementation make substantially the same choice? If yes, it is **impl**.
- Otherwise it is **rust**.

1.5 License

Everything here — text, figures, and data — is available under your choice of CC BY 4.0, MIT, Apache 2.0, or the Boost Software License 1.0; the [repository](#) has the details.

Patent pledge. I, Robert Bamler, will not assert any patent I own or control against any implementation of the kladde specification.

Chapter 2

Specification

The normative, language-independent description of kladde: what a file is, what invariants it must uphold, and what guarantees an implementation must provide.

Two audiences: **implementers** porting kladde to a new language, and **tool authors** writing something that reads kladde files without knowing the application that wrote them. The [reference algorithms](#) are a worked example of one way to satisfy this contract, not part of it.

Documents here state *why* as well as *what*. A specification that records only the rule invites reimplementations of the mistake it was written to avoid, so each decision below carries a short motivation — briefly, and clearly marked as motivation rather than requirement.

2.1 The layers

A kladde file is described by five layers, from the bytes upward.

layer	defines	document
Pages	the page grid, page framing, the two header slots, and epochs	File format
Durability	which pages a write may touch, the flush protocol, and what survives a crash	Durability
Address table	the statements that map allocation ids to sizes and content, and how they are resolved	Address table
Allocations	what an allocation is, how it is named, how one refers to another, and what its bytes mean	Allocations
Journal	the record set for durable mutation, transactions, framing, and recovery	Journal
Schema	how a value type’s byte layout is described, encoded, fingerprinted, and evolved	Schema

Above these sits the application’s own data, whose meaning kladde does not define.

2.2 The compatibility contract

A file written by any conforming implementation can be opened by any other conforming implementation, with no conversion step, as long as the opening application implements equivalent data structures.

“Equivalent data structures” is doing real work in that sentence. Kladde guarantees that the *representation* round-trips: an implementation that reads a file finds the same allocations, the same pointer graph, and the same declared types it would have found had it written the file itself. It does not guarantee that an application which has never heard of a type can do anything useful with values of that type. See [Schema](#) for exactly where that line falls, and [Tooling](#) for what a type-ignorant reader can still do.

2.3 What is fixed and what is free

Fixed here, and therefore identical in every implementation:

- the page grid, page framing, and the two-slot header commit;
- the reuse rule and the flush protocol that make durability a guarantee rather than a hope;
- the address-table statement types and the rules that resolve them;
- the journal record encoding and the rules for recovering a torn tail;
- the type-descriptor model, its canonical byte encoding, and the fingerprint computation.

Explicitly **not** fixed, and expected to vary:

- **placement** — which reusable page a flush writes to, and how it cuts content across pages;
- **consolidation** — whether an implementation reclaims garbage at all, and by what policy;
- **when** a flush happens, and how aggressively it optimizes what it writes;
- **batching** — how an implementation joins subsequent application-level operations or transactions into larger library-level transactions when the application temporarily allows the library to do so (useful during large batch operations).
- everything above the storage layer: the API shape, the mutation mechanism, the container implementations. These decisions should follow conventions of the host language to make a kladde implementation as idiomatic as possible, and this specification is language agnostic.

The test for whether something belongs in the fixed column is simple: *could two implementations disagree about it and still read each other's files?* If yes, it stays free.

2.4 Guarantees an implementation must provide

Beyond byte-level agreement, a conforming implementation owes the application four things.

Durability. When a mutating call returns, the mutation survives an application crash. When `flush()` returns, every transaction it folded survives a power cut. Recovery always reaches a transaction boundary — never a partial transaction, and never a partially applied flush. The details, and the one thing that is *not* promised (transactions issued after the last completed flush survive only as far as the operating system happened to write them out), are in [Durability](#).

Bounded sizes. An implementation must support the [bounds](#) the format states — allocation ids, allocation sizes, page numbers, file size — and must fail cleanly rather than silently wrap when an application exceeds them.

Asymptotic complexity. The format is designed so that these are achievable, and an implementation that misses them is conforming but not useful:

operation	required
read any byte of any allocation	$O(\log F)$ in the number of live fragments
allocation size query	$O(1)$
a mutation, in memory	$O(\log F)$ per contiguous range touched
a flush	$O((k + s) \cdot \log F)$ for k dirty ranges and s statements rewritten, plus the pages it writes
opening a file	$O(n)$ in the file's live bytes
a non-mutating operation once the file is loaded	no overhead over a comparable non-persisted data type

Nothing may require a scan of the file at run time, and nothing may require a stop-the-world pause: reclamation is incremental by construction, and a flush's work is bounded by a budget the implementation chooses.

Crash atomicity of transactions. A transaction is all-or-nothing under both an application crash and a power cut, and ordering is preserved: if a transaction survives a crash or power cut, then all previous transactions survive too, in their recorded order. These are the guarantees application authors build on directly, and they are why transactions — unlike batches — are part of this specification.

2.5 Versioning

The specification carries a version. A file records both the version it was written with and the minimum version required to read it, so that an implementation can distinguish “written by something newer, but still readable” from “written by something newer that used a feature I do not have.”

The details are in [File format](#).

2.6 Status

Draft. No part of this specification is frozen.

The schema layer is the most settled: it is specified precisely enough to implement, and has a reference implementation. The page, durability, and address-table layers are settled in design and worked out in detail, but their byte-level field lists are not final. The journal is at the same stage, byte encoding and page layout included.

Chapter 3

File format

The page layer: how a kladde file is divided, what every page carries, how a file is identified, and how versions are negotiated.

Status: draft. The shape is settled; exact field widths and offsets are TBD.

3.1 Integer encodings and CRCs

All integer fields in kladde files are unsigned. For every integer field, the file format specification states either a fixed width (in bytes) or a varint encoding. Fixed width integers are encoded in little-endian byte order. Varints are encoded as unsigned [LEB128](#).

CRC calculations use **CRC-32C** (Castagnoli), specified precisely as: reflected polynomial 0x82F63B78, initial value 0xFFFFFFFF, input and output both reflected, and a final XOR with 0xFFFFFFFF. This is the variant Btrfs, ext4 metadata, iSCSI and SCTP use, and the one both x86-64 (since SSE4.2) and ARMv8 compute with a single instruction. As a check value, the CRC-32C of the nine ASCII bytes 123456789 is 0xE3069283, which a kladde file stores — little-endian, like every other fixed-width field — as the four bytes 83 92 06 E3.

3.2 Pages

A kladde file is a sequence of fixed-size **pages**. The page size is recorded in the file header as a binary logarithm and is uniform throughout a file. Currently only 4 KiB is allowed; 8, 16, 32, and 64 KiB are reserved, and the bounds elsewhere are derived from a 64 KiB ceiling so that nothing has to be revisited if a larger page is ever wanted.

Why fixed pages, and why 4 KiB by default. The operating system rewrites a whole page even when the application modifies one byte of it, so the page is the honest unit of I/O; accounting in anything smaller measures a cost the file system does not charge. 4 KiB matches the write-back granularity of common file systems. 16 KiB may pay on platforms with 16 KiB native pages, which is a measurement question rather than a design one — nothing in this specification depends on the value.

Outside of journal appends, an implementation **writes whole pages only**, and only to pages that are reusable under the [reuse rule](#). A page must also consist of whole blocks of the file system that stores the file, which the reuse rule [relies on](#); 4 KiB pages do on common file systems.

3.3 Page framing

Every page except the pages that make up the current journal concatenates the following fields, in order, without delimiters:

field	width	meaning
header kind	0 unless it is a header page 1 byte	only present in header pages one of AddressTable (0x01) or Data (0x02). For header pages: always AddressTable (0x01)
epoch	8 bytes	the flush counter at the time the page was written
content_size	2 bytes	the size of content in bytes
content	content_size bytes	the payload of the page, encoded depending on kind
crc	4 bytes	checksum over all preceding bytes of the page except content_size (but including header)
padding	to the page boundary	arbitrary, excluded from the CRC, present even on the file's last page

A page whose CRC does not validate **must be ignored** during loading. This is safe because the durability protocol guarantees that no committed state ever references a page whose write did not complete, so an invalid CRC can only belong to garbage that nothing references. The CRC calculation excludes content_size so that a writer may encode a page and calculate its CRC before knowing the page size if this turns out to be easier in a given situation. This is safe because content_size tells the reader where to stop decoding content and expect the crc, so if the CRC at that position matches then content_size is correct. A reader must nevertheless **bounds-check content_size before using it**, since a corrupted value can point the CRC read past the end of the page; a value that does not leave room for the crc within the page is itself a failed validation.

MAX_PAGE_CONTENT — the largest content a non-header page can hold — is the page size minus the 15 bytes of framing above, i.e. **4081 bytes** for 4 KiB pages. A header page holds correspondingly less, by the width of its header field.

There is deliberately **no Free kind and no free marker**. Liveness is not recorded in a page; it is defined by reachability from a committed header.

3.3.1 How much of the framing is load-bearing

Less than it looks, and the distinction matters to anyone reasoning about the crash argument.

Only two checksums in the file are load-bearing: the **header** CRCs, because a header is written without a covering fsync before it matters, so a torn or missing header write must be detectable; and the **journal's** per-transaction CRC chain, because journal appends are likewise never fsynced before a power cut can hit them. The framing on Data and AddressTable pages is *not* load-bearing: [invariant I1](#) guarantees that any page a valid header can reach was fsynced before that header was written, so recovery never meets a referenced page whose write did not complete.

The framing is required on every page anyway, as defence in depth. The crc turns *later*, silent damage — bit rot, a misdirected write by other software, a bug that writes to a live page — into a detected failure instead of quietly wrong data. The epoch is an end-to-end assertion about the fsync contract. And kind with content_size keeps every framed page self-describing, which keeps a last-resort scavenger possible and keeps the page writer uniform. Journal pages carry no framing, so a scavenger cannot identify them — which costs nothing, since a scavenger's job is to recover committed state and the journal is by definition what has not been committed. The price is 15 bytes in 4096, or 0.37 %.

3.4 Header pages

Page 0 must always be present and it is always a **header page**, and page 1 must also be a header page if it is present (i.e., if the file is larger than 1 page). Page 0 must always have an even epoch field, and if page 1 is present then it must have an odd epoch field.

Each header page holds a fixed-size header field in the [page framing](#), which concatenates the following fields with no delimiters:

field	width	purpose
magic	11 bytes	identifies the file type as kladde and catches accidental opens of unrelated files; present even in page 1 to simplify size calculations. See below
log2_page_size	1 byte	binary logarithm of the page size; currently, the only allowed value is 12, indicating a 4 KiB page size, but the spec is designed not to prevent 13, 14, 15, or 16 (for powers of 2 from 8 KiB through 64 KiB) in the future if measurements deem them useful.
format_version	2 bytes	the specification version this file was written against; currently only version 0 is allowed, indicating “pre-stable”.
min_reader_version	2 bytes	the oldest specification version that can still read this file; currently only version 0 is allowed, and it stays 0 until kladde-rs is declared production ready: until then the format promises no compatibility, changes without raising it, and a file written by one pre-release need not open in another
root_allocation	4 bytes	ID of the allocation holding the root value
schema_table	4 bytes	ID of the allocation holding the descriptor table
consolidator_state	4 bytes	ID of an allocation in which an implementation keeps state of its own from one session to the next, or 0 if there is none. See The consolidator state .
root_fingerprint	16 bytes	the schema fingerprint of the root type
journal_pointer	4 bytes	the page number of the page holding the beginning of the journal that will be used after the flush that created this header page. See The journal pointer .

The header page is the root [address-table](#) page, so the address table is reached without indirection, and the commit that publishes a new header publishes a new root table page in the same write.

The two header pages are written alternately: the flush with epoch E writes header slot $E \bmod 2$. On open, a reader reads both header pages (if both are present), picks the CRC-valid one with higher epoch, and ignores the other header page. A torn header write can only damage the slot being written, which held the older of the two headers, so the previous state remains reachable through the other slot.

Why two alternating slots. This is the only fixed-location, overwritten-in-place structure in the file, and it is the minimum needed to publish a new state atomically without an allocation protocol for the root itself. It is LMDB’s meta-page scheme. Both slots corrupting simultaneously is unrecoverable without a scan, but they are single-sector writes at opposite ends of a two-page span and are never written in the same flush, so that requires two independent failures.

The **root fingerprint** is in the header rather than only in the schema table so that the common case — an application opening a file it wrote itself, with an unchanged schema — is a single 16-byte comparison with no need to parse the descriptor table at all. See [Fingerprints](#).

3.4.1 The journal pointer

The `journal_pointer` field of the header must be ≥ 2 and either point past the file or reference a page that would not be in either header’s world were it not for this pointer (also not part of the other header slot’s journal). If the page exists then a reader will try to decode it as a journal page assuming an epoch that is one higher than the header page’s epoch and consider the largest CRC-valid prefix (which may be empty) as the current journal, following links to subsequent journal pages if present.

If the `journal_pointer` points past the end of the file then the journal is considered empty, and the first journal page must be created before recording the first transaction in the journal — as a whole page, since

the file [always ends at a page boundary](#). Implementations should avoid writing a `journal_pointer` that points unnecessarily far past the end of the file since this would cause unnecessary file growth once the first transaction is recorded. However, readers must not assume that `journal_pointer` points at most directly after the file since violating this assumption is unavoidable in edge cases: a flush may create a file that ends in a fallback page and set `journal_pointer` to point to the page immediately past the file. Closing and reopening the file introduces another `fsync`, which transitions the last page from fallback to reusable, at which point the implementation is allowed to truncate the file past it, resulting in a file whose `journal_pointer` seemingly points unnecessarily far past the end. Readers must be able to handle such a file.

3.4.2 The consolidator state

consolidator_state names an allocation in which an implementation may keep whatever it wants to carry from one session to the next, or is 0 if there is none. Its content is up to the implementation. A consolidator’s heuristics, for instance, learn from what happens over many flushes, and a session that could start only from what the file records anyway would forget all of it at every close. [Consolidator state](#) describes what the reference implementation keeps there.

Four rules govern it:

- **Nothing in it affects what the file contains.** A reader that ignores the field resolves every allocation to the same content, and nothing reachable from the root names the allocation.
- **In every other respect it is an allocation like any other.** The field owns it, as `schema_table` owns the descriptor table, so it stays alive for as long as the field names it; and it never has id 0, which the field reserves for “none”.
- **Any writer may keep it, replace it, or clear the field.** A writer that keeps it keeps its id and content, as it would any allocation’s, whether or not it understands them; a writer that replaces it or clears the field frees the allocation the field named.
- **It may be stale, or another implementation’s.** Flushes by a writer that kept it without maintaining it, or by its owner if the owner updates it lazily, leave it describing an older state of the file. So an implementation must treat a state it does not recognize as its own as absent, and must check what it does recognize against the file before relying on it.

3.4.3 The magic

8B 4B 4C 41 44 44 45 0D 0A 1A 0A \x8B K L A D D E \r \n \x1A \n

Eleven bytes, following the pattern PNG established and HDF5 copied, because every byte of it does a job:

bytes	purpose
8B	high bit set, so any tool applying a “does this look like text?” heuristic classifies the file as binary immediately. Not 89, which is PNG’s and HDF5’s, so that a partial match cannot be mistaken for either
4B 4C 41 44 44 45	KLADDE in ASCII, so a human running <code>xxd -l head</code> or <code>strings</code> sees what it is
0D 0A	catches a transfer that mangled CRLF→LF: the pair arrives as a lone 0A and the magic fails
1A	DOS end-of-file, so <code>type file</code> on Windows stops here instead of spewing the whole file
0A	catches the reverse mangling, LF→CRLF: this byte arrives as 0D 0A and the magic fails

`file(1)` has no entry for it, so a `kladde` file reports as `data` — which is the desired outcome, and better than a wrong guess. Submitting an entry to the `file` magic database is worth doing once the format is frozen, not before.

3.5 Epochs

The **epoch** is a monotone counter, incremented by one per flush, stored in every page and in each header.

It serves three purposes:

1. It **orders contradicting statements** between address-table pages, which is what makes shadowing work without a rewrite ([Address table](#)).
2. It **salts the journal's CRC chain**, so stale bytes in a reused page can never impersonate valid journal content.
3. It is an **end-to-end assertion**: a page reachable from the header of epoch E must carry an epoch $\leq E$. A violation reveals that the platform broke the `fsync` contract, and the file must be treated as damaged rather than silently misread.

Epochs are 64-bit. At one flush per millisecond that lasts half a billion years, so **wrap-around is not a concern** and no implementation may assume it must handle one.

Why not 32 bits. A 32-bit counter would require actively rewriting laggard pages before the counter catches up to them — exactly PostgreSQL's transaction-id wraparound “freezing”, an operational burden it carries only because its format predates the lesson. A new format should pay the four extra bytes per page; they are 0.1 % of a 4 KiB page, and modern systems (ZFS transaction groups, LMDB transaction ids) are all 64-bit.

Epochs are **replay-stable**: the epoch of a flush is the committed header's epoch plus one, so a flush re-run during recovery reproduces the same epoch.

3.6 Versioning

Two version numbers, because “can I read this?” and “was this written by something I know?” are different questions.

- **format version** — what the writer used. Informational for a reader; useful for diagnostics and for deciding whether to rewrite the file in a newer form.
- **minimum reader version** — the writer's declaration of the oldest reader that can still make sense of the file. A reader whose own version is below this **must** refuse to open the file rather than attempt a partial interpretation.

A writer raises the minimum reader version only when it uses a feature that older readers would silently misinterpret. Adding a new [primitive code](#), for instance, does raise it, because an older reader would not know the width of the new primitive and would compute every subsequent offset wrongly. Reordering the descriptor table does not, because references are table-local and any conforming reader follows them.

This is the same slot a per-file *application* semantic-versioning scheme would use. Whether kladde exposes application versioning through the same mechanism or a separate one is **TBD**; it should be one mechanism, not two parallel ones.

3.7 Concurrency

The format assumes a **single process with exclusive write access**. Multiple concurrent writers, and readers in other processes observing a file being written, are out of scope.

This is a scoping decision, not a structural one. The single-owner pointer discipline and the copy-on-write commit are both compatible with an MVCC-style snapshot-isolation scheme, which is the direction a concurrent version would take. **TBD**.

Chapter 4

Durability

Which pages a write may touch, the protocol that commits a flush, and exactly what survives a crash.

This is the layer that makes durability a **guarantee by prevention** rather than a detection scheme: committed state is never overwritten, so there is nothing to detect.

4.1 What is guaranteed

Under an application crash — the process dies, the operating system survives — every mutation whose call returned successfully survives. The journal's pages remain in the OS page cache and are written back on the OS's own schedule, which an application crash does not interrupt.

Under a reported error the same holds, for a simpler reason: the process is still running. An implementation that cannot complete an operation reports the failure and refuses further work — in Rust by returning `Result::Err` and poisoning the `Kladde<T>`, in C++ by throwing — and every mutation that returned successfully before that is in the journal exactly as if nothing had gone wrong. The exception is a [failed fsync](#), after which the page cache may have discarded writes the kernel had already reported as clean; that is why a failed fsync ends the session instead of being reported as an ordinary error.

Under a power cut, recovery reaches the state of some **transaction boundary**, including at least every transaction folded by the last completed `flush()`. Transactions issued after that flush survive up to a valid journal prefix, minus whatever the operating system never wrote out.

That last clause is the one worth reading twice. Journal appends are not fsynced individually, so they sit in the page cache until write-back; a power cut freezes whichever subset of journal pages the OS happened to have written. Write-back is **not ordered**, so a later journal page can be on disk while an earlier one is not. The chained, epoch-salted transaction CRCs deliberately refuse to skip such a hole: what recovery keeps is the longest valid *prefix* ending at a transaction boundary, which can be shorter than the raw journal bytes that survived.

Never guaranteed: that a partial transaction is visible, or that a flush is partially applied. Both are excluded by construction rather than by cleanup.

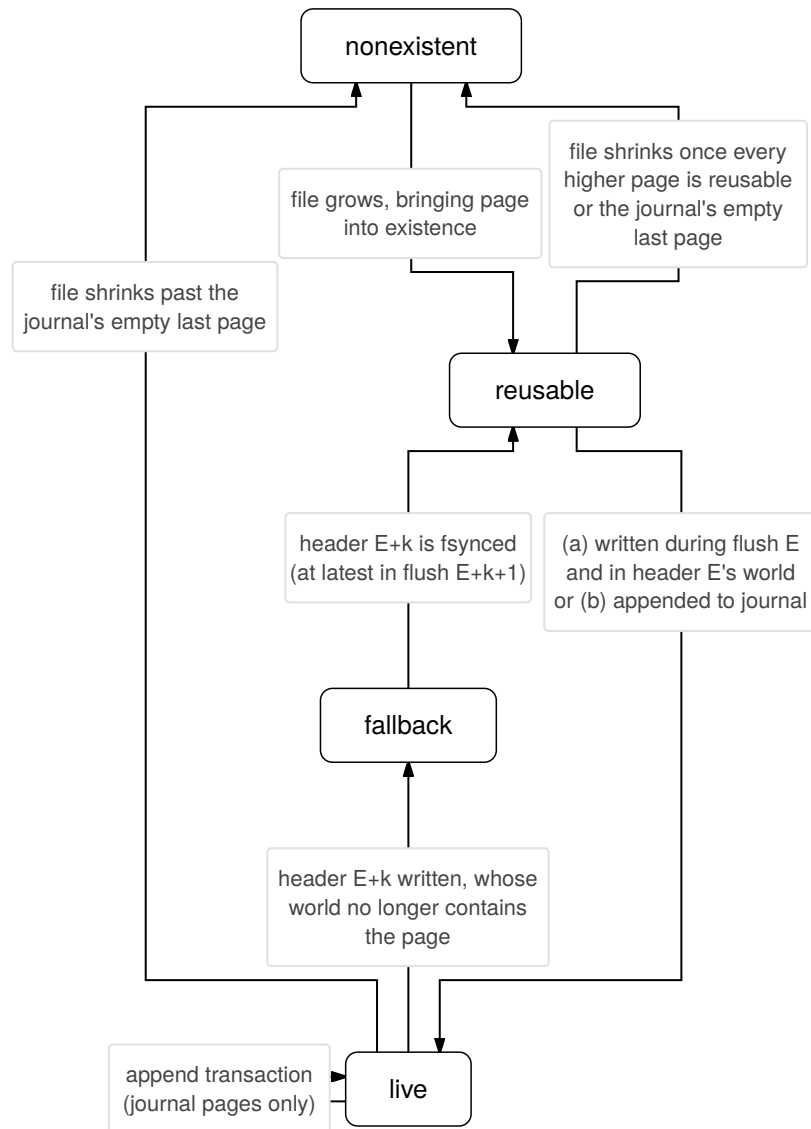
4.2 Page states

Page states are **derived, not stored**.

Define the **world** of a header as everything reachable from it: its address-table pages, the data pages those reference, and every page of the *next* epoch's **journal segment**, of which the header names only the first page; the rest are reached from it. At any moment the two on-disk header slots define at most two worlds that can (and typically do) overlap, and every page is in exactly one of three states:

- **live** — in the world of the newer on-disk header;
- **fallback** — in the world of the older on-disk header but not in the world of the newer on-disk header, and the newer header hasn't been fsynced yet;

- **reusable** — in neither on-disk header’s world, or only in the older header’s world after the newer header has been fsynced.



4.2.1 The reuse rule

A flush may write only to reusable pages, or extend the file.

The whole durability argument rests on this. It means a page that the commit of epoch E stopped referencing becomes writable in flush $E + 2$: it stays fallback until header E is durable, and within a session the fsync that makes it so is that of flush $E + 1$, which comes only after that flush has written its pages.

The same rule covers reopening a file: whatever headers are on disk define the worlds to respect. A load cannot know whether the governing header is durable, so the older world’s pages stay fallback until an fsync makes it so; [the fsync at open](#) does, and retires the older world at once.

The rule assumes that every page consists of whole blocks of the file system. A file system that overwrites a block in place writes all of it, and a power cut during that write can corrupt all of it, even the bytes that did not change. Were a block to span two pages, writing one of them — or appending to the journal’s last page — could therefore damage the other, which is exactly what the rule exists to prevent.

4 KiB pages consist of whole blocks on common file systems; a file system with larger blocks needs pages at least as large.

4.2.2 The truncation rule

A file may shrink only past pages that are reusable, and only to a page boundary.

Truncation is the other way a page can be lost, and it answers to the same argument as reuse: no header that recovery could choose may reach past the new end. The one exception is the current journal's last page while no transaction has been appended to it: the file may shrink past that page too, since it holds nothing to lose and a reader treats [a journal page past the end of the file](#) as empty as well. So the file can end at its last page of content, which suits the last flush before a file is closed, and an application that leaves a file untouched for a while. A truncation is not a step of the flush protocol and needs no `fsync` of its own: a power cut that loses it leaves the file longer than necessary, and nothing worse.

The file always ends at a page boundary. It never shrinks into the last page's padding, and a journal page is added to the file whole, before the first transaction is appended to it. Were the file to end inside a page, the file system would rewrite the block holding that page's end — when a truncation zeroes the rest of the block, and on some file systems again when the file grows past it — and a power cut during that write can corrupt the whole block, even though the page's own bytes do not change. A page boundary is safe because it is also a block boundary, as [the reuse rule](#) assumes.

4.2.3 Deallocating reusable pages

A reusable page's blocks may be given back to the file system without shrinking the file.

Punching a hole — `fallocate` with `FALLOC_FL_PUNCH_HOLE` on Linux, `fcntl` with `F_PUNCHHOLE` on macOS, `FSCTL_SET_ZERO_DATA` on a sparse file on Windows — releases the page's blocks, and the page reads as zeros until it is written again. Zeros serve a reusable page as well as any stale bytes do, since nothing reads it. Like a truncation, it needs no `fsync` of its own: a power cut that loses it leaves the blocks allocated, and nothing worse. And because every page consists of whole blocks, it never touches a block that holds any part of a live or fallback page. It returns the space of reusable pages anywhere in the file, where truncation returns only those at its end; the file keeps its length, which tools that copy it without preserving holes will notice.

A deallocated page needs its blocks back before it is written again. A write into a hole can find the disk full only at write-back, which is the [ungraceful way](#) to learn it, since a failed `fsync` ends the session. So an implementation allocates a deallocated page's blocks before writing to it, as it would when extending the file.

4.3 The flush protocol

Journal segment E collects the transactions issued between flush $E - 1$ and flush E . Appends to it are protected by the chained, epoch-salted transaction CRCs and are **not** `fsync`d individually.

Flush E then proceeds:

1. **Fold** journal E in memory into content descriptions for each allocation's dirty ranges.
2. **Choose target pages:** reusable pages, else grow the file.
3. **Write data pages** for the dirty ranges. Remaining space in each new page may be filled with live content relocated from sparse pages — consolidation that costs no extra page writes.
4. **Write further consolidation pages**, within whatever budget the implementation chooses.
5. **Write address-table pages:** the statements describing everything touched in steps 3–4, plus any [tombstones](#).
6. **Choose a reusable start page for journal segment $E + 1$.** That page needs no write, except in a session's first flush: journal segment $E + 1$'s CRC chain is salted with epoch $E + 1$, so stale bytes in the page cannot validate as journal content unless they belong to a segment whose header a crash lost, and only a session's first flush can re-issue that header's epoch ([details](#)).
7. **`fsync`.**
8. **Write the header** into slot $E \bmod 2$: epoch E , the root address-table payload, journal segment $E + 1$'s start page, CRC.

No further `fsync` is required, and `flush E + 1` may begin immediately: step 7 already made state `E` recoverable, so nothing waits for the header write of step 8 to become durable. See [below](#).

4.3.1 The two invariants

The protocol maintains exactly two, and they are what the argument rests on.

I1 — a header is issued only after an `fsync` that covered (i) everything the new header references, except the journal segment it names, and (ii) the previous header. The header write of epoch `E` happens after `fsync E` returns, and `fsync E` also flushed the header of epoch `E - 1`, which was issued before it.

The named new journal segment is exempted from invariant I1 because it is the one part of a world that is still being written after the header is issued: header `E` names journal segment `E + 1`, which is still empty when `fsync E` runs. It needs no protection from I1: it validates itself, transaction by transaction, through the chained epoch-salted CRCs.

Consequence: a header found on disk implies that everything in its world except the journal segment it names is durable. There is no such thing as a valid header pointing at a missing page (except for the journal page it names, which does not need to exist until its first transaction is recorded), so recovery never has to validate a world before trusting it — only the journal it replays.

I2 — a flush writes only to reusable pages, with one exception: the commit in step 8 overwrites the slot holding the older of the two headers (the reuse rule).

Consequence: no matter where a power cut lands, the newer of the two headers and its world are physically intact, and so is the older world for as long as recovery could still choose it.

4.3.2 Why one `fsync` suffices

Because I1 and I2 together mean the only thing a crash can destroy is the currently active journal or work that nothing yet points at. A crashed flush is simply re-run against an untouched previous state; there is no partial application to undo and no write-ahead copy to consult.

Note what makes `flush()`'s return value honest. When `flush E` returns, `fsync E` has completed, so header `E - 1` and journal segment `E` are durable, so **state `E` is recoverable even though header `E` itself may not be durable yet** — recovery would land on header `E - 1` and replay journal segment `E` in full. The header write is a *representation* change, not the durability point; durability is established by the `fsync`.

This is why `kladde` needs one `fsync` per flush where a random-access database like `LMDB` needs two: `kladde` can afford to let the header lag one write behind and lean on journal replay. By contrast, `LMDB` has no journal and thus `fsyncs` twice: once after writing updates into reusable pages (like `kladde` does) and again after overwriting the alternating meta page (unless turned off by `MDB_NOMETASYNC`).

4.4 Recovery / loading a `kladde` file

1. Read both header pages; keep the CRC-valid ones; take the one with the **highest epoch**.
2. Bulk-load its world.
3. Replay the valid prefix of the journal segment it names, stopping at the first transaction whose chained CRC fails.

Thus, recovery happens entirely on the storage layer and is agnostic to the schema.

If both header slots hold a CRC-valid header, the one with the higher epoch governs and the other is ignored.

A load may retire the older world outright — all live pages stay live, every fallback page becomes reusable, and none is left in between — **but only once the governing header is durable**. That is not automatic, because the header of the last flush was [never fsynced](#): its slot can still revert to the epoch before it, and recovery would then land on a world the new session has already started overwriting. One `fsync` at open, after the headers are read and before any page is offered to the allocator, settles it; a failure of it is [fatal](#) like any other.

This is worth a syscall per open, because respecting the older world instead is not free: it means resolving *its* address table as well, purely to learn which pages it reaches. An implementation that would rather not fsync at open can have the same effect by letting its first flush extend the file rather than reuse anything, since that flush's own fsync makes the governing header durable and its commit retires the older world. The reference implementation uses the “fsync at open” option for simplicity.

4.4.1 Case analysis for a power cut during flush E

- **Successful case: header E reached the disk whole** — the cut came after step 8, and that write happened to be written back before the power was lost. By I1 everything in header E's world except journal segment $E + 1$ is durable, and header E carries the highest epoch among the CRC-valid headers, so recovery picks it and reaches state E. It then replays whatever CRC-valid prefix of journal segment $E + 1$ it finds, which here is empty: its start page was only chosen in step 6, the writer is blocked for the duration of the flush, and the stale bytes in that page cannot validate as journal content because the chain is salted with epoch $E + 1$. So the flushed state E is recovered exactly.
- **Error case: header E did not reach the disk** — either its write tore, leaving an invalid CRC, or it never arrived, leaving the slot's previous occupant (header $E - 2$, stale but CRC-valid). Recovery is indifferent between those sub-cases because it selects the valid header with the *highest* epoch, not merely a valid one. The governing header is then $E - 1$, which fsync E made durable and which I2 leaves untouched; by I2 its world is intact except for journal segment E, whose first page it names. Recovery lands on state $E - 1$ plus any valid prefix of the (possibly torn) journal segment E — at worst losing transactions that no completed fsync ever covered, which is within the guarantee.

If the cut came *before* fsync E returned, header $E - 1$ may itself be missing, and its slot holds header $E - 3$. Recovery then falls one epoch further back, to header $E - 2$ and journal segment $E - 1$, which fsync $E - 1$ made durable — the same argument one step down, and still within the guarantee.

A finer-grained walk-through, step by step through one epoch, is in [the implementation notes](#).

4.5 fsync failure is fatal

After a failed fsync, the operating system may mark dirty pages clean without having written them. The only sound reaction is to **discard all in-memory state and reopen from disk**; an implementation must not retry the fsync and must not continue the session.

This is the “fsyncgate” lesson, and it is a requirement rather than a recommendation: continuing after a failed fsync can silently publish a header whose world is not durable, which breaks I1 and therefore every guarantee above.

4.6 Headroom

Copy-on-write needs free pages to make progress. An implementation *may* reserve slack, or grow the file early, so that a nearly-full disk cannot deadlock the very consolidation that would free space — this is why ZFS reserves slop space. It is a quality-of-implementation choice rather than a durability requirement, for two reasons.

Running out of space costs nothing that was acknowledged. This is a *reported* error rather than a crash or a power cut, so the [stronger guarantee](#) is the one in force: the operating system is alive, the process is alive, and every transaction whose call returned successfully is in the journal and stays there. A journal append that cannot get a page fails its own transaction, which therefore never returned successfully. A flush that cannot get a page fails in steps 2–5, before the fsync of step 7 and therefore before any header is issued; it aborts against a previous state that I2 left untouched, and the journal still describes every acknowledged transaction, so a later flush — once something has been freed — commits exactly what the failed one would have. What a full disk takes away is **progress**, not durability, which is the whole reason headroom belongs here as a policy rather than as a requirement.

For the session to stay usable afterwards, a flush that fails partway must not leave its in-memory effects half-applied: an implementation either applies them only once the commit succeeds, unwinds them on failure, or poisons the session as it would after a failed fsync.

The commit itself never needs to extend the file. Both header pages always exist, and a flush small enough to keep its statements [in the header](#) writes no other page at all. So “free something and flush” can commit with zero bytes free on the device, and each such commit retires an older world and turns its pages reusable. What is left is a file with no reusable page *and* no room to grow, which is a file that is essentially all live — and there “cleaning cannot keep up” and “the file is honestly this large” [coincide](#).

The reference implementation therefore reserves no slack beyond what a consolidation immediately needs, preferring small files.

4.6.1 Detecting that the disk is full

An out-of-space failure must not be *deferred past* the `fsync` of step 7, and it cannot be: that `fsync` is the last point before a header is issued, and [a failed `fsync` ends the session](#). So the choice of how to extend the file is about how *gracefully* the failure arrives, not about whether it is caught.

The graceful form is to extend by **allocating the blocks up front** — `fallocate`, `posix_fallocate`, `F_PREALLOCATE` — which reports out-of-space immediately, as an ordinary error the flush can return while the session stays usable. The ungraceful form is to extend by **making the file longer without allocating** — `ftruncate`, or a seek past the end — which produces a sparse region that fails only at write-back, turning a recoverable “disk is full” into a fatal `fsync` failure and a forced reopen.

4.7 What this costs

Stated honestly, because the trade is real.

- **Files are larger during operation.** Superseded pages linger until consolidation rewrites them. The overhead is a policy outcome, not a constant: garbage accrues in proportion to write traffic and drains in proportion to consolidation effort, so enforcing “consolidate until live fraction $\geq \tau$ ” bounds the file at `live_size /  $\tau$` .
- **Total bytes written exceed an in-place design’s**, because cleaning rewrites survivors. The worst case is uniform random small writes across a large file — the workload that hurts every log-structured system.
- **A minimum flush is a handful of pages** (data + address table + header) even for a tiny transaction. The crossover favours this design as soon as writes scatter at all, but frequent single-byte flushes pay more here than an in-place design would.
- **Full trust in `fsync`**, leaned on harder than an in-place design does, mitigated by the epoch cross-checks turning a broken contract into a detected failure.

In exchange, committed bytes are never in the write path, so there is no class of corruption to detect and no recovery path that has to repair anything. The [superseded in-place flush](#) is what this replaced, and why.

Chapter 5

Address table

The address table maps allocation ids to their existence, size, and content. It is the only structure in a kladde file that is interpreted rather than merely pointed at, and it is what makes a flush write a small delta rather than rewriting the world.

5.1 The idea

The table is a list of **statements**, written into pages of kind `AddressTable`, where each statement implicitly inherits its page's [epoch](#). Statements from all live pages are merged, and contradictions are resolved by recency.

Why statements rather than entries. An entry-per-allocation table has to be rewritten wherever an allocation's description changes, which under copy-on-write means rewriting a whole page to change a few bytes. Statements are small, independent, and shadowing: a flush appends what changed, and nothing it does not mention is affected. The cost is that a reader must merge, which is paid once at open rather than per access. The [superseded whole-entry design](#) records what the alternative looked like and why it was abandoned.

5.2 Logical level

5.2.1 Statement types

Statement	Existence of id	Size of id	Content of id	Usage note
<code>Ref(id, offset, size, address)</code>	exists	$\geq \text{offset} + \text{size}$, smallest compatible with all live statements	Content in range <code>[offset, offset + size)</code> equals <code>file[address, address + size)</code> .	<code>[address, address + size)</code> must be within a single Data page. As a <i>writer invariant</i> (readers never depend on it), every byte in a Data page is referenced by at most one live Ref; this is not needed for correctness but keeps data-page coverage a plain counter instead of a refcount. See Design directions .

Statement	Existence of id	Size of id	Content of id	Usage note
Zero(id, offset, size)	exists	$\geq$ offset + size, smallest compatible with all live statements	Content in range [offset, offset + size) is zero.	Costs no data bytes, so it is how a range that currently holds data is set back to zero without writing zeros anywhere. Consolidation may rewrite a [data] [zeros] [data] ... stipple as one Ref or Inline over a range it fills with actual zeros — useful for data types with unbalanced variant payloads.
Shrink(id, n)	exists	$\geq$ n, and anchors the size: extents below this statement's epoch drop out	Anything at or beyond n is zero.	The size-reducing half of a resize. Its content claim is dormant while the allocation stays at n, and activates if the allocation later grows past n — which is exactly what stops a truncated tail from resurfacing.
Grow(id, n)	exists	$\geq$ n	Nothing. Matches no probe.	The size-increasing half, carrying no content claim. Growing never needs to deny anything: every statement that could cover a probe at or past the old size has an epoch below the anchor and is therefore already denied by it. Grow(id, 0) is how an existent zero-sized allocation is stated.

Statement	Existence of id	Size of id	Content of id	Usage note
Tombstone(id)	does not exist	0	All bytes are zero.	Like Shrink(id, 0) except that it also asserts non-existence. If a later epoch asserts existence again, the tombstone's effect on content remains: ranges the new incarnation leaves uncovered still resolve through it.
Inline(id, offset, size, payload)	exists	$\geq \text{offset} + \text{size}$, smallest compatible with all live statements	Content in range [offset, offset + size) equals payload.	Small-size optimization. Requires $1 \leq \text{size} \leq 251$, since the payload length is encoded as an offset past the last statement tag. Zero-sized payloads stay unrepresentable, and nothing needs them: Grow(id, 0) states an existent zero-sized allocation.

Why the resize is split in two. A single Size(id, n) statement would do two jobs — anchoring the size, and claiming that everything at or past n is zero — and those two are needed in opposite directions. A shrink needs both; a grow needs only the size, because the territory it exposes is already denied by the last shrink. A grow-emitted Size would therefore carry a *dormant* content claim that activates later, when the allocation grows past its bound, which is a durable source of statements that survive without doing anything.

Why unwritten content is zero rather than unspecified. An earlier design left it undefined — a reader “must not assume anything”, and an implementation could hand out whatever was physically there. Specifying zero costs nothing in storage, since a range that resolves to zero by default occupies no bytes either way, and it buys three things. It makes two conforming implementations return **identical bytes** for the same file, which the rest of the format already insists on for canonical encodings and fingerprints, and which lets a conformance corpus state an expected value for every byte rather than masking holes out. It stops deleted data from **reappearing**: a consolidator filling a hole with “arbitrary data” fills it in practice with whatever is adjacent in the buffer it is packing, which is other allocations' content. And it gives [schema evolution](#) its default for free — a field added to a struct reads as zeros from every file written before it existed.

What it gives up is narrow: a fold may no longer elide a shrink immediately followed by a growth, because the re-grown region must now read as zeros rather than as whatever survived.

5.2.2 No conflicts within each epoch

Statements in the same epoch must not conflict with each other:

- no two statements in the same epoch may make contradicting statements about the existence of an id;

- no two statements in the same epoch may name any byte of content twice, not even if the two statements assign the same value to that byte. This includes zeroing: an assignment of zero to a given byte — whether by a Zero, Shrink, or Tombstone statement — conflicts with any other assignment to the same byte, including a second zero assignment.
- At most one Shrink and at most one Grow statement per id is allowed per epoch, and never both: a flush knows whether it is growing or shrinking.
- Multiple Ref, Zero, and Inline statements for pairwise disjoint ranges within the same allocation are allowed. They don't conflict in regard to the size of the allocation because they all merely bound the size from below.
- However, a Ref, Zero, or Inline statement conflicts with a Shrink(*n*) statement in the same epoch if its upper bound $\text{offset} + \text{size}$ is larger than *n*. A Grow never conflicts with anything, since it claims no content and only bounds the size from below.

5.2.3 Conflict resolution across epochs

Two statements with different epoch are allowed to contradict each other. The newer statement takes precedence, but only as far as it contradicts the older one; any part of the older statement that doesn't contradict it survives. This is resolved at the granularity of the allocation's existence, its size, and the value of each of its content bytes.

Existence. Allocation id exists if the statement with highest epoch that mentions id is Ref, Zero, Shrink, Grow, or Inline. It does not exist if that statement is Tombstone(id), or if no statement mentions id.

Size. Let *anchor_epoch* be the largest epoch of any Shrink(id, *n*) or Tombstone(id) statement (or *anchor_epoch* = -1 if the id has neither), and call the statement at that epoch the *anchor*, whose *n* is 0 when it is a Tombstone. The size of allocation id is the maximum over:

- the anchor's *n*, if the id has an anchor;
- the *n* field of every Grow(id, *n*) statement with epoch > *anchor_epoch*; and
- all values $\text{offset} + \text{size}$ of all Ref(id, offset, size, ...), Zero(id, offset, size) and Inline(id, offset, size, ...) statements with epoch > *anchor_epoch*.

The maximum is over a non-empty set whenever the allocation exists, since existence requires at least one statement mentioning id and forbids the newest of them from being a Tombstone. A **non-existent** allocation falls out of the same rule with size 0 and needs no separate case: its newest statement, if any, is a Tombstone, which is therefore its anchor with nothing above it; and if no statement mentions the id at all, the maximum is over the empty set, which is 0 by convention. (Existent zero-sized allocations are still distinguished from non-existent ones — by the **Existence** rule above, not by this one.)

Only Shrink and Tombstone anchor the epoch; a Grow contributes a lower bound and nothing else, which is why dropping one can never readmit older extents. A Tombstone anchors at *n* = 0 and denies every probe, so a re-allocated id is described entirely by the statements above its tombstone.

Note that, under the no-conflicts rule, the predicates epoch > *anchor_epoch* above could be relaxed to epoch ≥ *anchor_epoch* without changing semantics. The stricter formulation is chosen to be more robust to a buggy writer that violates that rule: the strict form then yields a bounded allocation with an illegal statement out of range, where the relaxed form would yield a probe with two same-epoch matchers.

Content at a given offset probe. Defined by the statement with highest epoch that matches any of these patterns:

- Tombstone(id);
- Shrink(id, *n*) where *n* ≤ probe; or
- Ref(id, offset, size, ...), Zero(id, offset, size), or Inline(id, offset, size, ...) where $\text{offset} \leq \text{probe} < \text{offset} + \text{size}$.

A Grow statement never matches, which is the whole of the difference between the two halves of a resize. There is at most one winner, because at most one matching statement per epoch is allowed. Given the winning statement:

- no matching statement, or a winner of type Tombstone, Shrink, or Zero: the content is zero;
- Inline(id, offset, size, payload): the content is payload[probe - offset];

- `Ref(id, offset, size, address)`: the content is `file[address + (probe - offset)]`.

5.3 Physical format

Address table pages follow the [general page framing](#) and encode their payload in the content field. There is only a single kind for address table pages: `kind == AddressTable`. The header page is always an `AddressTable` page. Every `AddressTable` page contains zero or more references to other `AddressTable` pages followed by zero or more statements. References between `AddressTable` pages form a tree rooted at the header page: every page reachable from the header other than the header itself is named by exactly one `child_ref` among the pages reachable from the header, and no `child_ref` names a header slot, page 0 or page 1.

```
; `content` field of an address table page:
address_table_page := child_ref* 0:byte statements:statement*
child_ref          := page_number_delta:varint ; see "Delta encoding" below
statement          := id_delta (tagged_ref | tagged_zero | tagged_shrink
                               | tagged_tombstone | tagged_grow | inline)
id_delta           := varint ; see "Delta encoding" below
tagged_ref         := 0:byte offset_delta:varint size:varint address:varint
tagged_zero        := 1:byte offset_delta:varint size:varint
tagged_shrink      := 2:byte offset_delta:varint
tagged_grow        := 3:byte offset_delta:varint
tagged_tombstone   := 4:byte
offset_delta       := varint ; see "Delta encoding" below

; Inline is optimized for lots of small payloads. Max size is 255 - 4 = 251.
; size >= 1: zero-sized Inline payloads are deliberately unrepresentable.
; size_tag = size + 4 >= 5, so it can't clash with any of the above tagged_*.
inline            := size_tag:byte offset_delta:varint payload:byte{size_tag - 4}
```

5.3.1 Delta encoding

Child refs within a page are sorted by their page number, and statements within a page are sorted lexicographically by `(id, offset)`. This order is well defined even for `Tombstone` statements, which carry no offset, because no other statement mentioning the same `id` can coexist with a `Tombstone(id)` in one epoch.

To save encoding space by exploiting the compactness of varints, page numbers, ids, and offsets are delta-encoded. When starting to decode an address table page, a reader initializes a `page_cursor`, an `id_cursor`, and an `offset_cursor` to 0, then decodes the page in reading order.

- For each `child_ref`, increment `page_cursor` by `page_number_delta`; the resulting `page_cursor` is the number of the referenced page. Decode `child_refs` until encountering the delimiter byte 0 (which is also a varint-encoded zero). The delimiter cannot be confused with a `child_ref`, whose `page_number_delta` is never zero: the first child's page number is at least 2, since no `child_ref` names a header slot, and each later one exceeds its predecessor, since the tree names each page at most once.
- Decode statements until the end of the page's content field is reached, which can be detected by the `content_size` field of the [page framing](#). For each statement, perform these steps in this order:
 1. Increment `id_cursor` by `id_delta`.
 2. Read off the statement's `id` from `id_cursor`.
 3. If `id_delta != 0`: set `offset_cursor = 0`.
 4. If the statement isn't a tombstone: increment `offset_cursor` by `offset_delta`.
 5. If the statement isn't a tombstone: read off offset (or `n` for Shrink or Grow) from `offset_cursor`.
 6. If the statement is a Ref, Zero, or Inline: increment `offset_cursor` by `size` (where `size = size_tag - 4` for Inline).

Since `offset_delta` is a varint and therefore non-negative, `offset_cursor` must never need to move backwards. This is a **writer invariant** the format depends on for decodability. For Ref, Zero and Inline

it follows from the no-conflicts rule: their ranges within one epoch are disjoint, so sorting by offset already leaves the cursor at or below the next statement's offset. For `Shrink(id, n)` it follows too, since a content statement in the same epoch may not reach past `n`. For `Grow(id, n)` it does **not** follow, because a `Grow` conflicts with nothing: a writer must simply never put a `Grow(id, n)` in a page whose content for `id` reaches past `n`. That costs nothing, since such a `Grow` would be dead on arrival anyway — content reaching past `n` means the size already exceeds `n`.

Delta encoding buys more than compactness. It makes non-compliant order of statements *unrepresentable*. The decoder of the reference implementation exploits the fact that statements within each page are sorted lexicographically by `(id, offset)`, which allows it to efficiently merge statements across pages and iterate over them by increasing offset without having to copy and sort them first. Since a wrong order is *unrepresentable* rather than just forbidden by the spec, the decoder does not have to verify correct order.

5.3.2 Notes on the physical format

- The root `AddressTable` page *is* the current [header slot](#), so before the payload described by this grammar it carries the file-header fields, all covered by the header CRC.
- A `child_ref` deliberately stores only a delta-encoded page number and no epoch. The child's own framing carries its epoch, and [invariant I1](#) guarantees that a page referenced by a valid header is present and intact, so a per-child expected-epoch field would only duplicate an assertion the child already makes.
- Nothing in the format constrains *which* page a statement lives in or how the tree is shaped; those are writer policies.
- The `child_refs` must form a **tree**, and a reader should verify this, to avoid getting trapped in an infinite loop on cyclic refs or parsing the same page twice — which can happen even in a DAG that is not a tree.

5.4 Bounds

An implementation must support these, and must fail cleanly rather than silently wrap when an application exceeds them.

- **Allocation IDs** are 32 bit. This is visible to data type implementations, which may serialize a 32-bit pointer into one allocation to point at another. Varint encoding of ids therefore takes up to $\lceil 32/7 \rceil = 5$ bytes. Once Segments land, each will likely have its own id space, so a file may then contain more than 2^{32} allocations.
- **Page sizes** are uniform throughout a file and are given by the header's `log2_page_size`, currently 4 KiB with 8, 16, 32, and 64 KiB reserved. Therefore, offsets into a page that don't point at the exact end of the page always fit into 16 bit.
- **Payload sizes** of `Ref` is bounded by `MAX_PAGE_CONTENT`, the page size minus the 15 bytes of [page framing](#), thus at most $2^{12} - 15$ bytes in the current 4 KiB page size setup, and at most $2^{16} - 15$ if 64 KiB pages become a reality. Their varint encoding thus takes at most $\lceil 16/7 \rceil = 3$ bytes.
- **Page numbers** are 32 bit. Therefore **addresses** fit into $32 + 16 = 48$ bit (49 bit to point at EOF), their varint encoding takes at most $\lceil 49/7 \rceil = 7$ bytes, and the maximum **file size** is $2^{(32+12)} \approx 17.6$ TB with 4 KiB pages and $2^{(32+14)} \approx 70.4$ TB with 16 KiB pages.
- **Allocation sizes** are bounded by $2^{32} - 1$ bytes ≈ 4.3 GB. Note the `-1`, i.e., allocations of size 4 GiB are (just) *not* supported. A varint-encoded offset into an allocation therefore uses up to $\lceil 32/7 \rceil = 5$ bytes.
- The **number of statements** in a file must not exceed $2^{32} - 1$, and an implementation may limit it further, so that statements can be tracked with 32-bit indices that leave one value free as a niche for optional references, and counted — per id, say — in 32-bit counters. Even at an optimistic 3 bytes per statement that still allows 13 GB of address-table pages alone.
- The **framing** of a statement — its encoded length excluding an `Inline` payload — is at most **21 bytes**: a `Ref` with 5 bytes of `id_delta`, 1 tag byte, 5 bytes of `offset_delta`, 3 bytes of `size`, and 7 bytes of `address`. Most statements are far shorter.
- The **framings of the live statements naming any one id** must sum to at most $2^{32} - 1$ bytes, so that a reader can track an allocation's description cost in a 32-bit counter. That allows over four gigabytes of address table describing a single allocation — at the 21-byte maximum, more than 200

million live statements for one id — so no writer approaches it without already being pathological. An allocation's *fragments* need no bound of their own: they are non-empty and disjoint within $[0, \text{size})$, so there are never more of them than the allocation has bytes.

- The size fields of the **live Ref statements pointing into any one Data page** must sum to at most $2^{32} - 1$, so that a reader can track a page's live bytes in a 32-bit counter. The [writer invariant](#) holds that sum below one page's worth. This bound binds only files that let several Ref statements claim the same byte, which is legal, which a reader must therefore survive, and which without a bound would let the sum reach 2^{47} .
- The size of an encoded [transaction](#) (without framing) must not exceed $2^{32} - 1$ bytes, so that its length prefix fits in 32 bits. Trying to encode a larger transaction must fail rather than silently cutting the transaction in parts.

5.5 Design directions

The byte-referenced-once invariant. Not required for read correctness; keeping it makes data-page coverage a plain counter and consolidation locally decidable. Relaxing it would enable copy-on-write clones of allocations and deduplication, at the price of refcounted coverage and clone-aware consolidation. Since it is a writer invariant invisible to readers, choosing the strict form now costs nothing later.

Decision: copy-on-write clones are not pursued. They add complexity and open a zip-bomb-like attack vector, and most of the use case is covered by a [Move operation](#) instead, which transfers rather than shares and so keeps the invariant intact. If the need comes up later, it may be worth implementing at the data-type level rather than the allocation level — allocations reference disjoint regions of the file, and a data type on top resolves multiple indices to the same allocation, as one would with ordinary in-memory allocations.

Per-page epochs versus per-statement epochs. Per page is cheaper and imposes exactly one discipline — rewrites emit resolved truth — which consolidation requires anyway. Per-statement epochs would spend bytes on every statement to relax a constraint the design already satisfies.

Sub-allocation statements versus whole-allocation entries. This design keys statements by *byte offset*, which is stable under overwrites and resizes — the operations that dominate. The price is middle insertion: a splice that shifts content shifts the offsets of everything behind it, so all later statements of that allocation must be restated, $O(\text{fragments behind the splice point})$ in address-table bytes. If splice-heavy containers ever matter, the open path is a per-allocation indirection (a rope of ranges) layered *on top of* offset-keyed statements for exactly the allocations that need it; the format does not block it.

The Inline threshold. The encoding fixes the ceiling at 251 bytes; the policy threshold belongs far below it, and is an implementation choice rather than a format one. See [the implementation notes](#).

Chapter 6

Allocations

What an allocation is, how it is named, how one refers to another, and what its bytes mean.

6.1 Allocations

An **allocation** is a byte range that application data lives in. It has:

- a **stable id**, assigned once, never changed, and never zero — which leaves zero available to encode a null pointer;
- a **size** in bytes;
- **content**, resolved from the [address table](#).

Nothing else. An allocation is an untyped byte range; what its bytes mean is a question for the [schema](#) layer, and the storage layer never asks it.

An allocation is **not necessarily contiguous on file**. Its content is described by whatever `Ref`, `Inline` and `Zero` statements currently win its offsets, and those may point into any number of data pages, in any order, with gaps that resolve to zero and occupy no data pages at all.

Why not contiguous. Contiguity would have to be maintained against every write, which under copy-on-write means relocating an allocation whenever a byte in its middle changes. Giving it up costs a lookup per read — served in memory, and paid only while loading or when a flush relocates bytes, since application reads go to the loaded values rather than to the file — and buys a flush that writes only what changed.

A **zero-sized allocation** is the natural endpoint of the rules rather than a special case: no content statements at all, and an on-file existence consisting of a single statement such as `Grow(id, 0)`. Nothing may assume “at least one content statement”: even a large allocation can have none, if it is entirely left to its default content of all-zero bytes.

There is no distinction between resizable and fixed-size allocations. *Why not:* it existed in a [superseded design](#) so that neighbours of a fixed-size allocation could rely on it not moving, and in this design nothing is adjacent to anything — every reshape is a statement edit.

6.2 Stable ids, and why pointers are not addresses

A pointer stored inside an allocation is serialized as the **stable id** of the allocation it points at — never as an address.

This is the most consequential decision in the format, and it is what makes relocation cheap. When an allocation’s bytes move, the bytes of every pointer that targets it are unchanged, because those bytes name an id, not a location. The mover writes new address-table statements and copies the bytes; it never has to find, interpret, or rewrite a pointer. It does not even need to know which of an allocation’s bytes *are* pointers, which means it can relocate allocations whose contents it cannot parse.

The cost is one level of indirection on every dereference, and the need to reconstruct the id-to-content mapping when a file is opened. That reconstruction is the bulk read: the address table is resolved once at open into an in-memory index, and epochs are never consulted again at run time.

An implementation is required to preserve ids across a flush and across consolidation. It is *not* required to assign them in any particular way, or to reuse them in any particular order — though [the reference implementation's order](#) is chosen for good reasons.

6.2.1 Id recycling

An id becomes reusable as soon as a Tombstone for it is committed. It does **not** have to wait for the tombstone to die or for the old incarnation's statements to be physically removed.

This is safe because the tombstone matches every probe and outranks every statement below it: the new incarnation's statements are written above it, so the old incarnation can never win a probe, no matter how much of it survives physically.

One case needs no tombstone at all: freeing and re-allocating an id **within a single flush**. A tombstone and the new incarnation's statements would make contradicting existence claims in one epoch, which the [no-conflicts rule](#) forbids. The flush must instead emit statements that fully cover the new extent, which denies the old incarnation on its own — for example a bare `Zero(id, 0, n)` plus a `Shrink(id, n)` if the new allocation is smaller than the old one.

6.2.2 Pointer encoding

How a pointer is represented **inside an allocation's bytes** is a separate question from the [address table](#), and is **TBD**:

- the width — the table fixes ids at 32 bit, but whether the in-allocation representation is always four bytes, or a per-file property, is not decided;
- the **null representation**, and whether it is a reserved id value or a separate tag;
- alignment requirements, if any.

The reference implementation uses a 32-bit value with zero reserved for null, which gives an optional pointer the null niche for free in memory as well as on file. That is [an implementation choice](#) until this section says otherwise.

6.3 Ownership

In the current spec, every allocation has exactly **one owning pointer**. This is an invariant of the format, not a convention: the value graph reachable from the root is a tree of ownership, and a tool may rely on that when walking it.

Reclamation follows ownership. Dropping or overwriting an owning value releases the allocation it owns, recursively, in the same transaction as the write that superseded the pointer and after it. Both halves matter. *After*, because a [journal prefix](#) that freed an allocation while something still pointed at it would be dangling; *in the same transaction*, because replay truncates at transaction boundaries, so no recovered state is ever left holding an allocation that is unreachable and unfreed.

A future design that allows aliasing for specific pointers is **TBD**. Ideas include defining an `Aliasing-Pointer` type in the schema that is completely separate from the normal owning pointer type, a `Reference` type that allows aliases to owning pointers but must not outlive them, or a combination: an `OwnedSharedPointer` that owns the allocation but may have additional `References` while it is alive. Its design constraints are that aliasing should be somehow restricted to a subtree of the data structure, and that generic tooling should be able to assume that common `Opaque` types that the tool knows about use pointers that no *other* type aliases.

6.4 Content semantics

Every byte of an allocation that has not been explicitly written **reads as zero**.

Concretely: after an allocation is created, its bytes are zero. After a resize, the first `min(old_size, new_size)` bytes are preserved and everything beyond reads as zero. Two conforming implementations therefore return the same bytes for the same allocation of the same file.

Zeroing uninitialized bytes of allocations costs nothing to store — a range that resolves to zero by default occupies no data bytes — so this is a constraint on what a *reader* returns rather than on what a writer must write. What it does constrain is the flush: a shrink immediately followed by a growth may not be elided outright, because the re-grown region must read as zero rather than as whatever survived. In practice the shrink is [emitted regardless](#) and the re-grown range then resolves through it, so the statements on disk are unchanged.

A consequence worth naming for data-type authors: a type whose zero bit-pattern is its natural default — an empty string, a null pointer, a false flag, a zero integer — needs no initialisation write at all. A vector of a thousand empty strings is one `Grow`.

6.5 Free space

Free space is not represented on disk, and neither is liveness.

A page is live if it is reachable from a committed header; content within a page is live if some current statement still refers to it. Both are derived, and neither is recorded. See [page states](#).

An implementation's in-memory accounting of how much of each page is still relied upon — what drives its consolidation policy — is entirely its own business, and nothing about it is persisted. See [Liveness accounting](#).

6.6 Reclamation is not part of this specification

Whether an implementation reclaims garbage, when it does so, and which pages it chooses are unconstrained.

This is safe for the same reason relocation is: a consolidation step rewrites content into fresh pages and emits address-table statements describing where it went, producing a file indistinguishable from one that happened to be laid out that way from the start.

Two implementations with wildly different policies therefore produce files that are equally valid and mutually readable. An implementation that never consolidates is conforming; it will simply accumulate garbage until it runs out of space, which is a quality-of-implementation failure rather than a correctness one.

The one thing that *is* required is that reclamation be **externally invisible**: it must not change any allocation's id, size, or content.

Chapter 7

Journal

What a mutation records, how records are grouped into transactions, how transactions are laid out in pages, and how a torn tail is recovered.

Status: draft. The record set, its byte encoding, and the page layout are specified; none of it is frozen.

7.1 What the journal is

An append-only sequence of **transactions**, each a sequence of **records** describing byte-level effects on allocations. Every transaction is appended before the call that completes it returns, so an application crash loses no transaction that has been reported as complete. Appends are not fsynced; what a power cut can take is stated in [Durability](#).

The journal is kept in **segments**, one per epoch. Segment E collects the transactions issued between flush $E - 1$ and flush E , in a chain of pages whose first page the header of epoch $E - 1$ names. Flush E folds segment E into address-table and data pages, and the header it commits names the first page of segment $E + 1$.

The journal is deliberately **type-agnostic**. It records effects on allocations, not application-level operations. There is no “push element”, no “insert key” — only `resize`, `free`, `write`, and three records that move content around.

Why. Replay never dispatches on an application type, never needs a type registry, and never calls back into a container implementation. That is exactly what makes a file readable by an implementation that has never heard of the container that wrote it. What it gives up is the ability to collapse operations that cancel only at the semantic level: a rope that applies a hundred edits which cancel out still records a hundred edits’ worth of byte changes, because nothing in the log knows they were edits.

7.2 Record kinds

record	effect
<code>Free(id)</code>	ends <code>id</code> ’s existence
<code>Resize(id, size)</code>	sets <code>id</code> ’s size, preserving <code>min(old, size)</code> bytes
<code>Write(id, offset, bytes)</code>	overwrites <code>bytes.len()</code> bytes at <code>offset</code> within <code>id</code>
<code>Splice(id, offset, old_len, bytes)</code>	replaces <code>old_len</code> bytes at <code>offset</code> with <code>bytes</code> , shifting the tail and resizing
<code>Copy(src, src_offset, len, dst, dst_offset)</code>	copies a byte range between, or within, allocations
<code>Move(src, src_offset, len, dst, dst_offset)</code>	copies like <code>Copy</code> , then leaves the vacated source bytes zero

Records take effect in the order they were recorded. Every record is defined on every state, so no record needs an earlier one to be meaningful, and replay never fails on a record that decodes.

7.2.1 Every record but Free brings its allocation into existence

A record that writes to an allocation asserts that the allocation exists and is large enough to hold what the record writes. If the allocation does not exist, the record creates it with size zero before taking effect; if the written range reaches past the allocation's end, the record first grows the allocation to that end. Bytes that growth exposes read as zero, like all [unwritten content](#).

The written range is `[offset, offset + bytes.len())` for `Write`, `[offset, offset + old_len)` for `Splice`, and `[dst_offset, dst_offset + len)` for `Copy` and `Move`. `Resize` sets the size exactly, and creates the allocation just the same. The rules hold for zero lengths too: `Write(id, n, [])` asserts that `id` exists with at least `n` bytes, and changes no content.

So there is no `Alloc` record. `Resize(id, n)` on an `id` that does not exist allocates `n` zero bytes, and `Resize(id, 0)` states an existing zero-sized allocation — just as the [address table](#) has no statement for allocating, and every statement but `Tombstone` asserts existence.

`Free(id)` ends `id`'s existence; afterwards `id` reads as size zero with no content, and on an `id` that does not exist `Free` has no effect. A later record that brings `id` into existence again begins a new incarnation, whose bytes read as zero wherever that record and its successors do not write.

7.2.2 Records that read

`Splice`, `Copy`, and `Move` read existing content; the others only write. That distinction matters to an implementation's flush, and is not visible in the file.

A record reads its whole source range before it writes anything, so a `Copy` or `Move` between overlapping ranges of one allocation behaves like `memmove`. Source bytes at or past the end of their allocation, or in an allocation that does not exist, read as zero, and reading them changes nothing about the source.

7.2.3 Move

Move is a Copy that then zeroes the source range, except where the destination has just overwritten it. The source keeps its size and its existence; only its content changes. With `src == dst`, `Move(a, 0, 8, a, 4)` leaves `[0, 4)` zero and `[4, 12)` holding what `[0, 8)` held.

Why a separate record. After a `Copy`, source and destination both hold the bytes, so a flush may not describe the destination by `Refs` to where the bytes already lie: those data-page bytes would then be referenced twice, which the [byte-referenced-once writer invariant](#) rules out, and the flush has to write the bytes again. After a `Move`, the source no longer holds them, so the flush can hand the source's `Refs` to the destination and state the vacated range as `Zero`, and [data-page coverage](#) stays a plain counter. Rope- and tree-like containers shift ranges between neighbouring nodes constantly, and `Move` makes each such shift a statement edit rather than a byte copy; [the draft](#) sketches how a flush states one.

7.3 Transactions

Transactions are the unit of atomicity. If the application crashes while recording a transaction, or while the journal is being appended to, the next open replays that transaction either completely or not at all.

This is the one durability guarantee application authors build on directly, which is why transactions are specified here while the mechanisms an implementation uses to *group* operations into them — batching, nesting, buffering — are not. Those are [implementation concerns](#).

How operations are grouped is free, but every record belongs to exactly one transaction. A record that occurs outside any transaction the application opened is recorded as a transaction of its own, so a lone operation can never be torn.

Atomicity hides intermediate states; it does not erase order. Records take effect in sequence within a transaction just as across transactions, so a `Copy` recorded after a `Write` copies what was written.

7.4 Ordering

The journal’s most important invariant is not about any single record.

Every transaction must take a valid state to a valid state.

Replay stops only at transaction boundaries, so the states inside a transaction are never observed and may be invalid, while the state at every boundary is one that recovery can land on. “Valid” permits stale, and permits slightly leaky. It does not permit dangling: nothing may become reachable before everything it depends on is already earlier in the sequence, and nothing may be freed before whatever superseded it is already published.

In practice this means every mutation decomposes as **prepare** new, unreachable state → one **publishing** write → **clean up** what it replaced, with the steps in one transaction or spread over several. Growing a container, removing from the middle of one, and replacing a map entry all fit this shape. A vector push that must grow, for instance, resizes and writes the new element *before* bumping the stored length, because the length is what makes the element reachable. A mutation that cannot be ordered this way belongs in a single transaction, which is the format’s only atomic-group mechanism and the only one it needs.

This is a discipline every container implementation must follow, and **no type system enforces it**. It is the single largest source of latent correctness risk in the design, which is why [the conformance suite](#) makes it a mechanical check: truncate at every byte offset, replay, assert validity.

7.5 Encoding

Integers are unsigned LEB128 varints and fixed-width fields are little-endian, as everywhere in the file ([integer encodings](#)).

7.5.1 Records

```
record := free | resize | write | splice | copy | move
free   := 1:byte id:varint
resize := 2:byte id:varint size:varint
write  := 3:byte id:varint offset:varint bytes
splice := 4:byte id:varint offset:varint old_len:varint bytes
copy   := 5:byte src:varint src_offset:varint len:varint dst:varint dst_offset:varint
move   := 6:byte src:varint src_offset:varint len:varint dst:varint dst_offset:varint
bytes  := len:varint byte{len}
```

Tags 0 and 7 through 255 are reserved. Ids and offsets are absolute rather than delta-encoded as in the address table, since records, unlike statements, are not sorted.

Every record respects the [bounds](#): ids are non-zero and fit in 32 bits, and every sum of an offset and a length that a record implies is at most $2^{32} - 1$, so that no record, growth included, can make an allocation larger than the bound allows.

7.5.2 Framing

```
transaction := length:varint record* crc:byte{4}
```

length counts the bytes of the records, and crc is the value of the segment’s [CRC chain](#) just after them. A transaction is complete — committed — once its crc is in the journal; a transaction whose crc is missing or does not match never happened. length is at most $2^{32} - 1$ ([bounds](#)).

Why a length prefix. It tells a reader where the crc is before the reader decodes a single record, so nothing unvalidated is ever parsed. It also means a writer must know a transaction’s size before appending its first byte, and so encodes the transaction whole — which costs nothing, since an unfinished transaction [cannot be folded](#) and is held in memory until it ends anyway.

A transaction whose crc matches but whose records do not decode to exactly length bytes, or break a bound, is not a torn tail — a torn write cannot produce a matching CRC — but a writer’s bug. A reader

must refuse the file, as it would refuse any other damage, rather than stop replay there and silently drop every transaction after it.

7.5.3 Pages

A **segment is a singly linked chain of pages**, starting at the page the header's `journal_pointer` names. Journal pages carry neither the `page framing` nor a header of their own; each page reserves only its last 8 bytes, for a **trailer**:

```
journal_page := stream:byte{page_size - 8} next:byte{4} crc:byte{4}
```

The segment's transactions form one byte **stream**, laid out across the stream areas of its pages in chain order. Where a page's stream area ends, the stream continues at the start of the next page — in the middle of a transaction, a record, or a field if it must, which is what lets a record carry a payload larger than a page. A transaction's `crc` is no exception: it splits like any other field, and its two parts are checked together once both have been read.

The trailer is written when, and only when, the stream continues past its page: `next` is the number of the page the stream continues on, and `crc` is the value of the CRC chain just after `next`. So the writer chooses each next page only once it needs one, and the last page of a segment has no valid trailer, only stale bytes where a trailer would go. `next` names a reusable page, or a new page added at the end of the file, as [the reuse rule](#) allows, and never a page already in the segment.

A reader follows `next` only once the trailer's `crc` has matched, so it never trusts a pointer that no checksum covers. The trailer also means that every byte a page holds is covered by a checksum as soon as the stream has moved past the page, including the first part of a transaction that continues on the next one.

7.5.4 The CRC chain

Every checksum in a segment is a value of one running CRC-32C over the segment's chain input, using the CRC-32C [specified for the file](#). The chain input is the segment's epoch as 8 little-endian bytes — one more than the epoch of the header that names the segment — followed by every stream byte and every `next` field in order, but not the checksum fields themselves. Each checksum field stores the CRC-32C of the chain input up to that field: a transaction's `crc` covers everything up to its last record, and a trailer's `crc` everything up to its `next`.

- *Chained*, so that recovery keeps the longest valid **prefix** rather than the longest valid set: a hole cannot be skipped, even if the bytes past it are intact. This matters because operating-system write-back is not ordered, so a later journal page can reach disk while an earlier one does not.
- *Epoch-salted*, so that stale bytes left in a reused page do not validate as journal content. The epoch is not stored in the journal, because the header that names the segment implies it. This is what lets a flush hand journal segment $E + 1$ a reusable page without writing anything to it first — with [one exception](#).
- *Blind to its own checksums*, because CRC-32C, fed a message followed by that message's own CRC, always ends in the same state. Feeding each stored checksum back in would restart the chain from a constant after every transaction, and a stale transaction from any earlier segment that happened to start where a valid one ended would validate.

7.6 Recovery

1. Read and validate the [header](#), taking the CRC-valid slot with the highest epoch, and bulk-load its world.
2. Walk the segment from the header's `journal_pointer`, feeding the chain as it goes: check each transaction's `crc` once its records have been read, and each trailer's `crc` before following its `next`.
3. Stop at the first check that fails, or at a `journal_pointer` or `next` that points past the end of the file. The complete transactions before that point are the journal; whatever follows them is the torn tail.
4. Replay the complete transactions in order.

Replay is purely mechanical and involves no application code.

7.7 The start of a session

Two rules keep a session from damaging, or misreading, a journal that an earlier session left behind. Each costs at most one flush or one page write per session.

A session appends only to a recovered journal that is empty; a non-empty one it folds with a flush first. Appending would rewrite the page holding the recovered journal's end, and a power cut during that write can corrupt the transactions already on it. Those can be durable: when a crash loses header E , recovery lands on header $E - 1$ and replays journal segment E , which flush E 's `fsync` made durable — so damaging it would lose transactions that a completed flush had covered, which [Durability](#) promises cannot happen.

A session's first flush writes the first page of the journal segment it names, for which zeros suffice, unless that page lies past the end of the file. The epoch salt tells every segment apart except in one situation. When a crash loses header E after transactions were appended to the segment it named, recovery lands on header $E - 1$, and the recovering session's first flush issues epoch E again. If the page it named still held the lost segment's first transactions, they would validate — same epoch, same starting offset — although the session committing the header knows nothing of them; its later headers would describe states without them, and after a second lost header, recovery could replay stale transactions on top of a state they were never appended to. Only a session's first flush can re-issue an epoch, because writing header E required the `fsync` that made header $E - 1$ durable, so a lost header E never leaves recovery further back than $E - 1$. And only the first page of a segment is exposed, since every later page validates only as the continuation of the exact bytes before it.

7.8 Folding

Periodically the journal is **folded**: its records are applied, the on-file state is brought up to date, and a fresh journal segment is started. The protocol that commits this is in [Durability](#).

A flush folds its whole segment, which holds complete transactions only, so every header describes the state at a transaction boundary.

An implementation has wide latitude in *what* it writes. It may apply records naively in order, or it may optimize — cancelling the creation and freeing of an allocation that never escaped, dropping a write that a later write fully supersedes, eliding a grow-then-shrink that returns to the original size with nothing written in between. It may reorder freely, so long as the result is indistinguishable from an in-order replay. “Indistinguishable” is measured against [content semantics](#): every byte has one right answer, so a fold's freedom is to reach that answer by any route rather than to choose among answers.

7.8.1 Checkpoint versus commit

Decided: every checkpoint lands on a commit. A *commit* is the end of a transaction, marked by its `crc`; a *checkpoint* is a flush, which folds complete transactions only, so every header describes a state some commit produced. [Durability](#) requires exactly this, since a header describing half a transaction would let recovery land inside it — and it means the format needs no commit record beyond the transaction's `crc`.

A checkpoint that folded part of an unfinished transaction, to bound the memory the transaction holds, would need a way to undo that part should the transaction never end, and the format has none. So an unfinished transaction stays in memory until it ends, and keeping that small is the application's side of the bargain: [short transactions, and batches for long sequences](#).

Chapter 8

Schema

How a kladde file describes the types it contains.

This is the most fully specified part of kladde: it is pinned down precisely enough that any conforming implementation, in any language, produces **byte-identical** descriptor serializations and **bit-identical** fingerprints.

8.1 Why a file carries its schema

Two reasons, one defensive and one enabling.

Defensive. Without a schema, opening a file written by a different version of an application is undefined behaviour dressed as a successful read: offsets shift, a field is interpreted as the field next to it, and the corruption is silent. An embedded schema turns that into a detectable error.

Enabling. A schema that describes *layout*, not merely identity, carries enough information to actually bridge a difference rather than just report one. A reader whose struct gained a field can determine, from the writer’s descriptors, where the fields it does share actually live, and read them correctly. That is what makes evolution possible at all.

The cost model is what makes this affordable. The schema is stored **once per file**, not per value — unlike a tagged format such as JSON or Protobuf, where every instance pays. For a file holding millions of instances of a type, the schema blob is negligible; for a small settings file it may be a noticeable fraction, which is the honest trade.

8.2 The documents

document	contents
Type descriptors	the model: what kinds of type exist and what each carries
Canonical encoding	the byte serialization of a descriptor table
Fingerprints	the reproducible 128-bit hash identifying a type’s representation
Evolution	how a reader reconciles its own types against a writer’s — largely TBD

8.3 The guiding principle

A descriptor describes a representation, not an identity.

A descriptor says how bytes are laid out and interpreted. It does not say which source-language type produced them. Two types with byte-identical representations have identical descriptors and identical fingerprints, and this is intentional.

It is also harmless, because a value is always reconstructed **top-down from a known root type**. A reader never asks “what type is this anonymous blob?” — the parent field’s declared type already determines

which loader to call. The descriptor exists only to answer “does the writer’s layout for this field match what my loader expects, and if not, how do I bridge it?”

This principle resolves several questions that otherwise look hard. A type with a hand-written implementation that happens to write exactly the bytes of a plain struct *has* that struct’s descriptor — the presence of non-persisted fields in the author’s source type is invisible and irrelevant. A type is [Opaque](#) not because its implementation is hand-written but because its representation is genuinely not decomposable.

8.4 Scope

These documents define how descriptors are **modeled, serialized, and fingerprinted**.

Where the descriptor table physically sits in a file is a [container-layer](#) question. How a reader *resolves* a mismatch — the compatibility policy — is [Evolution](#), and is the least settled part of the specification.

Chapter 9

Type descriptors

The language-neutral model of how a value type lays out and interprets its bytes.

Status: settled and implemented. This document is normative.

9.1 Terminology

- **Type descriptor** (*descriptor*): a node describing one value type’s on-disk representation.
- **Descriptor table:** an ordered array of descriptors. A descriptor refers to another by its **index** in this table — a table-local, otherwise arbitrary integer. Indices are an artifact of one serialization; a type’s fingerprint does not depend on them.
- **Type graph:** the directed graph whose vertices are descriptors and whose edges are the references one descriptor makes to others. It may contain cycles — recursive and mutually recursive types are legal.

9.2 The kinds

A descriptor is one of four kinds — **Primitive**, **Struct**, **Enum**, or **Opaque** — plus **Array**, reserved for a future revision.

9.2.1 Primitive

A fixed-width scalar with a fixed byte encoding, identified by a **primitive code**.

code	type	width	encoding
0	u8	1	little-endian
1	u16	2	little-endian
2	u32	4	little-endian
3	u64	8	little-endian
4	i8	1	little-endian two’s-complement
5	i16	2	little-endian two’s-complement
6	i32	4	little-endian two’s-complement
7	i64	8	little-endian two’s-complement
8	f32	4	little-endian IEEE-754
9	f64	8	little-endian IEEE-754
10	bool	1	0x00 false, 0x01 true
11	char	4	little-endian Unicode scalar value

The primitive code doubles as the descriptor’s kind tag: codes 0..=127 are reserved for primitives, so a Primitive descriptor is a **single byte** with no further content. Its inline width is fixed by its code.

9.2.1.1 Reserved primitive codes

Fixed here so later additions cannot conflict. An implementation need not accept them until a corresponding type exists and can be tested.

code	type	width	encoding
12	u128	16	little-endian
13	i128	16	little-endian two's-complement
14	f16	2	little-endian IEEE-754 binary16
15	bf16	2	little-endian bfloat16
16–19	u24, u40, u48, u56	3, 5, 6, 7	little-endian, full byte width, no packing
20–23	i24, i40, i48, i56	3, 5, 6, 7	little-endian two's-complement

9.2.2 Struct

An ordered, fixed set of named fields laid out consecutively — no discriminant, no header.

Carries:

- name: UTF-8, **diagnostics only**; a struct's own name is not fingerprinted.
- fields: an ordered list of (field_name, type_reference).

Field **order** is significant: it is the on-disk order, and a field's position determines its byte offset. Each field **name** is significant. A positional (tuple) field's name is the decimal rendering of its position — "0", "1", ... — so the model stays uniform.

9.2.3 Enum

A discriminated union: a discriminant, followed by the fields of the selected variant.

Carries:

- name: UTF-8, diagnostics only.
- discriminant_width: bytes occupied by the discriminant — 1, 2, 4, or 8.
- variants: each (discriminant_value, variant_name, fields), canonically ordered by ascending discriminant_value.

Each discriminant value, each variant name, and each variant's field list is significant. Variant *declaration* order is **not**: a value's layout is selected by the stored discriminant, not by a variant's position, so reordering variants in the source without changing their discriminants changes neither the representation nor the fingerprint.

9.2.4 Opaque

A type whose internal structure the format does **not** decompose — a black box identified nominally.

Used for types with hand-written representations that are not a plain field sum: length-prefixed blobs, externally serialized payloads, built-in container types.

Carries:

- library_name: the package, module, or crate defining the type. Language-neutral: not necessarily a Rust crate.
- type_name: UTF-8.
- version: a (major, minor, patch) triple.
- inline_size: the fixed number of bytes the type occupies **inline**. Required because it is not structurally derivable — a type that stores its payload out of line occupies its fixed header here.
- parameters: an ordered list of type references, for generic parameters.

Only the **compatibility component** of version is fingerprinted, and it has two parts, both of which are necessary:

- a **stability flag** — true when major > 0, false when major == 0;

- the **leading nonzero component** — major if `major > 0`, otherwise minor.

Without the stability flag, `0.1.z` and `1.y.z` would both reduce to 1 and collide, even though crossing `0.x` → `1.x` is a breaking change. Patch-level changes, and minor-level changes above `0.x`, are thereby treated as representation-compatible.

This tuple is the same identity that semantic versioning already relies on, and it carries the same risk: it trusts the author to bump the major version whenever the representation changes. A structural descriptor needs no such trust, which is a standing argument for preferring structural descriptors wherever a type's layout admits one, and reserving Opaque for representations that genuinely are not decomposable.

9.2.5 Array (reserved)

A fixed-length homogeneous sequence: count consecutive values of one element type, no header, no discriminant.

Carries `element` (a reference) and `count`. Inline width is `count × element.inline_size`.

Reserved, not yet implemented. An Array is deliberately *not* interchangeable with a Struct of count identically-typed positional fields: the byte layouts coincide, but the kind tags differ and therefore so do the fingerprints. The kind exists precisely to avoid the descriptor blow-up of spelling out a large count as that many fields.

9.2.6 Pointer (reserved)

A kind tag is reserved for describing an owning pointer, so that a schema-driven tool could follow the ownership graph without help from any language binding.

Content and semantics are **TBD**. It is not needed until a language-independent garbage collector or leak-detection tool exists; the language bindings currently walk ownership using their own type knowledge. See [Tooling](#).

9.3 References and the root

A **reference** is an index into the descriptor table. Every type, `element`, and parameter field is a reference.

By convention the descriptor at **index 0 is the root** — the type actually stored in the file.

9.4 Inline size

Every descriptor determines a fixed **inline size**, the number of bytes a value of that type occupies within its parent:

- a Primitive's is fixed by its code;
- a Struct's is the sum of its fields';
- an Enum's is `discriminant_width` plus the largest variant's field sum;
- an Array's is `count × element`;
- an Opaque's is stored explicitly, because it is not derivable.

That the inline size is *always* derivable from the type graph is what makes evolution tractable: a reader can compute the **writer's** offsets from the **writer's** graph, even when its own sizes differ. See [Evolution](#).

Chapter 10

Canonical encoding

The byte serialization of a [descriptor table](#).

Status: settled and implemented. This document is normative: two conforming implementations must produce **byte-identical** output for the same type graph.

10.1 Encoding primitives

- `byte(b)` — a single octet.
- `varint(n)` — unsigned LEB128: seven bits per byte, little-endian groups, high bit is the continuation flag.
- `string(s)` — `varint(byte_length)` followed by the UTF-8 bytes of `s`.
- `reference(index)` — `varint(index)`.

10.2 Descriptor encoding

Each descriptor begins with a single **kind-tag byte** that both selects the kind and partitions the tag space.

0..=127 — Primitive. The tag byte *is* the [primitive code](#). The descriptor has no further content.

128..=255 — a non-primitive kind, followed by kind-specific content:

tag	kind	content
128	Struct	<code>string(name)</code> <code>varint(field_count)</code> , then per field <code>string(field_name)</code> <code>reference(type)</code>
129	Enum	<code>string(name)</code> <code>byte(discriminant_width)</code> <code>varint(variant_count)</code> , then per variant <code>varint(discriminant_value)</code> <code>string(variant_name)</code> <code>varint(field_count)</code> , then per field <code>string(field_name)</code> <code>reference(type)</code>
130	Opaque	<code>string(library_name)</code> <code>string(type_name)</code> <code>varint(major)</code> <code>varint(minor)</code> <code>varint(patch)</code> <code>varint(inline_size)</code> <code>varint(param_count)</code> , then per parameter <code>reference(type)</code>
131	Array (<i>reserved</i>)	<code>reference(element)</code> <code>varint(count)</code>
132	Pointer (<i>reserved</i>)	content undefined

Tags 133..=255 are unassigned and reserved for future kinds.

10.3 Canonical order

The order in which sub-elements are emitted is fixed, so that the encoding — and therefore the [fingerprint](#) — is fully determined.

Struct fields are emitted in **declaration order**, which is their on-disk layout order. A field's position determines its byte offset, so this order is part of the representation and is never re-sorted.

Enum variants are emitted in **ascending discriminant value**. A variant's position does not affect any value's layout, so declaration order is discarded in favour of this canonical order. Two enums differing only in the source order of their variants encode identically.

Opaque parameters and an **Array's element** keep their given order, being positional type arguments.

10.4 Table encoding

`varint(descriptor_count)`, followed by each descriptor in index order, root first.

This is the canonical on-file encoding of a schema. Its placement, framing, alignment, and checksumming within a kladde file are a [container-layer](#) matter.

10.5 Worked example

The type `struct Point { x: i32, y: u8 }` as a two-entry table:

```
02          table: 2 descriptors
80          [0] tag 128 = Struct
05 50 6f 69 6e 74  name: "Point"
02          2 fields
01 78 01      "x" -> reference 1
01 79 02      "y" -> reference 2
```

...except that reference 2 does not exist in a two-entry table. A real encoding of this type has three descriptors — `Point`, `i32`, `u8` — because primitives occupy table entries like anything else:

```
03          table: 3 descriptors
80          [0] Struct
05 50 6f 69 6e 74  "Point"
02          2 fields
01 78 01      "x" -> [1]
01 79 02      "y" -> [2]
06          [1] Primitive code 6 = i32
00          [2] Primitive code 0 = u8
```

Note that the table is free to place primitives anywhere and to duplicate or share them; the [fingerprint](#) is invariant under any such choice.

Chapter 11

Fingerprints

A fixed-size, reproducible hash identifying a type's representation.

Status: settled and implemented. This document is normative: two conforming implementations must produce **bit-identical** fingerprints for the same type graph.

11.1 The value

A fingerprint is a **128-bit (16-byte)** hash. Two fingerprints are equal iff all 128 bits are equal. It carries no flags and no reserved bits — every bit is hash output.

11.2 Hash function

SHA-256, truncated to its leading 128 bits — the first 16 bytes of the digest, byte 0 first.

SHA-256 is chosen for exact cross-language reproducibility. It is present in every mainstream language's standard library, has universal test vectors, and "compute SHA-256, keep the first 16 bytes" is trivial to reimplement identically. 128 bits gives a negligible accidental-collision probability for any realistic number of distinct schemas.

Cryptographic strength is deliberately **not** relied upon. A fingerprint is an integrity and identity check against accident, not against an adversary. An attacker who can write to a file can do far worse than collide a schema hash, so the format does not attempt to defend against one here.

A future format revision may substitute another hash or width; the choice is a format-version property.

11.3 What is and is not fingerprinted

The fingerprint of a type is computed from **the type graph reachable from that type**.

Included — everything that affects the representation or the identities used to reconcile it:

- the kind of each descriptor;
- primitive codes;
- discriminant widths and discriminant values;
- field names and variant names;
- field order and canonical variant order.

Excluded:

- the name of Struct and Enum descriptors. A type's own name does not affect its byte layout, so renaming a type must not change its fingerprint.
- table indices, entirely. References are encoded structurally, so the fingerprint is invariant under any renumbering or reordering of the descriptor table.

Reduced: an Opaque’s version contributes only its [compatibility component](#) — the stability flag plus the leading nonzero component. Patch releases, and minor releases above 0.x, therefore do not churn the fingerprints of every type that transitively contains them.

11.4 Cycles

The type graph may contain cycles, so a naive post-order fold does not terminate.

The traversal is a **white/grey/black depth-first search with memoization**. A node currently on the stack (grey) that is encountered again is a back edge, and is encoded as a **relative reference** — “the k -th enclosing type currently being hashed” — rather than by recursing. This makes the hash a pure function of graph structure, and it is the standard technique for hashing mutually recursive definitions.

Memoization on black nodes keeps the traversal linear in the number of distinct reachable descriptors.

The same cycle-awareness is needed for descriptor *equality* on recursive types, so this is shared machinery rather than fingerprint-specific work.

11.5 Determinism

The fingerprint must be a pure function of the type graph. Two implementations that agree on the graph must agree on the hash, with no dependence on:

- table layout, index assignment, or descriptor ordering;
- traversal order beyond what the canonical order fixes;
- hash-map iteration order, allocation addresses, or any other incidental runtime state.

An implementation that cannot demonstrate this on the conformance vectors is not conforming.

11.6 As a fast path

The [file header](#) stores the root type’s fingerprint.

This makes the overwhelmingly common case — an application opening a file it wrote itself, with an unchanged schema — a single 16-byte comparison. If it matches, the reader knows its own compile-time layout is exactly the writer’s, and can read at static offsets without parsing the descriptor table at all.

Only when it *differs* does the reader parse the writer’s descriptors and attempt [resolution](#). So the fast path costs one comparison and the slow path costs one graph reconciliation per open — never per value.

This is the standard arrangement: the fingerprint is a registry key, the schema is the source of truth. A fingerprint alone can detect a difference; it cannot bridge one, which is why it complements the descriptor table rather than replacing it.

Chapter 12

Evolution

How a reader reconciles its own types against the types a writer used.

Status: the least settled part of the specification. The direction is decided; most of the policy is TBD. This document records what has been settled and marks the rest explicitly.

12.1 The levels

It helps to separate four things that “compatibility” usually runs together.

level	meaning
(a) Detection	a reader can tell that the file’s schema differs from its own, and refuses rather than misreading
(b) Backward-compatible migration	a newer reader can read an older writer’s file
(c) Tolerant structural read	a reader can read a file whose schema differs structurally — fields added, removed, reordered — without bespoke per-version code
(d) Forward compatibility with preservation	an <i>older</i> reader can read a newer writer’s file and write it back without losing the fields it does not know about

Kladde targets (a) **through** (c) for the single-authoritative-application case, which is its bread and butter.

Level (d) is partially available for free — an older reader keeps working after a newer one writes, as long as the newer one added no persisted fields — but the harder flavour, where a *new* field survives a round trip through an old reader, is a **non-goal** for the first frozen format. It requires a preservation or overflow area and fights the fixed-offset model directly.

12.2 What is settled

Detection is mandatory. The [header](#) carries a magic number, a format version, and the root [fingerprint](#), and a reader that finds a mismatch it cannot resolve **must fail closed**. Silent misinterpretation is the one outcome the format does not permit. This is a correctness property, not a feature.

Identity is by name, not by position. Fields and variants are identified by their names, and enum variants by their explicit discriminant values. This is what makes reordering non-breaking and removes the declaration-order discriminant hazard, where inserting a variant silently renumbers everything after it.

Resolution happens at load, once per type. This is the observation that makes the whole thing affordable. A kladde reader reconstructs a fresh in-memory value when it opens a file and writes native layout on the next fold, so it interprets a foreign schema **exactly once** and never pays for evolution on the hot path.

Concretely: reconcile the writer’s type graph against the reader’s once at open, producing a small per-type **read plan** — for each of the reader’s fields, either “read from writer offset *X*” or “use a default.” Every value then follows the cached plan, with no string comparison and no graph walk.

A reader must be able to read at writer offsets. A loader cannot assume its own compile-time inline size. This is the mechanism that lets inline sizes evolve at all: the writer’s graph encodes the old sizes, the reader’s encodes the new ones, and resolution bridges them. It is a real constraint on the shape of a language binding’s load path, and it needs designing in from the start even if resolution itself lands later.

In-place mutation must run against the allocation’s active layout. This is the corollary, and it is the subtle one. If a reader loads a foreign layout but keeps pointing at the original allocation, then mutating it with native offsets corrupts it. An implementation must make “which layout is this allocation in” explicit:

- an **upgrade**-mode allocation is re-laid-out natively, at load or at first touch, after which static offsets are valid;
- a **retain**-mode allocation carries a runtime layout that the mutation path must use.

Static offsets are valid **only** for native allocations.

Cycle-aware canonicalization is shared machinery. Recursive types need it for fingerprinting and for descriptor equality alike; see [Fingerprints](#).

12.3 What is TBD

Layout policy: default and granularity. Is *upgrade* or *retain* the default? Is it a per-type attribute, a per-open flag, or decided per allocation by inspecting the diff? And is the schema tagged per allocation, per schema-boundary region, or globally for the file? Global is simplest and all-or-nothing; finer granularity allows lazy migration but needs a per-region schema reference.

Default behaviour on the first fold after a resolving open. Upgrade-and-rewrite is simple but breaks readback by the older application. Pinning the writer’s format preserves that but complicates the write path. One should be the default and the other should be available.

How much semantic-migration machinery to expose. Structural resolution handles add, remove, re-order, and rename-with-annotation. It cannot handle a change that is semantic rather than structural — splitting one field into two, changing units. Whether kladde offers registered from-fingerprint/to-fingerprint hooks, or leaves such changes entirely to the application, is undecided.

Rename robustness. Name-based identity is ergonomic but not rename-robust. An optional explicit ordinal annotation, used when present and falling back to the name otherwise, would cover both; whether to specify one is undecided.

Defaults for added fields. A reader whose type gained a field needs a value for it when reading an older file. Whether the default is declared in the schema, supplied by the language binding, or required to be the type’s zero value is undecided.

Relationship to application-level versioning. A file-level semantic version for the *application* — as opposed to the format or the schema — has a natural home in the same header slot. These should be one mechanism, not two parallel ones.

12.4 What was considered and declined

A fingerprint alone, with no descriptor table. Compact and a fine integrity check, but a hash can only report that something differs. It cannot say *what* differs, so it cannot implement levels (b) or (c). It also conflates “I changed” with “something I contain changed,” and is unusable for generic types, where `Container<A_v1>` and `Container<A_v2>` hash differently with nothing shared. The fingerprint is kept, but as a fast path on top of the descriptors rather than as a replacement.

Per-value vttables, in the FlatBuffers style, giving true in-place forward compatibility. Declined: it taxes the steady state of every value to solve a problem kladde only has at load. Revisit only if level (d) with preservation becomes a hard requirement for files concurrently shared between application versions.

12.5 Prior art

The model borrowed here is **Avro's**: the writer's schema travels with the data, the reader has its own, and resolution reconciles them at read time. The fingerprint plays the role of Avro's schema-registry key.

What kladde does *not* borrow is Avro's wire format, which is designed for streaming records rather than for a randomly addressable, mutable heap.

Chapter 13

Tooling

What a generic tool may assume about a kladde file it did not write, and how far it can get without knowing the application’s data structures.

Status: design intent. No tools exist yet.

13.1 The premise

A kladde file is self-describing in two independent ways. It describes its **storage**: the [address table](#) is reachable from the header and names every allocation, so the file’s structure is recoverable without any application knowledge at all — and without a scan, since everything is rooted. And it describes its **types**: the embedded [descriptor table](#) says how every non-opaque value in the file is laid out.

Together these mean a tool can be useful at three levels, each requiring strictly more than the last.

13.2 Level 1 — storage analysis

Requires: the [page](#), [address-table](#) and [allocation](#) layers. Requires **no** schema and no application knowledge.

A tool at this level can:

- enumerate every live allocation with its id and size, and every fragment of its content with the page it lives in;
- compute total live bytes, file length, and therefore **garbage**;
- report the per-page live fraction, and how much of the file is superseded pages awaiting reuse;
- report journal length and how far the committed state lags behind it;
- **consolidate** the file into a tight new one, and truncate it.

Rewriting a file from an external tool is safe for exactly the reason it is safe internally: it changes no id, no size, and no content, so it is invisible to any reader. This makes a standalone kladde compact genuinely implementable, which is unusual for a format of this kind.

What it cannot do is tell you what any of it *means* — which allocation holds what, or whether a large one is a container’s backing array or a single opaque blob.

13.3 Level 2 — schema inspection

Requires: additionally the [descriptor table](#). Still requires no application knowledge.

A tool at this level can:

- list every type in the file, with its kind, fields, variants, and inline size;
- compute and display each type’s [fingerprint](#);
- identify the root type;

- report which types are Opaque, and their declaring library and version — which is exactly the list of things a reader would need an implementation of;
- diff two files’ schemas, or a file’s schema against an application’s.

This is what makes “why won’t my application open this file?” a diagnosable question rather than a guess.

13.4 Level 3 — structural export

Requires: additionally the ability to walk the value graph from the root. Requires no application knowledge, but is **inherently partial**.

A tool at this level can render the file as text — JSON, or something more structured — by starting at the root type and recursing. Primitives decode by their code, structs by their fields, enums by their discriminant.

It stops at two boundaries:

Opaque types. By definition their content is not decomposable. A tool can report identity, version, and inline size, and can dump the bytes, but cannot interpret them. This includes the built-in container types, which is a significant limitation — see below.

Pointers. A tool can follow the ownership graph only if it can tell which bytes are pointers. Today that knowledge lives in the language bindings, not in the schema, which is why the [Pointer descriptor kind](#) is reserved.

13.5 The container problem

The most significant gap between this design intent and what is achievable today is that the **built-in containers are Opaque**.

A vector is currently described as an opaque type with a library name, a version, and an inline header size. So a level-3 tool can see *that* a field holds a vector, but not its length, not its element layout, and not how to reach its contents — even though a vector is one of the most structurally regular things in the file.

Two possible resolutions, neither decided:

1. **Give containers structural descriptors.** A vector’s representation really is describable — a header plus a dense array of fixed-size element slots — and the reserved [Array](#) and Pointer kinds exist partly for this. This would make containers fully walkable by any tool, at the cost of freezing more of their layout into the format.
2. **Register container layouts out of band,** so a tool can be taught about the standard containers without them ceasing to be Opaque in the file. Cheaper, but it means a tool needs a table of known types and is no longer purely self-describing.

TBD. This decision determines how useful generic tooling can actually be, and it is worth making before the format is frozen: option 1 changes the descriptors that files contain, so it cannot be retrofitted silently.

13.6 Cross-implementation testing

Tools are also the natural vehicle for testing the cross-language contract. A conformance suite would consist of files written by each implementation, plus the expected level-2 and level-3 output for each, so that every implementation can be checked against every other’s output without either needing to link the other.

See [Conformance](#).

Chapter 14

Conformance

What an implementation must do to claim it implements kladde, and how that claim is meant to be checked.

Status: intent only. No conformance suite exists. This document describes what one should contain.

14.1 What conformance means

An implementation is conforming if:

1. every file it writes can be opened by every other conforming implementation, and
2. it can open every file that any other conforming implementation writes,

subject to the [compatibility contract](#) — the opening application must implement equivalent data structures for the types the file contains.

Note what this does *not* require. It does not require that two implementations produce the same file for the same sequence of operations. Placement, compaction, and fold timing are all free, so two implementations will routinely produce byte-different files with identical contents, and that is correct.

14.2 Must

- **Header.** Write and validate the magic, format version, and minimum reader version. Refuse to open a file whose minimum reader version exceeds the implementation's own.
- **Fail closed.** Never interpret a file whose [root fingerprint](#) differs from the expected one unless resolution actually succeeded. Silent misinterpretation is a conformance failure, not a quality-of-implementation issue.
- **Stable ids.** Preserve every allocation's id across flushes and across consolidation.
- **Ownership.** Maintain exactly one owning pointer per allocation.
- **The reuse rule.** Write only to [reusable](#) pages, or extend the file ([I2](#)).
- **One fsync before the header.** Issue a header only after an fsync that covered everything it references and the previous header ([I1](#)); treat a failed fsync as fatal for the session.
- **Page framing.** Write the kind, content_size, epoch and crc on every non-header page, and ignore any page whose CRC does not validate.
- **Journal encoding.** Encode records, transactions, and journal pages [as specified](#), and recover a torn tail by keeping the longest valid prefix — without skipping a hole.
- **The start of a session.** Fold a non-empty recovered journal before appending to it, and write the journal page that a session's first flush names ([the start of a session](#)).
- **Transaction atomicity.** Replay every transaction completely or not at all.
- **Valid transaction boundaries.** Order records so that every transaction takes a valid state to a valid state ([ordering](#)).
- **Bounds.** Support the [stated bounds](#), and fail cleanly rather than wrap when an application exceeds them.

- **Descriptor encoding.** Produce byte-identical output to the [canonical encoding](#) for the same type graph.
- **Fingerprints.** Produce bit-identical output to [the fingerprint computation](#), including on recursive types.
- **Crash consistency.** Ensure that a crash at any instant leaves a file that opens.

14.3 May

- Choose any placement policy — which reusable page a flush writes to, and how content is cut across pages.
- Consolidate, or not consolidate, by any strategy.
- Group application operations into transactions however it likes, including not at all.
- Fold at any time, and optimize the fold arbitrarily, so long as the result is indistinguishable from an in-order replay.
- Assign, reuse, and recycle ids however it likes.
- Expose any API shape at all. Nothing above the storage layer is constrained.
- Read none of an allocation's bytes while loading a value from it. A data type may leave any part of its content untouched, and a tool that only compacts or only edits one subtree reads no content at all — so what a file keeps resident must follow from the file, never from which types happened to look.
- Keep state of its own in the [consolidator state](#), and keep, replace, or clear a state it finds there.
- Give a reusable page's blocks back to the file system ([deallocating reusable pages](#)).

14.4 Must not

- Change an allocation's id, size, or content during consolidation.
- Overwrite or deallocate a page that is not [reusable](#), or truncate the file below one or to anything but a page boundary ([the truncation rule](#)). This rule is relaxed for the last journal page, which may be appended to, and if it is empty (i.e., if its prefix of CRC-valid transactions is empty) then the file may be truncated below it.
- Continue a session after a failed `fsync`.
- Return anything but zero for a byte that has not been explicitly written.
- Depend on the descriptor table's index assignment for any semantic purpose.
- Make a page's residency, or any storage-layer decision, depend on whether a data type read from it.
- Emit a fingerprint that depends on table layout, traversal order, or any runtime-incidental state.
- Let the consolidator state affect what any allocation contains, or rely on one without first checking that it is its own and still current ([the consolidator state](#)).

14.5 The suite

The conformance suite should have three parts.

Vector tests — pure functions, checkable without any file I/O.

- A set of type graphs with their expected canonical encodings, byte for byte.
- The same graphs with their expected fingerprints, including recursive and mutually recursive cases, and cases that differ only by table permutation (which must produce equal fingerprints) or only by variant declaration order (likewise).
- Descriptor decoding, including rejection of malformed input.

File tests — a corpus of files, each with a manifest describing its expected [level-2](#) and [level-3](#) interpretation. Every implementation must reproduce the manifest from the file. Files written by each implementation are added to the corpus, so that the suite grows to cover the cross product.

Crash tests — two kinds, checking different things.

- *Journal prefix.* For each of a set of mutation sequences, truncate the resulting journal at every byte offset, open it, and assert that it opens successfully and yields a state that is a valid prefix of the

intended one. This is the only mechanical check on the [ordering discipline](#), which is otherwise a matter of implementer care.

- *Flush interruption.* For each of a set of flushes, simulate a power cut after every individual page write — including reordering the writes, since write-back is not ordered — and assert that the file opens and lands on either the previous committed state or the new one, never in between. This is the mechanical check on [I1](#) and [I2](#).

A complementary check is a **leak detector**: walk every allocation reachable from the root via the type structure, add the allocations the header names directly, compare against the set the address table says is live, and assert they match. That checks the “clean up” half of the ordering discipline, which the prefix test does not.

14.6 Open questions

- Should conformance be **versioned** — an implementation claiming “conforms to spec version N” — and can a level of partial conformance (say, read-only) be meaningful?
- How are reserved features handled? An implementation that encounters a reserved [primitive code](#) or the Array kind must reject it rather than guess, but should that be a conformance requirement or a recommendation?
- Should the suite mandate a minimum level of tooling support, so that every implementation ships something that can dump a file?

Chapter 15

Implementation

The reference algorithms and data structures: what an implementation must maintain in memory to satisfy the [specification](#), and how it maintains it.

Everything here is **language-agnostic**. Pseudocode is written in a Rust-like notation, because enums and pattern matching make the cases concrete and easy to reason about, but nothing here depends on Rust. For implementations in other languages, it is recommended but not required to make substantially the same choices for, e.g., integer widths or fundamental data structures like B-trees or hash maps, both of which follow from the [bounds the specification states](#).

What does *not* belong here is anything a different language would do differently: trait shapes or class hierarchies, memory layout, alignment, the mutation-capture mechanism. Those decisions are documented in the language-specific documentations (currently only [kladde-rs](#)).

15.1 The documents

document	contents
In-memory state	the four structures a loaded file is held in, and why each is the shape it is
Address-table operations	pseudocode for every operation on that state: load, read, write, resize, free, recycle, evict, consolidate
Liveness accounting	pins, coverage, and what makes a statement or a page reclaimable
Id recycling	which recyclable id to hand out next, and why the order matters
Write-phase state	what is tracked between flushes, and why it is derived from the journal
The flush	folding the journal into pages: piece tables, hoisting, scheduling, and what gets written
Consolidation	reclaiming garbage: victim selection, the three candidate kinds, and the budget
Consolidator state	what consolidation carries from one session to the next, and how it checks that it is still current
Transactions and batches	grouping operations into transactions, and the buffering state machine
Related work	where the design comes from, and where it parts company

15.2 The shape of an implementation

A kladde file is currently **read in bulk and never searched on disk**.

That single decision is what keeps this design far simpler than a database engine, and it is worth stating first because almost every other choice follows from it:

- On-file representations are optimized for compactness and cheap incremental edits, never for random access.

- There is no on-disk search tree, no B-tree page splitting, no path copying.
- The address table's [tree of child references](#) is a *directory*, not a search structure: nothing ever descends it by key.
- Epochs are consulted exactly once, during the bulk read, and never again at run time.

What this costs is that opening a file is $O(\text{live bytes})$ and that the whole file image is resident. What it buys is that every read is an in-memory lookup and every flush writes only deltas.

A future feature called [segments](#) will address the cost of bulk reads by allowing application authors to manually declare a subtree of their persisted data structure as being loaded and resolved lazily.

15.3 The four structures

A loaded file is a mirror of its pages plus four indexes over them.

15.3.1 The mirror

An implementation reads pages into its own buffers with ordinary file reads, rather than mapping the file. Two rules keep that from meaning “the whole file, forever”.

Mirror only pages that are reachable, where reachable means *named by a surviving fragment* — not merely named by some Ref that is physically present. The two differ by every Ref that lost its probes to a newer statement, and the difference is not small in a file that has been written to for a while. So the set is accumulated during [resolution](#), from the Bytes fragments the sweep emits, rather than while decoding statements. It might be tempting to instead load each page lazily, the first time something reads from it. The decisive objection is not that a page could be missing — one can always be fetched — but that the resident set would stop being a property of the *file* and become a property of the *application*: nothing obliges a `Persistable` to read anything, so a partial loader, whether a sensor app that mocks the subtree it never touches or a tool that only compacts, would leave the storage layer on a path chosen by application code and exercised only when consolidation happens to relocate a page nobody loaded. Eager loading also turns scattered reads interleaved with `Persistable::load` into one sequential pass over a set that is known in advance, and validates every reachable page's CRC at open rather than at first touch — which may be long after the last backup that could have helped.

Nothing can read a page that only dead Refs name, which is what makes ignoring it safe rather than merely economical: a dead statement wins no probe at any epoch the file can be recovered to, so no read at any recoverable state resolves into that page. The page table agrees by construction, since such a page's [coverage](#) is zero and it is therefore already reusable.

Today this saves little, since a reasonably consolidated file is mostly reachable. It matters once [segments](#) arrive: each segment will live in its own subset of pages, and there is no way to guarantee that a segment's pages are contiguous in the file, so “load one segment” will mean exactly “mirror the reachable pages of that segment”.

Drop the data-page mirror once loading is done. After the load, application reads are served by the values' own in-memory representations and never touch the file, so a Bytes fragment's (`page`, `offset`) is dereferenced only when a flush relocates those bytes or a `Copy/Splice` reads them — both rare, and both able to afford a fresh read.

Address-table pages stay resident by default, for a reason data pages do not share: `Inline` payloads live *inside* them, and [both table-side consolidation mechanisms](#) copy those payloads on every consolidating flush — the [page rewrite](#), which also decodes its victim, and the rotating window. They are also small — on the order of 8 MB for a million allocations. Nothing depends on it, though: only the header must stay, being the write buffer, and dropping the leaves after load, as data pages are dropped, costs one read per table victim and one per table page whose inline payload the window restates.

Why not mmap. Mapping the file would let the operating system's page cache serve as the mirror, with pointer-dereference access and no second copy. It is rejected for now because **I/O errors arrive as SIGBUS rather than as an error value**: a page that cannot be faulted in — a failing disk, or the file truncated by something else — takes the process down, and there is no portable way to catch it. Keeping error handling explicit is worth the extra copy. Should that trade ever be revisited, note also that the mapping must be

read-only, since kladde writes with ordinary writes and `fsync` and mixing the two is only coherent by grace of the platform's unified page cache.

Reading through a cursor. When a data type reads its allocations during `Persistable::load`, the reference implementation hands it an iterator over shared byte slices (fragments) rather than an assembled owned byte slice to avoid unnecessary copies. Over the mirror, such a cursor costs no syscalls at all.

15.3.2 The indexes

1. **The fragment map** — the resolved content view, keyed by `(id, offset)`. Everything else hangs off it.
2. **The statement slab** — one record per *live* statement: where its encoding lives, and how many reasons it has to stay.
3. **The allocation map** — per-id metadata: size, anchor, when the application last wrote it, and two counters that no per-page number could replace: how much description this allocation costs, and how many statements still name its id.
4. **The page table** — per-page kind, epoch, and live-byte counter, bucketed for 0(1) victim selection.

Plus two derived structures rebuilt at open and never persisted: the **id allocator** and the **eviction clock**. A flush adds transient state of its own — the fragments it has taken but not yet stated, and the dirty set it states them from — which it drops when it commits; see [During a flush](#).

Content is never copied out of the mirror while the mirror holds it. Both `Ref` and `InLine` content have a file address — an `InLine` payload lives inside an address-table page, but it is bytes at a known address all the same — so a resolved fragment names a location rather than owning bytes, and a range that resolves to zero needs no bytes at all.

See [In-memory state](#).

Chapter 16

In-memory state

The four structures a loaded kladde file is held in, what each is for, and why each is the shape it is.

Integer widths below follow directly from the [bounds the specification states](#), and are given because they are the same in any language.

16.1 1. The fragment map

The resolved-content view, and the structure everything else hangs off: an ordered map from (id, offset) to Fragment, where each entry describes the resolved content from offset up to the next key with the same id.

```
enum Fragment {
    /// Resolves through a `Ref` or `Inline`. `(page, offset)` points
    /// at the *data*, never at the *framing* of the statement.
    Bytes { page: u32, offset: PageOffset, statement: StatementRef },

    /// Resolves to zero through a `Zero`, `Shrink`, or `Tombstone`.
    ZeroExplicitly { statement: StatementRef },

    /// Resolves to zero because no statement matches these probes.
    ZeroByDefault,

    /// Taken by the flush in progress, and not stated yet; see "During a flush".
    /// The cut turns it into `Bytes` or `ZeroExplicitly`, so no other code path
    /// outside a flush ever meets it.
    Pending(PendingRef),
}

struct PageOffset(u16);           // offset into a page
struct AllocationOffset(u32);     // offset into an allocation; the map's key
struct StatementRef(NonZeroU32); // See below why non-zero.
```

The three permanent variants answer two questions at once — what the bytes *are*, and who is *responsible* for them — and the third is where those answers come apart: a range that resolves by default has a definite answer to the first and none at all to the second. The fourth answers the second with “the flush in progress”, and exists only between a flush taking a range and [stating it](#).

An ordered map, because the query is a predecessor search. Reading offset *p* of allocation *i* means finding the greatest key $\leq (i, p) = O(\log F)$ for *F* live fragments — and a sequential read iterates from there. A hash map cannot answer that; a B-tree can, and its node-at-a-time layout suits a structure that is rebuilt in bulk at open and then edited incrementally.

A fragment stores no length. Its extent runs to the next key for the same id, or to the allocation’s size for

the last one. That keeps a fragment small and a split cheap, and it makes the next rule mandatory rather than merely tidy.

16.1.1 The fragments of an existing id exactly partition [0, size)

A hole is not representable: omitting an entry does not describe a gap, it extends the preceding fragment, and the predecessor lookup then answers with a neighbour's bytes. That is a wrong-bytes bug rather than a missing-data bug — the worst kind to leave representable — so a range that resolves to zero *by default* sits in the map like any other.

The only allocation with no fragments at all is a zero-sized one, where [0, 0) is empty and `Grow(id, 0)` is the allocation's only trace, in memory as on disk.

16.1.2 Where `ZeroByDefault` comes from

A range resolves by default when no statement matches its probes. Two sources, the first of which needs a name: call [previous_size, new_size) the **exposed range** of a growth — the offsets that were outside the allocation before it and are inside it after.

- a bare `Grow`, which matches no probe and therefore leaves its exposed range unowned;
- a `Ref` or `Inline` written at offset > previous_size, whose exposed range [previous_size, offset) it leaves uncovered;
- and, away from any resize, a range that was simply never written: write [0, 10) and then [20, 35), and [10, 20) is matched by nothing.

In the first two cases the range is `ZeroByDefault` **exactly when the id has no anchor with $n \leq \text{probe}$** , and the condition is decisive rather than merely necessary: an anchor `Shrink(id, n)` or `Tombstone` matches every probe from n upward (with $n = 0$ for `Tombstone`), hence every probe in an exposed range, and it outranks everything with older epoch; nothing above it reaches there, since content above the anchor is bounded by the size the allocation had before this flush. And an id with no anchor has never decreased in size, so every content statement it has is bounded by previous_size and none reaches into the gap either.

What an anchor does in general is put a floor under where an unowned hole can survive: `Shrink(id, n)` matches every probe from n up, so holes live only in [0, n), and a `Tombstone`, anchoring at $n = 0$, leaves none at all. That is why a tombstoned id's fragments are always owned, and why a re-allocated id's tombstone holds genuine pins.

16.1.3 Consequences the implementation must honour

- **Coalescing compares the whole variant.** Adjacent fragments merge only when they resolve identically *and* name the same statement, so two `ZeroByDefault` ranges merge but a `Shrink`-owned zero range and a `ZeroByDefault` one must not — they differ in exactly the thing that carries a pin.
- **A size increase adds at most one fragment; a write past the end at most two.** Raising the size either extends the last fragment, when the exposed range resolves the way that fragment already does, or adds one entry for it. A `Ref` or `Inline` at offset > size adds that entry *and* its own, which is the only way a single statement adds two.
- **The blow-up is bounded.** `ZeroByDefault` fragments are separated by owned ones, so they at most double an id's entry count.
- **Every code path that reaches for a fragment's owner must handle the variant that has none.** `ZeroByDefault` carries no statement field at all, so this is a match arm rather than a null check — and it is the arm that is easy to forget, because the other two variants both have one. The concrete case is pin accounting: overwriting a range releases the previous owner's pin, and a `ZeroByDefault` fragment has no owner and no pin to release, so a blind release there would decrement some unrelated statement or panic. The same applies to a consolidator asking "does this fragment belong to the statement I am rewriting?", where the answer for this variant is always no. During a flush `Pending` joins it: it has no owner either, and a source that takes a range already pending simply discards the older entry, with no pin to release.
- **Growth into default territory is free.** If the last fragment is already `ZeroByDefault`, raising the size extends it with no new entry at all, since its end is implied by the size.

16.2 2. The statement slab

One StatementRecord per live statement.

```
struct StatementSlab {
    records: Vec<StatementRecord>, // 8 bytes per slot, indexed by StatementRef
    framing_len: Vec<u8>,          // each entry is <= 21, per the spec's bounds
    free_head: u32,                // first free slot, or 0 for "none"
}

struct StatementRecord {
    /// Address-table page where the statement's encoding lives – not the data
    /// page a `Ref` points into. If `pins == 0`, the index of the next free
    /// slot instead, or 0 to end the list.
    page_or_next: u32,

    /// Number of reasons this statement must stay; see Section "Liveness".
    /// Nonzero for live statements, zero for empty slots.
    pins: u32,
}
```

A slab indexed by a dense integer, not a hash map. StatementRef is a 32-bit index, possible because of the specification's bound that at most $2^{32} - 1$ statements exist. This makes every lookup an array index rather than a hash. Dead slots thread into a free list through page_or_next, which is unambiguous because a slot is free exactly when pins == 0.

Slot 0 is never handed out. That costs one slot once and buys niche optimization for AllocationMeta::anchor, AllocationMeta::grow_witness, and RecyclableId::tombstone. These optimizations are the motivation why the specification bounds the number of statements to at most $2^{32} - 1$ rather than 2^{32} . Reserving slot 0 also makes for a simple free-list terminator (page_or_next == 0), although this argument is not load bearing since the end of the free-list could also be indicated by a self-reference instead.

A record exists exactly while pins > 0. At the 1 → 0 transition its framing is released from its page's coverage, and the slot is freed. A dead statement then survives purely as bytes in its page until a rewrite of that page decodes it and drops it.

This has a consequence worth naming early, because several arguments rest on it: **an id's physically present statements are not reachable from memory.** The fragment map indexes only live fragments, the slab holds only live statements, and the allocation map's mentions is a count rather than a list.

Ordering obligation. On the 1 → 0 transition, read the page number and release its coverage *before* overwriting page_or_next with the free-list link.

Stale references cannot arise, which is what lets the slab recycle slots freely: every holder of a StatementRef — a fragment, an anchor, a grow witness, or a recyclable id's tombstone — gives it up as part of the same event that removes the statement's last pin. That argument rests on every release site being correct, so a debug-only parallel array of generation counters is worth keeping in mind: it turns a violation into an assertion rather than silent corruption, at no cost in release builds.

16.3 3. The allocation map

A hash map from id to per-allocation metadata.

```
struct AllocationMeta {
    size: u32,
    fragment_count: u32,
    statement_bytes: u32,
    mentions: u32,
    last_written: u32, // the flush that last wrote the id; see below
    anchor: Option<StatementRef>, // newest Shrink or Tombstone
}
```

```
grow_witness: Option<StatementRef>, // Grow with n == size, if any
}
```

A hash map, because the query is a point lookup. Nothing ever iterates the allocation map in id order at run time; consolidation's id-range sweep uses the fragment map, which is already ordered.

field	updated when	read for
size	any flush that resizes the id, writes past its end, frees it, or re-allocates it	answering size() in O(1); bounding reads; supplying the extent of an id's last fragment
fragment_count	on every fragment created or destroyed for this id	pricing description defragmentation — description overhead relative to size. Counts ZeroByDefault fragments, which are real entries with real memory cost
statement_bytes	when a statement for this id is created, or when one's pins reach zero	pricing defragmentation, paired with fragment_count
mentions	+1 per statement naming the id written or read at open; -1 when one is physically dropped	one thing only: releasing a tombstone anchor's pin when the count falls to 1
last_written	whenever the application's writes reach the id — a flush that writes, resizes, or allocates it	the age of description defragmentation's candidates
anchor, grow_witness	anchor: when a Shrink or Tombstone is written. grow_witness: when a Grow is written, and when size moves past its bound	moving the anchor pin; telling a consolidator that a victim page holds an anchor it must replace

size is maintained **incrementally**, never recomputed: it is $\max(\text{anchor's } n, \text{Grow bounds and content extents above anchor_epoch})$, and every operation that could change it knows which term it changed.

last_written measures content age, where an epoch measures location age. A statement's epoch [is its page's](#), so every restatement re-stamps it — the header re-stamps everything it holds on every flush — whereas last_written moves only when the application writes the id. It counts flushes in 32 bits from a base the session sets at open, and a session that would outrun them rebases every entry in one pass. At load it is restored from the [consolidator state](#); an id the state has no current record for falls back to the youngest page that holds one of its fragments, which can only understate the age, since content is never younger than the page that holds it; so it errs toward treating an id as recently written.

16.3.1 A tombstoned id keeps no allocation-map entry

The flush that frees an id removes it and moves the only two fields a non-existent id still reads into the id allocator's recyclable set:

```
struct RecyclableId {
    mentions: u32,
    tombstone: Option<StatementRef>, // None once the tombstone dies, at mentions == 1
}
```

Everything else is either constant for a non-existent id — size 0, fragment_count 0, grow_witness absent — or already recorded in the tombstone's own slab entry.

So a page rewrite that decodes a statement looks its id up in the allocation map first, and in the recyclable set if it is not there. The tombstone reference is what the pin release must reach, what a consolidator consults before re-emitting a tombstone it has decoded, and what becomes the anchor if the id is allocated

again. Within the recyclable set mentions only falls, since nothing writes a statement naming a non-existent id; at 0 both fields are spent and the entry is a bare recyclable id.

16.4 4. The page table

For every page: its page number, kind, epoch, and live-byte counter (**coverage**).

The counter is 32 bits rather than page-sized, because a page's live bytes exceed its capacity whenever several Ref statements claim the same bytes — which is legal, and **bounded** precisely so that 32 bits suffice.

Pages are bucketed by live fraction, a handful of buckets suffices, together with an age mark. That makes victim selection $O(1)$ rather than a priority queue's $O(\log P)$, with $O(1)$ bucket moves as counters change. Data and non-header AddressTable pages are bucketed **separately**, and header pages in neither. The header is excluded here since it is rewritten every flush, so it is never a victim, and its content is accounted for by being **taken and stated again by every flush** rather than by its coverage.

The in-memory state of an AddressTable page carries one more field: its parent's page number, four bytes, updated whenever the parent is replaced. **Unlinking an emptied page** needs it, since a page may be named by an interior page rather than by the header.

An ordered set tracks reusable pages under the **two-generation quarantine**, handing out the lowest first: a page dropped by commit E becomes writable in flush $E + 2$.

16.5 Derived structures

Two more, rebuilt at open and never persisted:

- **The id allocator** — the next fresh id, plus the recyclable set described above. See **Id recycling**.
- **The eviction clock** — over the fragments the header states, recording how many flushes each has gone untouched, which open seeds from their allocations' last_written. See **The flush**.

16.6 During a flush

A flush changes the structures above in two steps: it **takes** the fragments it changes as it goes, and it **states** them only when it cuts the address table. **Why the two are separate** belongs to consolidation; between them, three pieces of transient state exist, all dropped when the flush commits.

Pending fragments. A fragment the flush has taken names an entry in a per-flush table rather than a statement:

```
struct Pending {
    origin: Origin,    // where its bytes are now
    place: Place,      // where they will be stated from
    heat: u32,         // flushes untouched, for the header's split
}

enum Origin {
    Arena(ArenaPos),  // written this flush
    File(Address),     // already in the file
    None,              // zeros: nothing to read
}

enum Place {
    Unplaced,          // a chunk: `pack` gives it a data page
    Data(Address),     // in a data page, placed or left where it was
    Inline,            // in the address table, as an `Inline` payload
    Zero,              // a `Zero` statement's range
}
```

A pending fragment has no owner and so holds no pin; pins are handed out when the cut binds its statement. Its coverage follows its place: a fragment in Data is charged to that page exactly as a Bytes fragment is, an Unplaced one is charged when pack places it, and an Inline payload when the cut decides which table page carries it.

The dirty set. Every key range the flush has taken, kept merged and sorted, and per touched id a record of what the fragment map cannot say: the size and existence the id had when the flush began, whether it was freed, and whether a page the flush retires held its anchor or grow witness. The cut [derives every statement](#) from these two.

Heat rides on the pending entries rather than on statements, since statements are derived afresh every flush: a statement derived from several fragments takes the hottest of them.

16.7 Cost

Maintenance of all of the above costs $O(\log F)$ per fragment created or destroyed, and a flush creates or destroys at most a small multiple of the statements it writes.

Epochs belong to pages, not to statements. A statement inherits the epoch of the page it currently sits in, so moving a statement re-stamps it. The header is rewritten every flush and states resolved truth, so everything in the header always carries the current epoch, and multi-epoch disagreement can exist only between the header and evicted leaves, or between two leaves. It also means an epoch says when a statement was last *moved*, not when its content was last written, which is why the allocation map keeps [last_written](#) as well.

Chapter 17

Address-table operations

Pseudocode for every operation on the [in-memory state](#).

Two levels are in play and it is worth separating them before reading any of this. The **application level** mutates a value’s native in-memory representation and appends a record to the journal; it does not touch the fragment map at all. The **storage level** — everything below — describes what the *file* currently says, and changes only when a flush writes statements or a page is rewritten. So `write_bytes` below is not what `vec.push()` calls; it is what a flush calls once, after folding a whole journal.

Notation: Rust-like, with ? for “may fail”, and with the obvious map operations left undefined. F is the number of live fragments, S the number of live statements.

17.1 Reading

17.1.1 `resolve(id, probe) -> Fragment`

```
fn resolve(id: Id, probe: AllocationOffset) -> Option<(Fragment, Range)> {
    let meta = allocations.get(id)?;
    if probe >= meta.size { return None; }           // outside the allocation

    // Predecessor search: greatest key <= (id, probe).
    let (key, fragment) = fragments.range(..=(id, probe)).next_back()?;
    debug_assert!(key.id == id);                     // the partition invariant

    let end = match fragments.range((id, probe+1)..).next() { // `probe+1` can't overflow because `probe` is a valid offset
        // If a B-tree implementation supports `fragment.next` in O(1) time then
        // that should be preferred over the above second `range` query.
        Some((next, _)) if next.id == id => next.offset,
        -                               => meta.size, // last fragment of this id
    };
    Some((fragment, key.offset..end))
}
```

$O(\log F)$. The end computation is where “a fragment stores no length” is paid for, and it is one extra step on the same cursor.

17.1.2 `read(id, offset, len) -> bytes`

```
enum Chunk<'a> {
    Bytes(&'a [u8]), // borrowed from the in-memory page mirror
    Zero { len: u32 },
}
```

```

fn read(id: Id, offset: AllocationOffset) -> Result<impl Iterator<Item = Chunk> + Read + Seek> {
    // Below is rust-like pseudocode using a hypothetical `yield` statement for generator
    // shorthand. The real rust implementation manually implements the state machine for
    // the iterator and also trivially implements `Read` and `Seek` on top of the iterator
    // (so that callers can easily read integers using `byteorder::ReadBytesExt`).

    let size = allocations.get(id).ok_or(InvalidPointerDereferenced)?.size;
    if offset >= size { return Ok(empty()); } // also covers every zero-sized id

    let (key, mut fragment) = fragments.range(..=(id, offset)).next_back().unwrap();
    debug_assert!(key.id == id); // the partition invariant, given offset < size
    let mut fragment_offset = key.offset;
    let mut skip = offset - fragment_offset;

    // Iterate over the fragments that follow `fragment`. In rust, this currently requires
    // a second B-tree descent due to a limitation of rust's B-tree implementation.
    // `offset + 1` cannot overflow, since we verified `offset < size <= u32::MAX` above.
    let mut iterator = fragments.range((id, offset + 1)..);

    while fragment_offset < size {
        let (next_fragment_offset, next_fragment) = match iterator.next() {
            Some((key, fgmt)) if key.id == id => (key.offset, fgmt),
            // `next_fragment` can be any `Fragment`; the loop exits before reading it.
            _ => (size, Fragment::ZeroByDefault),
        };
        // Assumes a constructor `Chunk::from_fragment(fragment, start, end)`
        // where `start` and `end` are relative to `fragment`.
        let chunk = Chunk::from_fragment(fragment, skip, next_fragment_offset - fragment_offset);
        fragment = next_fragment;
        fragment_offset = next_fragment_offset;
        skip = 0; // Only the first fragment can have some skipped head bytes.
        yield chunk;
    }
}

```

$O(\log F + k)$ for k fragments spanned: one predecessor search, then a walk of amortized $O(1)$ steps. The cursor is what buys that bound, and it is easy to lose. Calling `resolve` once per fragment instead might read more naturally but would cost $O(k \cdot \log F)$, because every call would be a fresh search.

There is no `len` parameter, because the iterator is the caller's to stop: it runs to the end of the allocation, and a caller that does not want to read to the end simply drops the iterator when it's done reading. The guard on `offset >= size` is what keeps the `debug_assert` honest — without it, reading at or past the end of an allocation, and reading any offset of a zero-sized one, lands the predecessor search on some other id's last fragment.

17.1.3 `size(id)` and `exists(id)`

```

fn size(id: Id) -> Option<u32> { allocations.get(id).map(|m| m.size) }
fn exists(id: Id) -> bool { allocations.contains(id) }

```

$O(1)$ both, because `size` is maintained incrementally rather than recomputed.

17.2 Opening a file

Loading is the one place where the whole address table is processed at once, and the only place where epochs are consulted. It is worth doing properly, because it is $\tilde{O}(\text{file})$ while everything else is $\tilde{O}(\text{change})$.

17.2.1 Three facts about the physical format that decide the algorithm

1. A page is a run of constant epoch, since every statement inherits its page's epoch.
2. A page is sorted by (id, offset), because id_delta and offset_delta are unsigned variants — the cursors only ever move forward.
3. Within one epoch, no two statements claim the same byte, which is the no-conflicts rule.

Two consequences carry the whole load:

Fact 2 makes loading a k-way merge of P sorted runs rather than a sort of m statements. Facts 1 and 3 together bound the sweep's open set by the number of distinct live epochs, not by the number of statements.

The first replaces $O(m \log m)$ with $O(m \log P)$, and P — the number of live address-table pages — is smaller than m by whatever the average statement density is, on the order of 600×. More importantly it means no sorted copy of the statement array is ever materialised: the merge streams.

17.2.2 The pipeline

```
fn open(file) -> std::io::Result<State> {
    let header = pick_header(file)?;           // CRC-valid, highest epoch
    file.fsync()?;                             // see "Why the fsync is here", below

    // Every page starts reusable and is claimed as it is discovered. After the
    // fsync above, the older header's world is retired, so "not claimed" and
    // "reusable" coincide and no page needs an epoch until something writes it.
    let mut state = State::with_pages(file.len().div_ceil(page_size));

    // 1. Walk the address-table tree, decoding every page and keeping it resident.
    //    The tree is a directory, never searched; a plain traversal suffices.
    let mut runs = Vec::new();                 // one decode cursor per page
    let mut seen_addr_table_pages = Set::new();
    let mut queue = vec![header.page];
    while let Some(p) = queue.pop() {
        if !seen_addr_table_pages.insert(p) { return Err(NotATree); }
        // Decode up to first statement and validate CRC:
        let (page, stmt_decoder) = decode(file.page(p))?;
        if page.epoch > header.epoch { return Err(DurabilityContractViolated); }
        state.pages.insert(p, page.kind, page.epoch);
        queue.extend(page.children);
        runs.push(stmt_decoder);               // sorted by (id, offset); epoch = page.epoch
    }

    // 2. One merge, one group at a time. Every index is filled from the same
    //    pass: the allocation map, the slab, the fragment map, coverage, the
    //    set of data pages to mirror, and the recyclable ids.
    let mut merged = merge(&mut runs).peakable(); // iter over stmts in (id, offset) order
    let mut emitted_fragments = Vec::new();       // will be emitted in sorted order
    while let Some(id) = merged.peek().map(|s| s.id) {
        resolve_id(&mut state, id, &mut merged, &mut emitted_fragments);
    }
    state.fragments = BTree::from_sorted(emitted_fragments); // real rust: `BTreeMap::from_iter`

    // 3. The eviction clock covers only what the header holds, and is rebuilt there.
    state.clock = EvictionClock::from(header.statements());

    // 4. Replay the journal segment the header names, stopping at the first
    //    failing check. Its pages are live too, and a non-empty journal is
    //    folded by a flush before anything is appended to it.
    for txn in file.journal(header.journal_pointer).valid_prefix() { replay(txn); }
```

```
Ok(state)
}
```

Why the fsync is here. It is the one [durability](#) requires of a loader that wants to retire the older header's world immediately, and what it must precede is the first page handed to the allocator. Putting it later would be a bug of the silent kind: every page the first flush reuses would be a page recovery might still have needed. Putting it before loading the header would also work but would incur the cost of an unnecessary fsync if `pick_header` fails.

The merge. merge holds one decode cursor per address-table page in a binary min-heap keyed by that cursor's current `(id, offset)`, pops the smallest, advances that cursor, and pushes it back. Because each run is sorted by `(id, offset)`, the output is globally sorted by `(id, offset)`, so an id's statements arrive contiguously and `resolve_id` can consume them straight from the merge, one peek ahead, with nothing buffered. $O(m \log P)$, and the only structures alive at any moment are the P decode cursors and a heap bounded by the number of distinct live epochs — so the working set is $O(P)$ rather than $O(m_i)$ for the widest id.

A tombstone carries no offset, and nonetheless sorts exactly where its match starts. [The delta encoding](#) leaves `offset_cursor` untouched for a Tombstone, which looks at first like a statement whose position says nothing about its match range of `[0, size)`. It is not, and the reason is the no-conflicts rule: a tombstone denies its id's existence, every other statement kind asserts it, and a page is a single epoch — so **a tombstone is the only statement naming its id anywhere in its page**. Its `id_delta` is therefore nonzero, which resets `offset_cursor` to 0, and nothing increments it again. A tombstone therefore decodes at `offset_cursor == 0`, which *is* its match start. So the merge needs no special case, and `resolve_id` needs no pre-pass: every statement's sort key is its match start, by three different routes — `offset` for content, `n` for a Shrink, and a forced cursor reset for a Tombstone.

17.2.3 Resolving one id

One pass over the buffered group, and no sorting. The size is not computed separately and then fed to the sweep: it comes *out* of the sweep, along with the fragments, the anchor, and everything else `AllocationMeta` holds.

Each statement matches an interval, and nothing is clipped:

statement	match range
Ref, Inline, Zero	<code>[offset, offset + stmt_size)</code>
Shrink(<i>n</i>)	<code>[n, u32::MAX)</code> — unbounded
Tombstone	<code>[0, u32::MAX)</code> — unbounded
Grow(<i>n</i>)	<code>[n, n)</code> — empty, so it never wins anything

Grow is the odd one, and giving it an empty interval *at its bound* is what lets it take part at all: it is rejected by `if s.end() > pos`, so it can never shadow anything, while [the delta encoding](#) already sorts it at `n`, which is exactly where the sweep needs to meet it.

The fragment covering a stretch is the matching statement with the highest epoch, so what the sweep computes is the **upper envelope by epoch** of these intervals.

```
fn resolve_id(state: &mut State, id: Id, stmts: &mut Peekable<Merge>,
    emitted_fragments: &mut Vec<Fragment>) {
    // `stmts` yields this id's statements in match-start order, tombstones included,
    // and then the next id's; see the merge, above.
    // `open` is a max-heap by epoch. Entries are (epoch, end, statement). By fact 3 it holds
    // at most one entry per distinct epoch at any point in the iteration below, so its size
    // is bounded by the number of live epochs – in practice a handful, never more than P.
    let mut open = MaxHeap::new();
    let mut newest = None;           // highest epoch seen; decides existence
    let mut grow = None;           // largest admitted `Grow`; see below
    let mut mentions = 0;          // physically present statements naming this id
```

```

let mut pos = 0;
let mut current: Option<Decoded> = None;    // winner of the run starting at `run_start`
let mut run_start = 0;

loop {
  while open.peek().is_some_and(|e| e.end <= pos) { open.pop(); } // lazy expiry
  while stmts.peek().is_some_and(|s| s.id == id && s.start() <= pos) {
    let s = stmts.next().unwrap();
    mentions += 1;           // one per statement consumed, and none is consumed twice
    newest = newest_of(newest, s);
    if s.end() > pos {
      open.push(s.epoch, s.end(), s);
    } else if s.is_grow() && beats(s, open.peek()) {
      grow = Some(s);        // `Grow`s arrive in increasing bound order, so
                             // the last admitted one is the largest
    }
    // expired statements fall out here too
  }

  let winner = open.peek();    // None => this stretch resolves to zero by default
  if winner != current {
    if pos > run_start { emit(state, id, run_start..pos, current, emitted_fragments); }
    (current, run_start) = (winner.copied(), pos);
  }
  // Stop once nothing is left to open and the winner can no longer change. An
  // unbounded winner is a `Shrink` or `Tombstone`, and it is then the anchor.
  let next_start = stmts.peek().filter(|s| s.id == id).map(|s| s.start());
  if next_start.is_none() && winner.map_or(true, |w| w.is_unbounded()) { break; }
  // The winner can only change where a statement starts or where it ends.
  // Statements that end while shadowed change nothing and are skipped over.
  pos = min(next_start.unwrap_or(u32::MAX),
            winner.map_or(u32::MAX, |w| w.end));
}

let size = max(run_start, grow.map_or(0, |g| g.n()));
if run_start < size {           // the tail an admitted `Grow` opened up
  emit(state, id, run_start..size, current, emitted_fragments);
}
finish(state, id, mentions, size, newest, current, grow);
}

```

17.2.4 What the sweep leaves behind

- **size = max(run_start, the admitted Grow's bound).** Nothing accumulates content claims; the next subsection is why.
- **The anchor is the winner at termination.** The unbounded statements are exactly the Shrinks and Tombstones, and the heap orders by epoch, so whichever is on top when the sweep stops is the newest of them — the anchor, by definition, tracked for free.
- **Existence is the one fact the envelope cannot supply.** `newest` is therefore maintained in the consumption loop, at one comparison per statement and no extra pass, and `exists` is `!newest.is_tombstone()`. The shortcut “the winner at termination is a tombstone and the size is zero” looks equivalent and is not: `Tombstone@3` followed by `Grow(0)@5` re-allocates the id at size zero and satisfies it.
- **A non-existent id takes the other exit.** No allocation-map entry; a recyclable-id entry carrying mentions and the tombstone; and exactly one slab slot, for the tombstone, holding its A pin. Every other statement of the group was outranked at every probe and gets none.
- **mentions is counted in the consumption loop,** one per statement, since every statement the sweep consumes under this id names it and the id's run in the merge is exactly its physically present statements. Counting there rather than measuring a buffer afterwards is what lets the group stay unbuffered.

- **The grow witness is the admitted Grow exactly when it *determined the size***, that is when its bound exceeds `run_start`. Where the size came from `run_start` instead, it is witnessed by a content statement or by the anchor, both of which are pinned already.

17.2.5 Why `run_start` is the size

At termination the winner is unbounded or absent, and `run_start` is where that final run began. **That position is the spec's `max(anchor.n, max{claim : epoch > anchor_epoch})` restricted to statements that match something — and the heap has done the epoch filtering for free:**

- a content statement *older* than the anchor cannot win at or above `n`, because the anchor outranks it there, so it can never push the takeover point past `n`;
- a content statement *newer* than the anchor either wins up to its own end, or is shadowed there by something newer whose claim is at least as large.

With no anchor at all, the winner at termination is `None` and `run_start` is where the envelope ended — the unfiltered maximum, which is correct, because with no anchor nothing is filtered.

Two things follow. No accumulator over content claims is needed, which is what makes the single pass cheaper than computing the size separately rather than dearer. And nothing has to be clipped, because a bounded statement can never win above `size`: newer than the anchor, and `size` would already cover its claim; older, and the anchor outranks it there.

17.2.6 Why Grow needs its own rule

The above algorithm has a dedicated branch `else if s.is_grow() && beats(s, open.peek())` because Grow would otherwise not be observable: it is never the winner because it is (correctly) never even pushed on the heap (`open.push` is gated by `s.start() <= pos` and `s.end() > pos`, which can't both be true for a Grow with `start == end`). Therefore, a Grow that is buried inside a sequence of statements and not the newest is unobservable after exiting the loop. For example, consider the following two sequences of statements, which only differ in the bold epoch number:

- `Shrink(10)@7, Ref(20, 5)@11, Grow(50)@3, Ref(100, 10)@5` (correct size: 25) and
- `Shrink(10)@7, Ref(20, 5)@11, Grow(50)@9, Ref(100, 10)@5` (correct size: 50)

Both exit the loop with the same state because `newest` is the same for both (`Ref@11`), the last read statement is the same (`Ref@5`) and thus `pos` and `run_start` exit the loop with the same value, and the only statement that differs between the two sequences (`Grow@3` vs `Grow@9`) never came up as winner. Thus, a sweep that doesn't explicitly keep track of the leading candidate for the grow witness could not distinguish these two sequences, and it would incorrectly assign the same size to both.

The pair rules out the two lesser repairs as well, which is worth noticing because each looks like it might avoid the branch. Admitting every Grow without the `beats` test would take `Grow(50)@3` in the first sequence and report 50 where the truth is 25 — the `Shrink@7` above it is exactly what denies it. Taking the sweep's final `pos` instead of `run_start` would report 100 for both, since `pos` is dragged out to the last statement's start whatever its epoch. Only the epoch comparison against what is open at the Grow's own position separates the three answers.

Since the root of the issue is that Grow never comes up as winner due to its empty range, it might be tempting to assign Grow an unbounded range `[n, u32::MAX)` (like `Shrink`) instead. This would indeed make Grow observable in the sweep but wrongly assign fragments to it: consider `Ref(0, 500)@3, Shrink(10)@5, Grow(100)@9, Ref(150, 50)@11`, which, with an unbounded range for Grow would result in a split Zero fragment: `[10, 100)` assigned correctly to `Shrink@5` and `[100, 150)` assigned incorrectly to `Grow@9`. This would be a shortcoming of the algorithm, not of the spec: the invariant that Grow never owns a fragment, which allows it to be consolidated earlier than `Shrink`, is the whole reason why the format distinguishes between Grow and `Shrink` instead of defining only a single `Size` statement.

17.2.7 Four details that carry the correctness

Each is easy to get wrong:

- **The heap, not a stack.** A shadowed statement re-emerges when the statement covering it ends — `Zero(0, 100)@3` under `Ref(0, 10)@9` wins `[10, 100)` — so the open set has to be a priority queue and entries must survive being overtaken.
- **Lazy expiry is enough.** Only the top matters, so entries that expired while shadowed can be discarded whenever they surface. Each statement is pushed once and popped once: $O(m_i \log P)$ for the whole sweep.
- **Emit on winner change, not on event.** Advancing `pos` to a start that loses, or to an end that was shadowed, must not cut a fragment — which is exactly what makes the output **minimal**, with neighbours already joined, rather than something a coalescing pass has to clean up afterwards.
- **The loop runs past the end of the group.** Exhausting the statements is not a stopping condition on its own, because bounded winners are still expiring and expiry is what uncovers what lies beneath them. Over `Ref(0, 1000)@3`, `Shrink(500)@5` and `Ref(0, 600)@7`, the `Shrink` is opened at 500 and shadowed there; only when `Ref@7` expires at 600 does it surface, which is also where the size turns out to be.

Nothing denies a truncated statement except the sweep itself, which is worth stating because the instinct is to look for a clipping rule. Over `Ref(0, 1000)@3`, `Shrink(10)@5` and `Ref(100, 20)@7` the old `Ref`'s interval is left running to 1000; the `Shrink` simply outranks it from 10 upward, so the sweep emits `[0, 10)` to `Ref@3`, `[10, 100)` to the `Shrink`, and `[100, 120)` to `Ref@7`, and stops with the size at 120. Clipping intervals at the anchor instead would be a bug, because `[0, n)` is precisely where an older statement still wins.

17.2.8 Why a sweep, and not an intermediate tree

The sweep writes the fragment map directly, in the order the map wants, and needs no structure bigger than a heap of a handful of entries. Two alternatives are worth naming, because both are the natural first idea and both are worse here.

An interval tree, or a segment tree keyed by epoch. Building one costs $O(m \log m)$ and a second allocation the size of the statement set, and it answers a question the load never asks — “who wins at this probe” — one probe at a time. Its output also arrives in whatever order the extraction walk produces, so the fragments still have to be sorted before they can be bulk-loaded. It would earn its keep if probes arrived interactively; at load they do not.

Painting newest-first into a set of unclaimed gaps. Sort the id's statements by descending epoch, and let each claim only the parts of its range that nothing newer has taken; with a union-find “next unclaimed offset” this is $O(m \alpha(m))$ after the sort, which is great asymptotically. But it needs the group in epoch order, which the merge does not give and which would cost a sort per id; and it produces fragments in epoch order, which is not the order the fragment map is built in, so it needs a second sort at the end. Two sorts to avoid a heap of five entries is a bad trade.

The sweep is what fits the data the format actually hands us: already sorted by offset, already grouped by epoch.

17.2.9 What falls out of emission

`emit` is where every other index is filled, and the reason the load is a single pass rather than four:

- **Slab slots are allocated on first ownership.** A statement gets a `StatementRef` the first time it wins a stretch, and a statement that never wins one — and is neither the anchor nor the grow witness — is **never given a slot at all**. That is what makes the slab hold exactly the live statements, and it is why dead statements cost memory only as the bytes they occupy in their page.
- **Pins are counted, not computed.** Each emitted fragment adds one `F` pin to its owner; the anchor and the grow witness each add their `A` pin once, at the end of the group.
- **Coverage is accumulated twice per statement, for different reasons.** A live statement charges its framing to the address-table page holding it; each `Bytes` fragment charges its length to the page holding the data, which for an `Inline` is that same table page.
- **The set of data pages to mirror is the set of pages named by emitted `Bytes` fragments**, and it is also the set of data pages the page table marks live. Physically present `Refs` that lost every probe name pages that nothing can read, and **those pages must not be mirrored** — nor kept alive. Their

epochs are read from their framing as they are mirrored; a page nothing claims needs no epoch, because it is already reusable.

- **A non-existent id goes to the recyclable set instead of the allocation map**, carrying mentions and a reference to its tombstone.

Then the fragment map is built in bulk. Ids are processed in ascending order and each id’s fragments are emitted in ascending offset, so the emitted stream is globally sorted by (id, offset) — which lets the B-tree be built bottom-up in $O(F)$ from the sorted sequence, instead of F separate $O(\log F)$ insertions into a tree that rebalances all the way up. This is the single largest constant-factor win available at load, and it is free: it only requires that nothing reorders the stream between the sweep and the build.

17.2.10 Cost

step	cost
decode every address-table page	$O(\text{address-table bytes})$
merge	$O(m \log P)$
resolve, all ids	$O(m \log P)$, and $O(m)$ when few epochs overlap
build the fragment map	$O(F)$
replay the journal	$O(\text{journal bytes})$

No step sorts, and no step allocates a second copy of the statement set. The remaining $O(\text{live bytes})$ term is reading the file itself, which is the bound [the specification states](#) and the one the design is willing to pay.

17.3 Fragment-map primitives

Everything that changes content goes through these three.

17.3.1 split(id, at)

Ensure a fragment boundary exists at at, so that a range edit has somewhere to start and stop.

```
fn split(id: Id, at: AllocationOffset) {
    let (fragment, range) = resolve(id, at).unwrap();
    if range.start == at { return; } // already a boundary
    let shifted = match fragment {
        Bytes { page, offset, statement } =>
            Bytes { page, offset: offset + (at - range.start), statement },
        Pending(p) => Pending(pending.split(p, at - range.start)), // shifts origin and place
        other => other, // zero fragments carry no offset
    };
    if let Some(s) = shifted.statement() { pin(s); } // the statement now owns two
    fragments.insert((id, at), shifted);
    allocations[id].fragment_count += 1;
}
```

A split **adds a pin**, because one statement now owns two fragments. This is the operation behind “pins can rise after creation”.

17.3.2 overwrite(id, range, new: Fragment)

```
fn overwrite(id: Id, range: Range, new: Fragment) {
    split(id, range.start);
    if range.end < allocations[id].size { split(id, range.end); }

    for (key, old) in fragments.drain(range) { // strictly inside
```

```

    if let Some(s) = old.statement() { unpin(s); }
    allocations[id].fragment_count -= 1;
    if let Some(page) = old.data_page() { coverage[page] -= length_of(key); }
}
if let Some(s) = new.statement() { pin(s); }
if let Some(page) = new.data_page() { coverage[page] += range.len(); }
fragments.insert((id, range.start), new);
allocations[id].fragment_count += 1;
coalesce_around(id, range);
}

```

The only non-trivial primitive: split at both boundaries, drop what is strictly inside, insert the new one. `data_page()` is the page a fragment's bytes sit in — a Bytes fragment's, or a pending one's whose place is Data — so a range taken in place is released and charged again to the same page, and a pending fragment [carries its coverage](#) exactly as a stated one does.

17.3.3 `coalesce_around(id, range)`

Merge each new boundary with its neighbour when both fragments resolve identically **and** name the same statement. Two `ZeroByDefault` ranges merge; a `Shrink-owned` zero range and a `ZeroByDefault` one must not. Two Bytes fragments merge only when they are contiguous in the same page *and* owned by the same statement. Pending fragments are left alone: they have no statement to compare yet, and the cut merges them when it derives one.

Merging releases one pin and decrements `fragment_count`.

17.3.4 `pin / unpin`

```

fn pin(s: StatementRef) { slab.records[s].pins += 1; }

fn unpin(s: StatementRef) {
    let rec = &mut slab.records[s];
    rec.pins -= 1;
    if rec.pins == 0 {
        let page = rec.page_or_next; // read BEFORE overwriting
        coverage[page] -= slab.framing_len[s];
        allocations[id_of(s)].statement_bytes -= slab.framing_len[s];
        rec.page_or_next = slab.free_head; // now reuse the field
        slab.free_head = s;
    }
}

```

The ordering in the `pins == 0` branch is the obligation named in [In-memory state](#): the page number must be read before the field becomes a free-list link.

17.4 Taking and stating

A flush changes the fragment map in two steps, and only the second creates statements: every source **takes** the fragments it changes while the flush runs, and the cut **binds** a statement to them once it knows what to state and in which page. [Why the two are kept apart](#) is consolidation's story; the mechanics are these.

17.4.1 `take(id, range, pending, dirty)`

```

/// Re-owns `range` for the flush in progress. Its old owners lose their pins and
/// its coverage now; the statement that will own it is the cut's to derive.
fn take(id: Id, range: Range, p: Pending, dirty: &mut Dirty) {

```

```

    overwrite(id, range.clone(), Fragment::Pending(pending.push(p)));
    dirty.ranges.insert(id, range); // merged with whatever it touches
}

```

Taking is how every source changes content — the fold writing, a victim’s survivors moving, a description defragmentation rewrite, a page rewrite or the rotating window restating, the header being carried forward. What differs between them is only the [Pending entry](#): whether the bytes stay where they are, need a page, go inline, or are zeros.

17.4.2 bind(stmt, page, run)

```

/// Gives `stmt` its slot and its fragments, once the cut has decided which page it
/// goes in. `run` is the pending fragments the cut derived it from.
fn bind(stmt: Statement, page: PageNumber, run: &[Key]) {
    let (id, framing) = (stmt.id(), framing_of(&stmt)); // encoded length minus payload
    let s = slab.alloc(page, framing); // pins = 0 for now
    *mentions_of(id) += 1; // the recyclable set's, for a `Tombstone`
    *statement_bytes_of(id) += framing;
    coverage[page] += framing;
    for &key in run { // Pending → Bytes or ZeroExplicitly
        fragments[key] = fragments[key].stated_by(s, page);
        pin(s);
        if stmt.is_inline() { coverage[page] += length_of(key); } // payload, per byte
    }
    match stmt {
        Grow { n } => set_grow_witness(id, s, n),
        Shrink { .. } | Tombstone => set_anchor(id, s),
        - => {}
    }
}

```

Everything else a statement implies has already happened by then: the take released the old owners, and the size changes happened when the flush [made them](#). What bind adds is ownership, framing, and the anchor — the three things that need the statement to exist.

17.4.3 grow_size_to(id, n) and shrink_size_to(id, n)

```

fn grow_size_to(id: Id, n: u32) {
    let meta = &mut allocations[id];
    if n <= meta.size { return; }
    let exposed = meta.size .. n;
    meta.size = n;
    // The exposed range is owned by the anchor if there is one, else by nobody.
    let f = match meta.anchor {
        Some(a) => { pin(a); ZeroExplicitly { statement: a } }
        None => ZeroByDefault,
    };
    // Extends the last fragment if it already resolves that way; else one entry.
    insert_or_extend(id, exposed, f);
    retire_dead_grows(id); // size > n kills them
}

fn shrink_size_to(id: Id, n: u32) {
    let meta = &mut allocations[id];
    for (key, old) in fragments.drain_from((id, n)) { // everything at or past n
        if let Some(s) = old.statement() { unpin(s); }
        meta.fragment_count -= 1;
    }
}

```

```

    if let Bytes { page, .. } = old { coverage[page] -= length_of(key); }
  }
  if n < meta.size { split(id, n); } // truncate the straddler
  meta.size = n;
}

```

grow_size_to is where the anchor earns its F pins, and retire_dead_grows is the three $O(1)$ death tests from [Liveness](#).

17.4.4 set_anchor(id, new)

```

fn set_anchor(id: Id, new: StatementRef) {
  if let Some(old) = allocations[id].anchor { unpin(old); } // may die here
  allocations[id].anchor = Some(new);
  pin(new);
  retire_dead_grows(id); // now below anchor_epoch
}

```

17.5 Mutation, at flush time

What the flush calls for each id its fold touched, once per flush and with the fold's net result: each call takes what it changes, applies the size change at once, and leaves the id a record for the cut.

17.5.1 write_bytes(id, offset, len, origin)

```

fn write_bytes(id: Id, offset: AllocationOffset, len: u32, origin: Origin, dirty: &mut Dirty) {
  touch(id, dirty); // records the id; sets `last_written`
  grow_size_to(id, offset + len); // a write past the end
  take(id, offset..offset + len, Pending::bytes(origin), dirty);
}

```

No page is chosen here: pack places the bytes later — unless they are short enough to be stated [Inline](#) — and the cut states them. A write past the current end produces **two** fragments — the gap [size, offset) and the written range — which is the only way one write adds two.

17.5.2 resize(id, n), allocate(id, n), and free(id)

```

fn resize(id: Id, n: u32, dirty: &mut Dirty) {
  touch(id, dirty); // records the size before the change
  if n > size(id) { grow_size_to(id, n); } else { shrink_size_to(id, n); }
}

fn allocate(id: Id, n: u32, dirty: &mut Dirty) {
  touch(id, dirty); // records that it did not exist
  allocations.insert(id, AllocationMeta::empty());
  grow_size_to(id, n);
}

fn free(id: Id, dirty: &mut Dirty) {
  touch(id, dirty);
  shrink_size_to(id, 0); // every fragment goes, and with it `F`
  let meta = allocations.remove(id);
  release_anchor_and_witness(&meta); // the cut states the tombstone
  ids.recyclable.insert(id, RecyclableId { mentions: meta.mentions, tombstone: None });
}

```

None of them states anything: touch records, the first time the flush meets an id, the size and existence it had, and the cut reads that record back. The allocation-map entry goes away at the free; only mentions and, once the cut binds it, the tombstone reference survive, in the recyclable set.

17.5.3 What the cut states for a touched id

The fragment map says what an id's bytes are, but not how its size and existence changed, so these statements come from the id's record:

how the id changed during the flush	the cut states
it existed and was freed	Tombstone(id)
it existed, was freed, and was allocated again	no tombstone, which would contradict the new incarnation in the same epoch; the fold takes the whole new extent instead, so that Zero covers what it did not write, plus Shrink(id, size) if the new allocation is smaller than the old one
it shrank	Shrink(id, size) — always, for the reason Liveness gives
it grew, or is new	Grow(id, size) unless a statement the cut states for the id reaches size — so an allocation of size 0, which nothing reaches, always gets its Grow(id, 0), the only evidence it exists
a page this flush retires held its anchor or grow witness	replacement_anchor(id), or a fresh Grow, unless a row above already states one

At most one of these per id, so they cannot contradict one another in one epoch, and because the cut evaluates them last, “a statement the cut states reaches size” is exact.

17.5.4 drop_physically(stmt)

Called when a page rewrite decodes a statement and does not re-emit it.

```
fn drop_physically(stmt: Statement) {
    let id = stmt.id();
    let m = match allocations.get_mut(id) {
        Some(meta) => &mut meta.mentions,
        None       => &mut ids.recyclable[id].mentions, // tombstoned id
    };
    *m -= 1;
    if *m == 1 {
        if let Some(t) = ids.recyclable[id].tombstone {
            unpin(t); // releases A; may die
            ids.recyclable[id].tombstone = None; // clear NOW, not at sweep
        }
    }
}
```

Clearing the tombstone reference at the pin release rather than at the eventual sweep is what keeps the slab free of stale references: the slot is freed at the 1 → 0 transition.

17.6 Page rewrites

The header being carried forward and both table-side consolidation mechanisms are the same operation — **take a page's live content in place, and let the cut state it again** — differing only in which content and why.

17.6.1 rewrite_page(victim, dirty)

```
fn rewrite_page(victim: PageNumber, dirty: &mut Dirty) {
    let page = decode(victim);
    for stmt in page.statements {
        if keep(stmt) == Keep::AsResolved {
            // The bytes stay where they are; only the statement stating them is new.
            for (key, len) in owned_fragments(stmt) {
                take(stmt.id(), key..key + len, Pending::in_place(key), dirty);
            }
            if is_anchor(stmt) || is_grow_witness(stmt) { dirty.record(stmt.id()).replace = true; }
        }
        drop_physically(stmt);
    }
    adopt_children(page.children); // into the header if it has room, else the replacement
}
```

It writes no page of its own, and states nothing itself. What it takes joins the flush's [dirty set](#), and the cut derives the statements, merged with whatever else the flush took nearby, into whichever page the cut chooses. So a later source that takes the same fragments — an evacuation of their data page, say — simply changes what the cut will state, and cannot [state the same fragment twice](#).

Order does not matter either. A statement's owned fragments interleave with its neighbours' — with `Ref(id, 0, 100)` shadowed over `[20, 25)` by `Zero(id, 20, 5)`, the `Ref` owns `[0, 20)` and `[25, 100)` and the `Zero` the gap between — so the ranges arrive in the victim's id order but not in offset order within an id, and the dirty set, which keeps its ranges merged and sorted, absorbs that.

17.6.2 rewrite_key_range(lo, hi, dirty)

The [rotating window](#) takes a **key range**, not a page, and it is the one rewrite that decodes nothing at all.

```
fn rewrite_key_range(lo: Key, hi: Key, dirty: &mut Dirty) {
    let mut touched = SmallSet::new(); // pages it restates from
    // The fragment map is resolved truth, so it is the source.
    for (key, fragment) in fragments.range(lo..hi) {
        // `ZeroByDefault` has nothing to state, and a pending fragment is taken already.
        let Some(s) = fragment.statement() else { continue };
        touched.insert(slab.page_of(s));
        take(key.id, key.offset..end_of(key), Pending::in_place(key), dirty);
    }
    // Anchors are invisible in the fragment map, so their replacement is recorded by
    // hand — but only where it pays, since a fresh anchor costs bytes.
    for id in ids_in(lo..hi) {
        if anchor(id).is_some_and(|a| touched.contains(slab.page_of(a))) {
            dirty.record(id).replace = true;
        }
    }
}
```

No statement is dropped physically here, and that is deliberate: the old statements stay in their pages, mentions does not move, and what changes is that they lose their fragments and therefore their pins. `unpin` releases their framing from their pages' coverage at the `1 → 0` transition, which is how pages that were never touched become reusable.

`ZeroByDefault` fragments are left as they are, because they are the *absence* of a statement rather than a statement about zero — taking them would make the cut state them as `Zero`, correct on content and wrong on cost, turning free gaps into bytes.

17.6.3 owned_fragments(stmt)

The consolidator finds a statement's fragments by range-scanning the fragment map and keeping those whose owner matches:

statement kind	scan range
Ref, Inline, Zero	(id, offset) .. (id, offset + size)
Shrink(id, n), Tombstone	(id, n) .. (id, size) — the only probes it can win
Grow	none; it wins nothing

It can stop early, because pins already says how many to expect: for a content statement $F == \text{pins}$ exactly, and for a Shrink or Tombstone, $F == \text{pins} - 1$ when it is the id's anchor and $= \text{pins}$ otherwise.

This is not an implementation tax — it *is* the consolidation algorithm. A page is rewritten as resolved truth, and one cannot emit resolved truth without first determining which parts of it this page is responsible for. Its cost is predictable in advance from `fragment_count`, which lets the ranking price a candidate before committing to it.

A page rewrite decodes exactly **one** page: its victim. It never needs an id's physically present statements across the table, because it only rewrites what the victim holds — and a [key-range rewrite](#) decodes none at all, since it reads the fragment map instead.

17.6.4 replacement_anchor(id) — the one correctness obligation

A fragment-map-driven consolidator will faithfully re-emit the content a statement owns and **silently corrupt the file** if that statement was also the anchor, because being the anchor is not a fragment and is therefore invisible in the fragment map.

Concretely: with `Ref(7,0,1000,P1)@3` in another page and `Shrink(7,10)@5` in the victim, re-emitting only `Zero(7,10,40)` drops the sole Shrink, `anchor_epoch` falls back to -1, and the size becomes $\max(1000, 50, 60) = 1000$ — re-exposing stale bytes across `[10, 1000)`.

A page holding a statement with an A pin cannot be rewritten without transferring the anchor.

The replacement depends on what the anchor is doing, and the cut decides it for every id whose record asks for one:

```
/// Evaluated at the cut, after every drop this flush makes — so the anchor being
/// replaced is no longer among the id's `mentions`.
fn replacement_anchor(id: Id) -> Option<Statement> {
  match anchor_kind(id) {
    Shrink                                => Some(Shrink { id, n: size(id) }),
    Tombstone if exists(id)               => Some(Shrink { id, n: size(id) }), // re-allocated
    Tombstone if mentions(id) > 0         => Some(Tombstone { id }),           // still needed
    Tombstone                             => None,                          // nothing left to deny
  }
}
```

A Shrink there would resurrect a non-existent id, and stating nothing while some older statement still names the id would let that statement decide existence again.

A Grow carries no such obligation, since it anchors nothing — but the grow witness must still be stated again if the victim held it and it is still the sole witness of the size.

17.6.5 The header

The header is taken whole at the start of every flush and stated again at the cut, which is what makes it the address table's [write buffer](#): everything it holds is rewritten every flush at no extra page write, and the cut decides afresh which statements stay in it.

```

fn take_header(dirty: &mut Dirty) {
    for stmt in header.statements() {
        for (key, len) in owned_fragments(stmt) {
            take(stmt.id(), key..key + len, Pending::in_place(key).with_heat(clock[key]), dirty);
        }
        if is_anchor(stmt) || is_grow_witness(stmt) { dirty.record(stmt.id()).replace = true; }
        drop_physically(stmt); // the new header replaces it in the governing world
    }
}

```

It is `rewrite_page` for the one page every flush retires, with one addition: the [eviction clock](#)’s entries ride along as the pending fragments’ heat, so that the cut can keep the hottest statements in the header and send the rest to leaves.

Because the header is rewritten unconditionally, a Shrink or Grow carried in it becomes a *new* statement record at the new epoch each flush, so the anchor re-points and the A pin moves every flush. That is routine rather than churn — header pages are exempt from coverage-driven victim selection — but it is why the anchor must be a movable pin rather than a flag baked into the statement.

17.6.6 Consolidating a data page

Unlike an address-table page, a data page is opaque bytes with no ids in it, so decoding it tells nothing about who references it. This is an **unresolved gap**: the design specifies data-page victim selection but not how a flush gets from a victim page number to the Ref statements it must re-point. See [Consolidation](#) for the candidate answers.

17.7 Worked trace

One allocation through its life, to make the rules concrete. Pages: H is the header, L1/L2 leaves, P1/PB data pages.

flush	operation	resulting state
3	allocate id 7, write 1000 bytes	Ref(7,0,1000,P1)@3, evicted to L1. Fragment (7,0) → P1+0. size 1000, anchor none, mentions 1
5	truncate to 10	Shrink(7,10)@5 in L2. Ref@3’s fragment narrows to [0,10); the Shrink owns nothing yet ([10,10) is empty) and holds A alone. size 10, coverage[P1] −990
9	grow to 60, write [50,60)	Ref(7,50,10,PB)@9 in H; the grow emits nothing, since that Ref reaches 60. [10,50) comes into existence and resolves <i>through</i> Shrink@5, which therefore gains a fragment pin: pins 1 → 2 (A + F). size 60, fragment_count 3
10	resize to 55	The header carries Ref(7,50,10,PB) forward, but at epoch 10 its extent would exceed the new size, so resolved truth narrows it to Ref(7,50,5,PB)@10. The shrink always emits, so Shrink(7,55)@10 is physically present and becomes the anchor; Shrink@5 loses A, keeps F over [10,50), drops to pins 1. coverage[PB] −5

Two things this trace is chosen to show.

Shrink(7,55) is semantically unnecessary and emitted anyway. Without it, anchor_epoch stays 5 and the size is $\max(10, 50+5) = 55$ — the target, reached with no new statement, because the narrowing forced on the tail created a statement ending exactly at the new size. Semantic necessity and physical presence are different questions, and a state table must answer the second.

The superfluous statement then has a short life: it holds A alone and dies at the next shrink, which takes A from it.

If instead flush 10 had resized to **30**, $\text{Ref}(7, 50, 10, \text{PB})@9$ would fall entirely outside the new size and be dropped rather than narrowed, leaving nothing above epoch 5 and a size of $\max(10) = 10$. There the $\text{Shrink}(7, 30)$ is genuinely required — and the two cases are indistinguishable from inside the flush without the bit that [Liveness](#) declines to maintain.

17.7.1 Re-allocation

A separate sequence, since id 7 above is never re-allocated. $\text{Ref}(12, 0, 100, \text{P1})@3$, then $\text{Tombstone}(12)@5$, then id 12 recycled at flush 7 to 100 bytes with only $[0, 10)$ written — as $\text{Ref}(12, 0, 10, \text{PX})@7$ **and** $\text{Grow}(12, 100)@7$, the grow being required since the flush's own output reaches 10, not 100 — then a truncation to 50 at flush 9.

statement	after flush 7	after flush 9
$\text{Ref}(12, 0, 100, \text{P1})@3$	pins 0 — every probe it matches is won by the tombstone or by $\text{Ref}@7$	pins 0
$\text{Tombstone}(12)@5$	pins 2 = A + F: still the newest Shrink-or-Tombstone, and it wins $[10, 100)$	loses A to $\text{Shrink}@9$; keeps F over $[10, 50) \rightarrow$ pins 1
$\text{Ref}(12, 0, 10, \text{PX})@7$	pins 1	pins 1
$\text{Grow}(12, 100)@7$	pins 1 = grow witness	dead
$\text{Shrink}(12, 50)@9$	—	pins 1 = A

The tombstone's F pin is doing real work: drop it and $[10, 50)$ would resolve through $\text{Ref}(12, 0, 100, \text{P1})@3$, serving the *first* incarnation's bytes as the second incarnation's content.

$\text{Grow}(12, 100)@7$ dies by the **third** of its death tests, which nothing else exercises: $\text{size} > n$ is false ($50 < 100$) and $n \leq \text{anchor}.n$ is false ($100 > 50$); what kills it is $\text{Shrink}(12, 50)@9$ anchoring above its epoch.

Chapter 18

Liveness accounting

What makes a statement worth keeping, what makes a page worth reclaiming, and how both are decided in $O(1)$.

Nothing here is persisted. All of it is rebuilt during the bulk read at open and maintained incrementally afterwards, and all of it is **policy**: an over-count only delays cleaning, an under-count only cleans a page earlier than ideal, and neither can make a resolved read wrong.

18.1 Coverage

A page's **coverage** is the number of bytes in it that current state still relies on. It is what makes a page a consolidation victim, so it has to fall as the page's contents become useless.

One rule covers both page kinds, and it is the rule data pages already follow.

Content bytes are charged to the page that physically holds them. Every Bytes fragment names a page and an offset, so when a fragment is destroyed, decrement that page's coverage by the fragment's length. A Ref's fragments point into a Data page; an Inline's fragments point into the AddressTable page carrying the payload; a zero fragment points nowhere and costs nothing. Splitting a fragment changes nothing, since both halves still cover the same bytes.

An inline payload is therefore just content that happens to live in a table page, and it gets per-byte accounting for free — no per-statement payload counters, no special case, the same line of code.

Why per-byte accounting for inline payloads is not optional. For a Ref, partial shadowing strands only ~10 bytes of framing in the address-table page, while the content bytes it no longer reaches are tracked exactly in the data page's counter, which duly falls. For an Inline there is no data page: the payload *is* the content, so charging it all-or-nothing would let shadowing half of a 200-byte inline strand 100 bytes that no counter in the system ever notices, and a page holding sixty mostly-shadowed inlines would report itself nearly full and never be cleaned.

Framing bytes are charged to the statement's own page, and released in one step when the statement loses its last reason to live. The two charges cover disjoint byte ranges of the encoding, so they cannot double-count.

During a flush, releases come first and charges wait for locations. A range the flush **takes** is released from its old owners at once, so victim selection within the same flush already sees what the flush will leave. Its bytes are charged to a page once they have one — at once when they stay where they are, when pack places them when they move, and when the cut picks a table page for an inline payload — and the new statement's framing when the cut binds it.

18.2 Pins

A statement is live exactly while $\text{pins} > 0$, and pins is a **single counter over heterogeneous holders**.

pin	held by	taken when	released when
F	any statement	a fragment resolves <i>through</i> it	that fragment is destroyed or re-owned
A	the anchor — the newest Shrink or Tombstone — and a Grow with <code>n == size</code>	the anchor: on becoming newest; a Grow: while it may be the sole witness of the size	the anchor: a newer Shrink or Tombstone supersedes it, or — a tombstone anchor only — mentions falls to 1; a Grow: as soon as <code>size > n</code> , or <code>n <= anchor.n</code> , or it falls below <code>anchor_epoch</code>

So a content statement (Ref, Inline, Zero) holds only F, a Grow holds only A, and a Shrink or Tombstone can hold both.

Why one counter and not two. A Shrink holds A and F simultaneously — with `Ref(id, 0, 1000)@3`, `Shrink(id, 10)@5` and `Ref(id, 50, 10)@9` the size is 60 and `[10, 50)` resolves *through* the Shrink — and no consumer ever asks which kind a pin is. Two fields would encode a distinction that is never read, while inviting the bug where a predicate checks one and forgets the other.

Why “pin”. This is a refcount whose holders are of different kinds, so liveness would read as a boolean and refcount would say nothing about what is doing the referring. “Pin”, in the buffer-manager sense of *something prevents this from being reclaimed*, is exactly the relationship.

Pins can rise after creation. Growing an allocation can bring a range into existence that an existing Shrink wins.

A fragment the flush has taken holds no pin, since the statement that will own it does not exist until the cut [binds it](#); the old owner’s pin was released by the take.

18.3 The droppability rule

A Shrink or Tombstone may be dropped iff it is not the anchor and owns no fragment. A content statement may be dropped iff it owns no fragment. A Grow may be dropped iff it is not the size’s sole witness.

Equivalently, in all cases: `pins == 0`.

Why a Shrink needs no denial pin. A non-anchor `Shrink(id, n)` cannot affect the size, since its epoch is below `anchor_epoch` and the size’s maximum ranges only over Grow bounds and content extents above the anchor. It cannot affect content beyond the probes its F pins already count, since it wins exactly the probes in `[n, size)` that nothing newer covers. And a later grow cannot expose anything either, because the anchor always has `n <= size` and so already denies every probe from size upward.

Why a content statement needs none. A content statement’s denial is exactly co-extensive with what it defines. If a Ref covering `[a, b)` has lost every fragment, then every byte of `[a, b)` is covered by something strictly newer, and that newer thing also outranks everything the Ref was denying. The size is safe too: covering `[a, b)` requires some newer statement to reach offset `b`, so the size cannot shrink when the Ref goes.

Why a tombstone needs none either. The same argument runs one step further: every probe in `[0, size)` is won by something newer, or the tombstone would own that probe; every probe `>= size` is denied by the anchor; and existence is decided by a statement newer than the tombstone, since one exists by assumption. Nothing it was denying can resurface — *even though the statements it denies may still be physically present*.

18.4 Tombstones

A Tombstone is a `Shrink(id, 0)` that also denies existence. It matches every probe, exactly as the [resolution rules](#) say, and it takes the same pins a Shrink does.

So a tombstone *can* own fragments — the ranges a re-allocated incarnation leaves uncovered, which it wins because it outranks everything older and nothing newer covers them. While the id does not exist its size is 0, so no probes exist and it owns nothing; it is then the anchor, and that alone keeps it alive.

Why a matcher and not an epoch floor. An earlier design inverted this: discard every statement at or below `tombstone_epoch`, so that a tombstone could never own a fragment and needed a denial pin instead. The two framings resolve identically, but the matcher framing is what lets one droppability rule cover Shrink and Tombstone alike, and it costs no fragment-map entry — only a pin — since an uncovered range has an entry either way. A resolver may still walk each id's statements newest-first and stop at the first tombstone; that is an implementation shortcut, not a separate model.

18.4.1 The last tombstone

The one thing the two pins do not decide on their own.

A tombstone that is its id's newest statement is the anchor, so A keeps it alive — correctly, because dropping it while any older statement mentioning the id is still *physically present* would let that statement decide existence again and resurrect the allocation. The count must be over physically present statements, not resolution-live ones: a statement with zero pins that has not yet been swept out of its page resurrects just as well.

So a tombstone anchor **releases A when mentions falls to 1**: it is then the id's last statement, the id reads as non-existent without it, and `pins == 0` is once again an exact droppability test.

This is the only reader of mentions. A re-allocated id's tombstone escapes it entirely, being kept alive by A and F on its own terms.

18.4.2 The lemma

Nothing older than the newest physically present tombstone is live.

Such a statement cannot hold F, since the tombstone matches every probe and outranks it at all of them; it cannot be the anchor, which sits at or above the tombstone's epoch; and it cannot be the grow witness, since a Grow below `anchor_epoch` is dead by its third death test.

Two consequences follow with no bookkeeping at all.

At most one tombstone per id is ever live, so the allocation map needs no tombstone pointer beside anchor and tombstone chains need no special handling, even though recycling does not wait for the old tombstone to die — an older tombstone is neither the anchor nor an owner of fragments, so the rule above retires it on sight.

A tombstone is the bound-0 element of the stray chain below, and a strictly decreasing chain of non-negative bounds holds at most one 0. That is the whole of the difference between Shrink and Tombstone here: the chain is unbounded for Shrink because `Shrink(id, m)` still wins the probes in `[m, n)` that a newer `Shrink(id, n)` does not match, and there is no bound below 0 for a tombstone to leave uncovered.

18.4.3 The cascade

Statements shadowed by a tombstone contribute no content bytes, so their pages sink toward zero live fraction and become victims; consolidating those pages drops the statements, which decrements mentions; when it reaches 1 the tombstone goes dead, sinking *its* page's live fraction, which eventually recycles it too. Nothing has to reason about epochs to make this happen — only about counters.

18.5 Grow is locally decidable, and Shrink is not

A Grow matches no probe, so it can never own a fragment and never becomes a fragment-pinned stray. It is dead as soon as `size > n`, since some other term then achieves the maximum; as soon as `n ≤ anchor.n`, since the anchor's own term does; and as soon as it falls below `anchor_epoch`, which tombstoning also causes. All three tests are `O(1)`.

Because a Grow's bound strictly exceeded the size at emission, and any later decrease moves the anchor above it, the bounds of the Grow statements above the anchor are distinct and **at most one Grow per allocation can be alive** — the one with $n == \text{size}$.

Dropping a Grow also carries no anchor hazard, because it does not anchor the epoch and so can only lower the maximum, never readmit older extents.

18.5.1 The anchor pin is not substitutable

Three separate arguments say the A pin cannot be replaced by a test on live state.

It may be the sole witness of the size. With $\text{Grow}(\text{id}, 100)@5$ and $\text{Ref}(\text{id}, 0, 10)@9$ and nothing older, $[10, 100)$ resolves to zero *by default*, so the statement owns no fragment and denies nothing — yet dropping it would shrink the id from 100 to 10. More starkly, an allocation created and never written has $\text{Grow}(\text{id}, n)$ as the *only evidence it exists*.

Dropping an anchor readmits older extents. The near-miss test “the anchor is redundant whenever $\text{size} > n$ ” is **false**, because dropping an anchor also moves `anchor_epoch`. With $\text{Ref}(\text{id}, 0, 1000)@3$, $\text{Shrink}(\text{id}, 10)@5$ and $\text{Ref}(\text{id}, 50, 10)@9$, the size is 60 and $\text{size} > n$; dropping the anchor exposes $\text{Ref}@3$'s extent of 1000 and yields 1000. This is the hazard a consolidator must respect: **a page holding a statement with an A pin cannot be rewritten without transferring the anchor**.

A dead predecessor makes an anchor load-bearing. Even with every statement above the anchor dead, dropping the anchor re-anchors at the newest physically present Shrink beneath it — possibly one that is itself dead and superfluous — and readmits *its* bound.

18.5.2 A Shrink can raise the size, not only lower it

$\text{Shrink}(\text{id}, n)$ asserts $\text{size} \geq n$ — as anchor it contributes n to the maximum — as well as denying content at or past n . Relative to its absence it lowers the size by excluding older extents and raises it through its own bound, and which effect dominates depends on what else is physically present.

Consolidation can remove the older extents it was excluding while the statement itself stays, after which only the raising effect remains: with $\text{Shrink}(\text{id}, 30)@10$ as anchor and $\text{Ref}(\text{id}, 0, 10, P)@11$ as the only other content statement, the size is 30 with the Shrink and 10 without it. This is not a defect — the statement then behaves exactly as a $\text{Grow}(\text{id}, 30)$ would — but a consolidator cannot rewrite it as one, since it cannot verify that nothing older still reaches past 30. **The name records the operation that emitted the statement, not its steady-state role.**

18.6 Emission: when a resize must write a statement

The rule splits by direction, and only the shrink half is undecidable. Both halves are applied at the cut, from the id's record, once everything the flush states is known; see [what the cut states for a touched id](#).

A grow emits $\text{Grow}(\text{id}, S_{\text{new}})$ iff nothing this flush states reaches offset S_{new} . Exact: everything *not* stated by this flush is bounded by the old size, so the flush need only inspect its own output — which at the cut is complete. The added extent needs no protection either, since the anchor already denies the exposed territory. So the common append pattern — growing *and* writing at the new end — emits nothing at all, and a grow without writing costs one three-byte statement that plants no latent content claim.

A shrink always emits $\text{Shrink}(\text{id}, S_{\text{new}})$. In principle it is needed exactly when the target is not already implied, i.e. when it differs from the maximum of the anchor's n and the extents of every **physically present** content statement above `anchor_epoch`. Run directly, that check is not implementable: no structure reaches an id's physically present statements, for the reasons given in [In-memory state](#).

It does not follow that the decision is unknowable. Everything that can lurk is leaf-resident, since the header is rewritten as resolved truth every flush and so holds exactly this flush's own output and never a dead statement. One bit per allocation therefore makes the test exact — set it whenever a statement naming the id is written into a non-header page above the id's `anchor_epoch`, clear it whenever a Shrink installs a new anchor; with the bit clear, the maximum that survives the flush is $\max(\text{anchor}.n, \text{this flush's own size claims})$, and the statement may be omitted exactly when that equals S_{new} .

The design declines to pay for it: the bit costs a lookup and a store on the eviction path for every evicted statement, and a fifth piece of per-allocation state that silently corrupts the file if maintained wrongly, to save three bytes on the subset of shrinks whose id has not been evicted since its last shrink. Always emitting is sound, since an explicit `Shrink(id, T)` fixes the size at `T` whatever extents lurk in leaves, and a superfluous one retires at the very next shrink.

So the honest statement of the rule is **not** that an implementation cannot know, but that the one bit which would decide it costs more to maintain than the statements it saves.

18.7 The residual

Redundant Shrink statements can survive, and the over-count is not bounded at one per allocation.

A superfluously emitted Shrink becomes the anchor, holds `A` alone, and dies at the next shrink — unless an intervening grow hands it fragments over `[n, size)`, in which case it was not superfluous for long. A non-anchor Shrink lingers whenever it owns a fragment, and it can own one *while being redundant*: when the range it wins would resolve to zero anyway, because nothing older matches it. In fragment-map terms the redundancy is sharp — dropping such a statement would turn its `ZeroExplicitly` fragment into a `ZeroByDefault` one covering the same range with the same resolved content.

What bounds it. Order an id's physically present Shrink statements newest-first: one can hold an `F` pin only if its bound is strictly below the minimum bound among all newer ones, so the live set is **contained in the running-minimum chain**. Any shrink to bound `b` therefore permanently retires every Shrink with bound $\geq b$, and a later grow never revives one.

Each extra link costs an *alternating* shrink and grow, which is what the Grow/Shrink split buys: a run of plain grows plants no content claims at all. Under a single `Size` statement, `Size(id,10)`, `Size(id,20)` and `Size(id,30)` followed by `Ref(id,50,10)` left three fragment-owning strays over `[10,20)`, `[20,30)` and `[30,50)`; with Grows the same history leaves **none**.

Whether a link is a stray or load-bearing depends on what is physically present beneath it, which nothing can see. The same Shrink is load-bearing while the extent it truncated is still in a leaf, and a stray once that leaf has been consolidated. This is the same undecidable predicate as the shrink-emission test, seen from the consolidation side — and exactly the predicate a Grow never needs, since it denies nothing.

It is not self-reinforcing at scale: three bytes per stray does not meaningfully shift a page's live fraction.

18.8 A possible extension, not adopted

An id's live statements are exactly the owners of its fragments together with its anchor and grow witness, since `F` and `A` are the only pins there are. Walking the id's fragments and collecting the distinct owners therefore yields a live count, and comparing it with mentions says whether any *dead* statement naming the id is still physically present.

For a tombstone that test is **exact**, by the lemma above: if no dead mention remains then nothing older than the tombstone remains at all, so dropping it could change neither existence nor the size, and it could retire while still the anchor — which the mentions $== 1$ rule never permits for a re-allocated id, since the new incarnation's own statements hold mentions above 1.

For a Shrink the same test would be **unsound**, and instructively so: statements older than a `Shrink(id, n)` stay *alive* below `n` and carry their extents with them, so an anchor could pass the test while an older `Ref` is live with a far greater extent, and dropping the anchor would readmit it.

The cost is $O(\text{fragment_count})$ per check, which is why it is left out for now. Two cheaper triggers are worth revisiting if tombstones turn out to linger: running the count **at open**, which decodes every physically present statement anyway and so can retire every tombstone that is its id's only statement directly; or when the recyclable-id pool grows past a threshold. The two do not cover the same population, and the load-time one covers strictly more: a tombstone frozen by re-allocation belongs to an id that is *not* in the pool, so no pool-size threshold notices it and mentions $== 1$ can never fire for it again.

18.9 Confirmed sound

Three properties that the adversarial walk-through of this model checked explicitly, and that any reimplementation should preserve:

- **The pin graph is acyclic**, trivially so now that both kinds point the same way: F and A both point from the resolved view into statements, and no statement pins another. The denial pin of the earlier model was the only statement-to-statement edge it ever had.
- **A statement holds at most two pin kinds**, and there is no third kind left in the model.
- **mentions is a net count**, so a flush that replaces a statement leaves it unchanged and the order in which a flush applies its writes and drops does not matter.

18.10 Ranking

Cleaning should be ranked by **retirements enabled** as well as bytes reclaimed.

Victim selection ranked purely by live fraction cannot see the benefit of cleaning a page whose dead statements are holding some tombstone's mentions above 1, so a page at 99.8 % live fraction is never chosen even though cleaning it would let a tombstone elsewhere retire. A benefit term counting "dead statements whose removal would let a tombstone retire" makes that visible.

The case is confined to the tombstone of an id that was freed and never re-allocated.

Chapter 19

Id recycling

Which recyclable id to hand out next, and why the order is worth choosing deliberately.

The [specification](#) fixes only that an id becomes reusable as soon as a tombstone for it is committed, and leaves the order free. The order still matters, for a reason that is not the obvious one.

19.1 The policy

Recycle lowest-id-first, taking ids whose tombstone is already dead first, and never hold a recyclable id back in favour of a fresh one.

What decides this is **delta encoding**, not tombstones. Statements are sorted by (id, offset) and id_delta is a varint, so what matters is how tightly the *live* id set is packed: id_delta costs a byte more for every factor of 128 by which the set is spread out, and it costs that once per id per page — which in the many-small-allocations regime this design targets is once per statement. Against that, a stale tombstone is two bytes once per id.

Lowest-free-first keeps the live set packed, and needs only a min-heap over the recyclable set or a bitmap scanned for its lowest set bit.

19.2 The refinements

Prefer ids whose tombstone is already dead. That is exactly the mentions == 1 case, and it is free to identify because the recyclable-set entry's tombstone reference is then already absent. Recycling one of those costs nothing at all: no anchor is carried into the new incarnation, nothing older is left to deny, and no tombstone is frozen.

mentions is a weak signal pointing the opposite way from the intuition. Recycling an id freezes the mentions == 1 path permanently, since the new incarnation's own statements hold the count above 1 from then on. So recycling an id at mentions == 2 discards progress that was one page-clean from completing, while recycling one at mentions == 50 discards almost nothing — an argument for taking *high*-mentions ids first. At two bytes apiece this is a tiebreak at most, and it should lose to density wherever the two disagree.

Do not hand out fresh ids while recyclable ones wait. It consumes the 32-bit id space at the rate of allocations *ever made* rather than concurrently live, and it sparsifies the live set — paying per statement to save per id. The [tombstone sweep](#) is the better answer to the same worry, and reaches tombstones this one cannot.

19.3 What an earlier draft got wrong

Gating recycling on mentions == 0 — waiting for every statement naming the id to be physically dropped — coupled id availability to the cleaning of unrelated pages. One dead seven-byte statement in a page

that is never chosen as a victim could hold an id out of circulation for the life of the file, and a workload that churns allocations would leak ids at the rate its cold pages resist cleaning.

Recycling now waits only on a committed tombstone, so the stake is two bytes of tombstone framing rather than an id.

Chapter 20

Write-phase state

What an implementation tracks between flushes, and why it is derived from the journal rather than kept alongside it.

20.1 The log is the sole authority

It is a totally ordered sequence of every mutation. Everything else is derived from it and may be discarded and rebuilt.

This is the one structural rule of the write path, and it is worth stating as a rule because the obvious alternative — keeping a *state snapshot* beside an *operation log* — fails in a way that is easy to miss.

A snapshot keyed by id is unordered, last-writer-wins, and self-annihilating: allocate-then-free collapses to nothing. A log is ordered and append-only. If both exist and no order relates them, then any operation that invalidates an earlier log entry must reach into the log and repair it by hand — and the failures that produces are not all annihilation failures:

- **alloc → write → free cannot replay.** The allocate and free annihilate in the snapshot, so the id is never claimed, while the buffered write survives in the log and replays against an id that was never created.
- **A write followed by a shrinking resize corrupts a neighbour,** if replay does not bounds-check: a write buffered while the allocation was large replays at an offset now outside it.
- **And the case with no annihilation at all:** a value that allocates a child and writes the child's id into its own header emits a Write the log keeps and, for the child, a Resize the log does not. Recovery replays the log, finds a live pointer in the parent, and that pointer names an allocation nothing ever created.

The last one is the reason [the journal records existence and size](#) as well as content, rather than leaving geometry to a snapshot beside it.

20.2 Two derived structures

Both are derived, and it matters that they are.

- **Write-phase geometry** — per-id size and existence, so that bounds checks and size queries during the write phase are $O(1)$. Discarded at flush.
- **The fold's output** — geometry *and* content, built at flush from the log alone, consumed by the placement phase, then dropped. See [The flush](#).

The fold recomputes the geometry the write phase already had, and **that redundancy is the point**: comparing them on every flush is a consistency check that all three failures above would have tripped. It is asymptotically free, since the fold is already linear in log length and the comparison is linear in touched ids.

20.3 One id, one lifetime

An id returns to the recyclable pool only at flush, after the frees are applied.

This is a precondition, not an optimization. The log and both derived structures are keyed by id, so if an id could be recycled mid-transaction then

```
Resize(id, 8), Write(id, ...), Free(id), Resize(id, 16), Write(id, ...)
```

would place two allocation lifetimes under one key. The fold cannot emit two claims for one key, and the first write would be attributed to the second allocation.

It also makes the Freed → New transition unreachable rather than merely discouraged, which is what keeps the state machine below finite.

Note that this is a *within-transaction* rule and does not conflict with [recycling an id as soon as its tombstone is committed](#): the tombstone is committed by the flush, which is exactly when the id returns to the pool.

20.4 The geometry state machine

Five states, of which two are “absent” and must be distinguished:

state	tracked?	live in the file?	flush action
Absent/live	no	yes — claimed earlier, untouched this transaction	nothing
Absent/dead	no	no — never minted, or freed and flushed	nothing
New(S)	yes	no	allocate
Resized(S)	yes	yes	resize
Freed	yes	maybe	free if live , then recycle the id

New and Resized are distinct because they produce different statements at flush, not because they carry different data.

20.4.1 Transitions

operation	from	to	note
alloc	Absent/dead	New(S)	the only source state
write	any live state	unchanged	content only; this map is geometry
resize	New(S)	New(S')	
	Absent/live	Resized(S')	
	Resized(S)	Resized(S')	
	Freed, Absent/dead	—	error
splice	as resize	as resize	size change plus content operations
free	Absent/live, Resized(S)	Freed	
	New(S)	Freed	not Absent
flush	New, Resized	Absent/live	map cleared
	Freed	Absent/dead	ids recycled here, and only here

write earns a row precisely because it is a no-op on the map. If write ever needs to touch geometry, something has gone wrong.

Freeing a New entry goes to Freed, not Absent. Going to Absent loses the information that the id must be recycled, which is an id leak. Making the flush’s free lookup-guarded lets one state serve both cases without a fourth variant.

Most error cells are unreachable if the API consumes an owned handle on free. The escape hatch is any operation that can mint a second owner, so those cells *are* reachable through a duplicate handle and do need runtime checks. Query order is therefore **this map first, the committed state second**; never trust a typestate alone.

20.4.2 Repopulation after a flush

Because an explicit flush can occur while a transaction is still open, with its operations buffered past the committed cursor, this map no longer describes the same point in the operation sequence as the flush does. It must be **repopulated** from the remaining operations after the flush rather than simply cleared. See [Transactions and batches](#) for exactly which transitions this affects and why the ordering is worth keeping explicit.

Chapter 21

The flush

How the journal becomes pages: what to write, where to put it, and in what order.

The [protocol that commits a flush](#) is normative; everything about *what* a flush chooses to write is not, and is described here.

This is a small optimizing compiler, and it should be built like one: **oracle first, optimizer second**.

21.1 Phase A — the fold

Per id, linear, no graph. For each id the journal touches, walk its record subsequence maintaining whether it exists, its current size, and a **piece table** over its bytes.

```
enum Source {
    Literal(ArenaPos),    // bytes buffered in the journal's byte arena
    Storage(Id, Offset),  // must be read from the file
    Zero,                 // resolves to zero; occupies no bytes anywhere
}
```

A piece table is a sorted map from **segment start offset** to Source — one entry per segment boundary, never per byte. A byte's origin is found by taking the greatest key $\leq$ the offset and advancing that source by the difference. Source is offset-relative precisely so that advance is meaningful.

The distinction that drives everything:

- **Fresh** ids — absent when the fold first meets them, or released by a Free earlier in the journal — start $\{0 \rightarrow \text{Zero}\}$ at size zero. Their content is *fully symbolic*; nothing about them ever needs to be read.
- **Persistent** ids — live from an earlier flush — start $\{0 \rightarrow \text{Storage}(\text{self}, 0)\}$.

Every table therefore begins with exactly **one** entry, which is what makes the compact representation below natural rather than bolted on.

21.1.1 Per-record rules

record	effect on the table
Write(off, bytes)	overwrite [off, off+len) with Literal
Resize(s')	clip to s', or extend with Zero
Splice(off, old_len, new)	overwrite, then shift the suffix segments
Copy(src, ...)	overwrite the destination range with pieces taken from <i>src's current table</i>
Move(src, ...)	as Copy, then overwrite what the destination left of the source range with Zero

record	effect on the table
Free	mark released; the next record that brings the id back starts a fresh table

Every record but Free [first brings its id into existence](#) and extends the table with Zero to the end of the range it writes, so a record on an absent id starts a fresh table. Overwrite is the only non-trivial primitive: split at both boundaries, drop the entries strictly inside, insert the new one.

Coalesce on insert. A loop of sequential writes produces adjacent literals contiguous in the arena, and without merging you get one entry per write where one per run would do. This is not a micro-optimization; it is what keeps the common case at one or two segments.

21.1.2 What falls out unasked

- **A splice on a fresh allocation costs zero I/O.** It is a table edit; the shift never touches bytes. Since strings and vectors splice in loops, this is probably the largest single win available.
- **A transient allocation — fresh and freed in the same journal — never touches storage.** Recycle its id and emit nothing.
- **An identity piece emits nothing.** `Storage(self, k)` with `k` equal to the segment start means the bytes are already where they belong. Without this check, every untouched region of every persistent allocation would emit a self-copy.

21.1.3 Representing a piece table cheaply

Most touched allocations have exactly one segment, so a tree each is wasteful:

```
enum Content {
    Uniform(Source),    // one segment over [0, size) — no allocation at all
    Spilled,            // entries live in the flush-wide map
}
```

Uniform covers freshly-allocated-and-fully-written, untouched-persistent, and grown-but-unwritten. On the second segment, spill into **one** flush-wide map keyed by `(Id, Offset)`. Zero tree allocations in the common case, exactly one in total otherwise.

21.2 Phase B — placement

The fold says *what* each allocation’s content is; placement says *where* the new bytes go.

21.2.1 Cutting content across pages

A flush writes whole pages, so a dirty range must be cut into pieces that each fit in one page. Two rules govern the cutting:

- **The writer prefers maximal pieces.** A long dirty range becomes `MAX_PAGE_CONTENT`-sized pieces in whole pages plus a remainder in a shared page; a small allocation is written as a single piece in a shared page. This is what keeps address-table statements compact.
- **Whoever rewrites bytes may re-cut them.** Piece boundaries carry no meaning beyond “these bytes are stored contiguously here”, so a flush re-cuts the ranges it rewrites at will, and consolidation may split a piece it relocates — or merge adjacent pieces it relocates together — so that its target pages come out full; [packing](#) says when a split is worth its statement.

Why free re-cutting rather than a fixed head/middle/tail scheme. The problem it solves is **worst-case consolidation**. With rigid piece sizes, an application that only ever allocates `MAX_PAGE_CONTENT / 2 + 1` bytes pins every data page at ~50 % fill, and no amount of relocation can fix it, because the pieces cannot be made to fit. One flexible cut per allocation only softens this — sizes just over half a page still strand ~25 % — and fixed-size middle pieces freeze a large allocation’s page alignment at creation, so consolidation could never re-pack it without rewriting all of it.

Free re-cutting dissolves both problems at once: consolidation *may* pack pages the way stock is cut rather than the way bins are packed — fill the page, cut whatever piece crosses the boundary, continue with its remainder in the next page — so a high fill is reachable for *any* population of live bytes, and there is no alignment left to preserve because there is no grid. The price of each such cut is bounded and small: one extra statement, roughly 10 address-table bytes, plus a fragment-map entry.

That is a capability, not the routine policy, and the distinction matters because the price is permanent while the slack it avoids is not. Free re-cutting is what removes the worst case above, where no packing exists at all; in ordinary flushes the packer prefers [leaving slack to paying for a split](#), and cuts only where it is forced — a chunk longer than a page — or where a page would otherwise close more than θ empty.

A flush writes every piece it produces **whole**, even when only one byte of the range changed. This is not new cost in disguise: the operating system would have rewritten the surrounding page in place anyway. What is new is that the old bytes remain behind as garbage until consolidation reclaims them.

21.2.2 Page contiguity buys nothing

Adjacent pages need exactly as many statements as scattered ones, because a Ref’s payload is [bounded by one page’s content](#): no Ref spans a page boundary, so a bulk write of many pages is one statement per page wherever those pages are. Placement therefore does not seek runs of pages at all; it takes the lowest reusable page, which is what keeps [outer fragmentation](#) from forming.

What *does* fragment is the description of an allocation filled incrementally across many flushes, whose pieces land wherever each flush put them. There the honest statement is that the description fragments in proportion to the number of flushes that touched the allocation, and that repairing it is **consolidation’s** job, not the flush’s.

Uninitialized ranges help the sparse case for free: an allocation created large but written sparsely stores locations only for the ranges actually written, and occupies no data pages for the rest.

21.2.3 Where the pages come from

The reusable pool is an ordered set, a list, and a watermark, because the [two-generation quarantine](#) is a rotation rather than a computation:

```
struct FreePages {
    ready:    BTreeSet<PageNumber>,    // reusable: neither live nor fallback
    retiring: Vec<PageNumber>,          // dropped by the last commit; not yet writable
    next_new: PageNumber,               // one past the last page of the file
}

// At each commit, in this order:
free.ready.extend(free.retiring.drain(..));
free.retiring = pages_this_commit_stopped_referencing();
```

allocate_reusable_page takes the **lowest** page in ready, and extends the file only when ready is empty; journal pages come from the same pool. Lowest-first is what lets pages at the end of the file stay dead once they die, so that [truncation](#) can return them, and it costs one pop_first — a bitmap with a cursor on its lowest set bit does as well. No search beyond that, and no interval tree: [contiguity buys nothing](#), so the pool never has to answer “find me k adjacent pages”.

21.2.4 Cutting and packing, concretely

The objective is to fill pages without describing their contents in more statements than necessary — every cut is one more statement — and the two pull against each other only where chunks fit pages badly.

```
fn place(dirty: Vec<Range>, rewrites: Vec<Chunk>, elastic: &mut Elastic) -> Vec<Page> {
    // 1. Whole pages first: a long range becomes floor(len / C) full pages, each
    //    described by exactly one statement, plus a remainder below C. This is the
    //    one cut that is forced rather than chosen.
    let mut remainders = Vec::new();
```

```

for r in dirty {
    let (whole, rest) = r.split_at_multiple_of(MAX_PAGE_CONTENT);
    for piece in whole { write_whole_page(piece); }
    remainders.push(rest);
}

// 2. Pack the remainders — with the description defragmentation rewrites this
//    flush pays for, which are never longer than a page — in (id, offset) order
//    with a bounded look-ahead, pulling in consolidation's victims wherever a
//    page has room its own content cannot use, and cutting only to keep a page
//    from closing more than 0 empty.
pack(Lookahead::new(merge_by_key(remainders, rewrites), L), elastic)
}

```

`pack` is described with consolidation, because what it has to know is consolidation's side of the interface: the flush's chunks must be written and a victim's need not be, which is what lets a victim fill the room a cut would otherwise have filled. It guarantees that every page a flush writes, except its last, is at least 1 – 0 full, and when it must cut, it cuts the flush's own chunk rather than a victim's — the flush's content being the likelier to be overwritten soon, taking the extra statement with it.

21.2.5 The Inline threshold

The `format` fixes the ceiling at 251 bytes; the policy threshold belongs far below it.

Below roughly the encoded size of a Ref (~10 bytes), inlining is strictly better — the pointer would be as large as the data. A starting policy: inline everything up to ~64 bytes, plus anything whose eviction would leave a data page holding only scraps; then measure.

Between there and the ceiling it is a trade: inline payloads ride the header for free while hot but inflate the address table when cold, whereas a Ref plus its data bytes is written once but occupies a data-page slot and keeps that page partially live. Note the accounting asymmetry that argues for keeping the threshold low: an inline's payload is charged to the address-table page that holds it and is only reclaimed by rewriting that page, whereas a Ref's content is charged to a data page that consolidation can clean independently of the statement describing it.

This is what makes small allocations first-class. Below the threshold, an allocation lives its entire life — creation, every update, deletion — as a few bytes of statements inside pages the flush was writing anyway, with no page-granularity amplification anywhere. That is the cost profile a dynamically typed guest language, allocating by the thousands, needs.

21.3 Phase C — ordering

Vertices are the **emitted actions**, not the journal's records — the fold changed the set:

```

Claim(id, size)          Release(id)
Reshape(id, size)        Transfer(src, src_off → dst, dst_off, len)
Write(id, off, bytes)

```

Reshape stays one vertex. It is tempting to decompose it into claim/transfer/release — that decomposition is a good *explanation* of why conversion and multi-flush resize are the same latent hole — but it is the wrong granularity here: splitting it would put the old range's release into the schedule as a separate vertex, and a frees-first pass could then hand that range to another claim before the copy ran.

21.3.1 Edges

1. Claim(id) → every action writing into id.
2. A transfer reading id → Release(id).
3. A transfer reading a range → later actions writing that range. A write-after-read anti-dependency, easy to forget because it runs backwards from the usual intuition.

4. Earlier writes to a range → a transfer reading it. The fold has usually already turned these into literals, so this edge mostly evaporates.

It is a DAG by construction: every hard edge points from a lower journal position to a higher one.

21.3.2 Frees-first is a preference, not an edge

Running releases before claims gives placement more space to work with. It must be encoded as a **priority**, never as an edge — a conversion produces `Claim(new) → Transfer → Release(old)`, and adding `Release → Claim` as a hard edge makes that a three-cycle.

So: Kahn's algorithm with a priority queue over the ready set, in order

1. anything that unblocks a release,
2. releases,
3. claims, largest first,
4. everything else.

First-fit-decreasing therefore becomes *best-effort over the ready set* rather than global. Only conversion-style chains block, but the guarantee is weaker than an unconditional sort and should be stated where it is relied on.

21.3.3 Why the graph may be unnecessary

A worked example. After an earlier flush the file holds A1 (size 256) and A2 (size 128), both persistent. The journal records:

1. `Copy(src = A1, src_off = 0, len = 64, dst = A2, dst_off = 32)`
2. `Free(A1)`
3. `Resize(A3, 200)` // A3 is new; $200 \leq 256$, so it would fit in A1's space
4. `Write(A3, 0, [64 bytes])`

With a naive priority the ready set is `{Transfer, Claim(A3)}` — the release is blocked — and claims outrank transfers, so `Claim(A3)` runs first while A1 is still live, and A3 lands elsewhere. Note the scheduler did not *postpone* the free; it ran it as early as the dependencies allowed, and that was not early enough. With priority rule 1, the transfer runs first, the release becomes ready and top-priority, and A3 reuses A1's space with no address-level reasoning at all.

Hoisting. At fold time, if a `Storage(src, ...)` piece references a range this flush will **disturb**, read those bytes immediately and turn the piece into a `Literal`. Disturbed means any of: this flush releases `src`, writes the range, or reshapes `src` (since it may relocate). All three are known during the fold, which already runs at flush with full read access.

Applied to the example: A1 is released, so A2's middle piece becomes a literal during the fold. The transfer disappears, the release loses its only predecessor, and frees-first applies unconditionally. Same result, and the only cost is 64 buffered bytes — bytes that were going to be read anyway.

Hoisting every disturbed read collapses the DAG. Every surviving transfer then reads a range that is not released, not written, and not reshaped, and its source stays live so no claim can be placed over it. The only remaining edge is `Claim(dst) → Transfer`, which the fixed phase order *releases → claims → reshapes → transfers and writes* already satisfies.

So: **build hoisting first and skip the graph entirely.** It is dramatically simpler and it is correct; its only cost is memory proportional to the volume of disturbed reads, which for kladde's containers is small. Build the scheduler when a workload shows the buffering hurting.

The weakest point is that the hoist condition is conservative on *reshaped*: an allocation whose size changed might not actually relocate, but the fold cannot know before placement runs.

21.3.4 The content-blind fast path

Set a flag when any record produces a piece `Storage(other_id, ...)` with `other_id` not equal to the id itself. That is `Copy`, `Move`, and `conversion-on-a-persistent-id`, and nothing else — in particular **not** `plain resize`, and **not** `splice`, whose source and destination are the same id.

When the flag is clear at flush — the overwhelming majority of flushes — run *releases* → *first-fit-decreasing claims* → *reshapes* → *writes* with no graph, no hoisting, and no per-piece analysis.

21.4 The header as write buffer

The root of the address table is the header page, and the header page is rewritten by every flush no matter what. That page is free real estate, so it doubles as the address table's **write buffer**: every flush *takes its content whole* and states it again, and the cut fills the header before any leaf, at zero additional page writes.

Consequences:

- The common small flush writes **no** address-table page beyond the header.
- The header always holds **resolved truth** at the current epoch, which is why a statement evicted from it needs no re-resolution, and why multi-epoch disagreement can exist only between the header and leaves, or between two leaves.
- A statement carried in the header becomes a *new* record at the new epoch each flush, so an anchor re-points and its pin moves every flush. Header pages are exempt from coverage-driven victim selection, since they are rewritten unconditionally.

When the header overflows, the coldest statements go to a fresh leaf instead, ranked by an **eviction clock** recording how many flushes each has gone untouched.

The name overstates the machinery. What the header can hold is bounded by `MAX_PAGE_CONTENT`, so the clock covers on the order of 700 entries however large the file is, and the cut can simply sort the header's candidates by heat and keep a prefix. Nothing here needs a CLOCK sweep, an approximate-LRU, or a heap: an $O(n \log n)$ sort of 700 entries, on the rare flush that overflows, is beneath measurement.

The clock's entries belong to fragments, not to statements, since the cut *derives statements afresh* every flush; a statement derived from several fragments takes the hottest of them. An entry is reset when the application writes its fragment — when the fold takes it — not when the header merely carries it, and not when its allocation is read, since reads never reach the address table at all. Eviction and consolidation both state statements sorted by id, so leaves *tend* to cover coherent id ranges — a soft property worth cultivating and not depending on, since it keeps delta encoding dense. Evicting the coldest *key-contiguous run* rather than the coldest statements wherever they sit cultivates it at the source: heat still decides that a run goes, and key order decides which (see *the dirty set*).

21.5 The shape of the tree

Address-table pages form a tree rooted at the header, and it is a **directory, not a search structure**: every lookup is served by the in-memory fragment map, nothing ever descends the tree by key, so there is no ordering invariant to maintain across pages — and a B-tree's ~25 % expected slack is precisely the price of maintaining that invariant under inserts.

Interior pages are packed full, in arrival order; at each level only the newest interior page is partially filled.

- While everything fits in the header: **zero extra pages** — the steady state of every small file.
- Overflow spills to leaves referenced directly from the header, with statements continuing to share the header alongside the child references. At ~3 bytes per child reference, a header can reference on the order of a thousand leaves, i.e. a few MiB of statements, before this stops sufficing.
- Beyond that, one layer of interior pages reaches roughly a million leaves; deeper is not going to be needed.

Structural updates are copy-on-write path copies, but the paths are short and the root is free: adding or removing a leaf rewrites at most depth - 1 pages besides the leaf itself. Because interior pages are plain arrays of page numbers, removing a leaf's entry compacts its interior page for free during that rewrite. A depth change is a single-page event: when the header's child list outgrows its budget, move the whole list into one fresh interior page and reference that instead.

21.6 Correctness: the differential oracle

Build the oracle before the optimizer.

Keep a naive in-order replayer as a reference implementation, and differential-test the optimized flush against it: same journal, then compare the entire observable state — every live allocation’s geometry and bytes, compared in full. Nothing is masked out: every byte of a live allocation has exactly one right answer, so a schedule that differs from the replayer anywhere is wrong. Randomized journals with shrinking.

Two properties, asserted separately because they fail differently:

- **fold agreement** — the fold’s geometry equals the write-phase geometry the mutation path already tracked. Cheap enough to leave enabled outside tests.
- **schedule equivalence** — the differential comparison above.

21.7 Implementation order

1. **Log every mutation.** Fixes the recovery hole; the only step that is not optional.
2. **Write-phase geometry as a δ** over the committed state.
3. **The differential oracle**, before any optimization.
4. **The fold**, with the Uniform/spill representation and the fold-agreement assertion.
5. **Read hoisting** and the content-blind fast path. At this point splice and conversion become implementable, and copy becomes addable.
6. **The scheduler** — only if buffering measured in step 5 proves too expensive. It may never be needed.

21.8 Walk-through of an epoch

A finer-grained version of the [recovery case analysis](#), tracking page states step by step. This exists because two points in the argument are easy to get backwards: most pages of the older world are also in the newer one and therefore simply live, and a header slot whose write never arrived holds a *valid* stale header rather than garbage.

During the journaling phase of epoch E:

- Header E – 2 exists and has been fsynced.
- Header E – 1 was written but not fsynced yet.
- Journal segment E is partially written, not fsynced.
- All Data and AddressTable pages of world E – 1 have been fsynced and are **live** — this flush must not overwrite them.
- Pages of world E – 2 that are *not also* part of world E – 1 are **fallback**, and must not be overwritten either. This includes journal segment E – 1, which header E – 2 names. Pages belonging to both worlds are simply live, which is the common case rather than the exception: typically most pages of world E – 2 are in both.

Steps 1–7 of the flush protocol write data and address-table pages for epoch E, then fsync. No header page is touched. On a power cut during or after this:

- *If the slot written by flush E – 1 holds header E – 1* — guaranteed once fsync E completed, possible earlier — state E – 1 is recovered, the start page of journal E is found from that header, and a possibly-empty prefix of journal E is replayed.
- *If that slot does not hold header E – 1* — either its write tore, leaving an invalid CRC, or it never reached the disk, leaving the stale header E – 3 with a **valid** CRC, which is the more common sub-case — then header E – 2 has the highest epoch among valid headers and governs. Recovery reaches state E – 2 and replays journal E – 1 in full, arriving at the same state as the other case with an empty journal-E prefix. The lost journal-E transactions were never covered by a completed fsync, so this is within the guarantee.

Recovery is indifferent between the torn and never-arrived sub-cases, because it selects the valid header with the *highest epoch* rather than merely a valid one.

Step 8 overwrites header $E - 2$ with header E . This turns journal $E - 1$ and everything reachable only from header $E - 2$ from fallback to **reusable**, and it turns everything reachable from $E - 1$ but not from E from live to **fallback** (including journal E). On a power cut during or after:

- *If the slot holds header E :* the flush completed; recover to state E plus whatever prefix of journal $E + 1$ accumulated.
- *If it does not:* header $E - 1$ exists with a valid CRC, because it was fsynced in step 7, and carries the highest epoch among valid headers. Same situation as above, except that header $E - 2$ may no longer exist — which is fine, since it is only consulted on the path where header $E - 1$ is absent.

Chapter 22

Consolidation

Reclaiming garbage: which pages to rewrite, in what order, and how much per flush.

Consolidation rewrites live content out of sparse pages so their slots become reusable. It is [not part of the specification](#) — an implementation that never consolidates is conforming, and merely useless — but the policy below is what makes the copy-on-write design’s space overhead bounded rather than unbounded.

22.1 The high-level picture

Consolidation fights **three independent debts**, and the counters that measure one are blind to the others.

Description fragmentation is a property of an *allocation*: how many statements it takes to say what its bytes are. It is measured by [fragment_count](#) and [statement_bytes](#) against size; it costs memory in the fragment map, bytes in the address table, and time at load; and no page-level counter can see it, because an allocation described by two hundred statements packed densely into one full page makes that page look perfect.

Page fragmentation is a property of a *page*, Data and AddressTable alike: how much of its current state still relies on. It is measured by [coverage\[p\]](#) against capacity; it costs file size and nothing else; and no per-id counter can see it, because an allocation described by one Ref per flush has ideal `fragment_count` however thinly its bytes end up spread.

Outer fragmentation is a property of the *file*: reusable pages below its last live page. It costs file size, since a file can only shrink from its end, and neither kind of counter can see it, because every page involved is either wholly reusable or wholly live. It arises whenever the live size falls and stays down — after a large deletion, say — since pages freed anywhere but at the end become holes that only future writes can fill.

The first two convert into each other, each in exactly one direction, which is why neither can be left to the other’s mechanism:

- description fragmentation *becomes* address-table page fragmentation, since more statements need more table pages;
- repairing description fragmentation *creates* data-page fragmentation, since rewriting a stipple as one Ref kills the patches it supersedes.

Only page fragmentation is urgent, and that asymmetry is what decides how each debt is scheduled. Page fragmentation grows with write traffic and compounds, since garbage left uncleaned occupies the pages the next flush wanted to reuse — so it is attacked greedily, worst first. Description fragmentation is a standing debt that does not grow while nobody writes the allocation — so it need only be paid down *eventually*, which a rotating sweep discovers without any ranking structure of its own. Outer fragmentation is not a debt at all while the file grows, because growth fills the holes first; it becomes one only once the live size has fallen, and is then paid down in [a mode of its own](#).

22.1.1 Four mechanisms

mechanism	pays down	selected by	reads	writes
Evacuation	page fragmentation of Data pages; outer fragmentation, in compaction mode	the sparsest data pages, one victim at a time — the highest first, in compaction mode	the victim's survivors, from the file, found through the reverse index	the survivors, into data pages
Page rewrite	page fragmentation of AddressTable pages; outer fragmentation, in compaction mode	the sparsest table pages, one victim at a time — the highest first, in compaction mode	the victim, decoded	the victim's live content, stated again at the cut
Rotating window	delta-encoding density	a cursor that rotates through the fragment map	the fragment map	restatements, into the room the cut leaves over
Description defragmentation	description fragmentation	the stipples the rotating window's walk finds	the stipple's bytes, from the file	one Ref per candidate, at most a page long, among the flush's own chunks — or one Inline, below the threshold

All four end in the same place — fresh pages plus fresh statements — which is what lets [one placement pass](#) serve all of them.

22.1.2 Why address-table pages have two mechanisms

The page rewrite drains table pages and the rotating window keeps them dense, and it is [deriving every statement from one dirty set](#) that divides the work that way.

Once every table-side mechanism merely takes fragments and the cut states them all together, a restatement costs its encoded bytes rather than a page, and the cheapest way to empty a victim is to restate exactly what its statements still own — which is what the page rewrite does. A window that emptied the same victim would restate a superset: everything live in the victim's key span, including whatever other pages hold there. So the page rewrite does the draining, and it selects victims exactly as evacuation does, sparsest first from [a bucket queue of its own](#). That includes a table page whose live statements are scattered across the whole id space, which no window could drain but which is, to the page rewrite, simply a sparse victim.

What the page rewrite cannot do is make its output denser in key space than its input. Statements that went cold together and were evicted from the header together may lie far apart in the id space, and each of them then pays nearly full-width varints for its id and offset — two to four bytes more than among dense neighbours, on statements of about ten. The rotating window repairs that: it restates a contiguous key range, so its output is as dense as the encoding allows. That is worth too little per byte to compete for the budget, so the window gets only the room the victims leave over — which is also all it needs, since density is never urgent.

Both mechanisms copy Inline payloads out of table pages, and the page rewrite decodes its victim, so [address-table pages stay resident](#) by default; at ~8 MB per million allocations that is cheap. Nothing depends on it, though: dropping leaves after load, as data pages are dropped, costs one read per victim and one per table page whose inline payload the window restates.

Runs of small InLines are handled in two stages, and neither needs a mechanism of its own. The cut [merges adjacent inline fragments](#) into one Inline — every flush for the header's, and for a leaf's whenever a page rewrite or the window takes them — capping the merged payload at the [Inline threshold](#) rather than the format's 251 bytes, since a larger payload is exactly the cold, resident bulk the threshold exists to keep out of the address table. A run that outgrows the threshold — a vector accumulating small elements over many flushes, say — is then a dense stipple, which [description defragmentation](#) finds and

hoists into Refs a page at a time; the cut could not do that itself, since it runs after pack and places no data.

22.1.3 Description defragmentation rides the rotating window

Description defragmentation needs no victim selection of its own, because the rotating window already walks every fragment of a contiguous key range and therefore already sees which allocations are stippled.

The decision is per id and entirely local: the walk visits this id's fragments anyway, so the only question is whether a patched region is worth replacing with one fresh statement over zero-filled bytes. That statement is a Ref over a new data chunk, or — when the region is no longer than the [Inline threshold](#) — an Inline, which needs no data page at all: its pending fragment is placed Inline, so pack never sees it, and the cut puts its payload in whichever table page it chooses.

Both sides carry data bytes and statement bytes, and getting that right is the whole of the criterion. The data terms very nearly cancel, because the fragments a rewrite supersedes lose their coverage exactly where they pointed:

```
freed(a, b)    =  $\Sigma$  live bytes of the `Bytes` fragments in [a, b) // coverage that drops
               +  $\Sigma$  framing of the statements that die
written(a, b)  = (b - a)                                           // including the gaps
               + the new statement's framing                       // a `Ref`'s or an `Inline`'s
```

Subtracting leaves a criterion that is worth stating on its own, because it is the thing to reason about and it is not what the ratio looks like at first glance:

A defragmentation pays when the statement bytes it retires exceed the zero bytes it materialises — by a margin of μ for every byte it writes. $\text{framing_retired} - \text{new_framing} - (1 - \text{density}) \cdot (b - a) > \mu \cdot (b - a)$, with density the fraction of [a, b) that currently resolves through a statement.

μ and the [churn floor \$\lambda\$](#) are the same kind of price — file bytes freed per byte written — and differ because description defragmentation is paid mostly in a currency neither of them measures. Its net saving in file bytes is at most the framing it retires, while what it chiefly buys is invisible in the file: a fragment-map entry per statement retired, and the work of resolving those statements at every load. So its floor has to sit far lower than evacuation's for it to do anything at all: λ near 1, and $\mu \geq 0$ near 0 — at $\mu = 0$ a defragmentation must merely not grow the file. Tying the two as $\mu = \lambda - 1$ would give sensible values only at $\lambda \approx 1$: a cautious $\lambda = 2$ would force $\mu = 1$, where a rewrite had to save a byte of statements per byte it writes, which a run of 16-byte inlines saving 3 bytes of framing each never does. If one price is wanted after all, the way to it is to make the hidden benefit explicit — credit each retired fragment with the memory it frees — and then weigh defragmentation against the same λ as everything else.

22.1.4 The envelope is a maximum-subarray problem

There is no need for a heuristic, and no need to choose between “all” and “nothing”. Attribute the criterion above to each fragment and the best subrange is the maximum-weight contiguous run of them — Kadane's algorithm, one pass, on a sequence the rotating walk is traversing anyway.

```
/// The rewrites worth doing in `id`, each at most one page long. Weights are *net*
/// file bytes — what a rewrite releases minus what it consumes — less `mu` per byte
/// written, so a range that would grow the file, or cost more to write than it
/// saves, weighs below zero.
fn defrag_candidates(id: Id, size: u32, mu: f32) -> SmallVec<DefragCandidate> {
    // `share` spreads a statement's framing over the fragments it owns — `pins`
    // counts them — so a statement wholly inside the range is credited in full.
    let weight = |f: &Fragment, len: u32| {
        let share = f.statement().map_or(0.0, |s| framing_len(s) as f32 / pins(s) as f32);
        let net = match f {
            Bytes{..}          => share, // its bytes move: released, consumed
            ZeroExplicitly{..} => share - len as f32, // its zeros become real bytes
            ZeroByDefault      => -(len as f32), // likewise, and nothing is released
        }
    }
    ...
}
```

```

    Pending(p)          => estimate(p, len),      // taken this flush; see below
  };
  net = mu * len as f32
};
let Some(⟦_, range⟧) = kadane(fragments.range((id, 0)..(id, size)), weight) else {
  return SmallVec::new();
};
// A `Ref` holds at most one page, so a longer range becomes consecutive tiles,
// each weighed on its own; see below. A tile's gain is net of the statement it
// adds, an `Inline`'s framing or a `Ref`'s, so candidates rank by gain directly.
tiles(range, MAX_PAGE_CONTENT)
  .map(|tile| {
    let framing = if tile.len() <= INLINE_THRESHOLD { INLINE_FRAMING } else { REF_FRAMING };
    DefragCandidate { id, gain: weight_of(tile, weight) - framing, age: age_of(id, tile), rang
  })
  .filter(|c| c.gain > 0.0)
  .collect()
}

// A lower bound on how long the application has left `tile` unwritten; see below.
fn age_of(id: Id, tile: Range) -> u32 {
  let youngest = fragments.range(tile)
    .filter(|f| !matches!(f, ZeroByDefault))      //no statement, so no age
    .map(|f| f.statement().map_or(0, flushes_since_placed)) // pending: 0
    .min();
  max(flushes_since(allocations[id].last_written), youngest.unwrap_or(0))
}

```

This code example shows the calculation of the maximum envelope as a free-standing function for best readability. The real implementation folds Kadane's algorithm into [the rotating walk](#), which iterates over fragments anyway, evaluating and resetting the max-weight subarray records at every `id` boundary of that walk. A walk that starts or stops inside an `id` sees only part of it, and Kadane then finds the best subrange of that part; a stipple that a walk boundary cuts in two is judged as two halves, which errs toward doing less.

The weights are what make the answer come out right at both ends. A run of `ZeroByDefault` scores $-(1 + \mu) \cdot \text{len}$, so the envelope stops at the edge of a stipple rather than swallowing the sparse allocation around it. A single large `Ref` scores $\text{share} - \mu \cdot \text{len}$, which is negative for any `Ref` longer than share / μ — a few hundred bytes at $\mu = 0.02$ — so an already-contiguous region is excluded, which is correct, since rewriting it merges no statements and only moves bytes. And a long allocation carrying a short, dense knot of tiny `Inline` patches is exactly where the run of large positive weights is, so it is what Kadane returns.

A candidate is at most one page long, so that it is exactly one `Ref` — a `Ref` [holds at most one page's content](#) — and therefore exactly one chunk, or, if short enough, one `Inline` and no chunk at all. Kadane's range has no such bound, so it is tiled into consecutive candidates of at most a page each, and each tile is weighed on its own and kept only if its gain, net of the statement it adds, is positive. Tiling loses nothing, since a range longer than a page needs one `Ref` per page however it is rewritten, and it lets the reserved share take a long stipple a page at a time. Bounding Kadane's range instead would find only the single best page-sized window, and would leave the rest of a long stipple to later rotations.

22.1.5 Where it is called, and how it is executed

Discovered by the rotating walk, executed from a share of its own. The walk visits every `id`'s fragments in offset order, so Kadane costs nothing extra, and [rotating_window](#) returns the candidates it found. They cannot compete with evacuation for the budget, since what they pay down is description rather than space, and a net saving of a few bytes of framing would lose to any victim. So they are paid from a **reserved share** of each flush's budget, and **the share is spent before packing, not during it**: at the start of each flush, the best-scoring candidates are chosen up to the share, and their rewrites join the flush's

own chunks. Once chosen, a rewrite is a write like any other, so `pack` places it in key order among the flush's own chunks, and every rewrite chosen is written. Offering rewrites as filler instead would starve the valuable ones: what a page has left after the flush's own content is rarely room enough for a rewrite of any size, and one near a page long would fit only by the accident of a nearly empty page.

Among themselves, candidates are ranked the way data pages are,

`score = gain · age / written`

`age = max(the allocation's age, the youngest age among the tile's fragments)`

Both terms are lower bounds on how long the application has left the tile's content unwritten, and each covers the other's blind spot:

- **The allocation's age** — flushes since `last_written` — sees through restatements, which re-stamp a statement's epoch without touching its content. The header re-stamps everything it holds on every flush, so in a small file whose statements all live there, epochs alone would make every allocation look brand new forever.
- **A fragment's age** — flushes since the statement that owns it was placed, or zero if the flush in progress has taken it — sees through writes elsewhere in a large allocation, which reset the allocation's age but not the tile's.

`ZeroByDefault` fragments have no statement, and are left out of the minimum. The allocation's age survives a reopen through the `consolidator state`; rebuilt from the file alone, it would make every allocation with inline content in the header look young at every open, since the header rewrites what it holds.

The age term is the same *prediction* it is in `data-page scoring`: a range the application patched recently is likely to be patched again, which would re-stipple it and waste the rewrite. So a cold allocation's candidate rises the longer it waits, however marginal its gain, and the reserved share guarantees that it is reached.

A candidate found in one flush is executed in a later one, because the walk rides on the rotating window, which runs at the cut, after the flush's data pages are packed. So it is re-checked when it is chosen, after the executing flush's fold has taken what it changes: its gain and age are recomputed from the fragments now in its range, which is cheap, and it is dropped unless its gain is still positive and **its age is at least one flush**. That filter is what keeps a rewrite off bytes the executing flush is writing — a tile holding a fragment the fold just took has both terms at zero — and it has to be a filter rather than only a factor in the score, since a zero score can still be chosen while the share has room. The walk itself may weigh fragments the flush has already taken only by estimate, since the statements that will own them are not bound yet; the re-check makes that harmless.

A session's first flush draws on a read-only walk at open, over the window the previous session's last walk covered, which the `consolidator state` records, or else from wherever `open seeds the rotation`. Otherwise the candidates that last walk found would be lost at close, and a session of one flush would find candidates and never execute any.

Executing one needs no new operation: it is a `take` of the whole range, with the range's resolved content — read from the file and the resident table pages, gaps memset to zero, assembled in the journal's arena — as the bytes to write. Taking replaces everything under the range with one pending fragment and releases the old owners' pins, and the cut states it as one Ref, or one Inline if the range is short enough. Unlike `write_bytes`, it leaves `last_written` alone, since the content does not change.

A per-id bucket queue over `statement_bytes * B / max(size, 1)` remains available if one rotation ever proves too slow to discover what needs doing. It should not be built before that measurement exists, being a second structure to maintain on every statement created or destroyed.

22.1.6 How a flush and consolidation compose

Every source takes fragments as it goes, and statements come last; that gap is the whole interface between a flush and its consolidation. The fold, the header, evacuation, the page rewrite, the rotating window, and description defragmentation all do the same two things: they `take` the fragments they change, which releases the old owners at once, and they add the range to the flush's dirty set. No statement exists until the address table is cut; then every one is `derived from the fragment map as it finally stands`, so that what the flush states is by construction exactly what it changed.

A taken fragment is **pending**, and its [entry](#) records where its bytes are now and where they will be stated from:

```
enum Origin {
    Arena(ArenaPos),    // written this flush
    File(Address),      // already in the file
    None,               // zeros: nothing to read
}

struct ArenaPos(u32); // bounds the arena at 4 GiB; see below
```

A **pending fragment whose bytes have no place yet is a chunk**, and that is the one thing pack needs to know: it places the chunk, and the fragment has its location. A pending fragment that keeps its location — carried by the header, restated by a page rewrite or the window, shifted by a splice — needs no page, only a statement. **A relocated survivor is therefore the same object as a fresh write**: a chunk, differing only in whether its bytes are read from the file or from the journal’s arena. A survivor’s bytes are always a data page’s, since evacuation takes only data victims, and an inline payload never becomes a chunk by moving: a page rewrite takes it in place, and the cut copies it into its new table page. The one chunk that reads resident table pages is a description defragmentation rewrite over a range holding inline payloads, and it assembles its bytes in the arena first, so to pack it too is a fresh write. Placement cannot tell them apart, and does not need to.

A 32-bit ArenaPos halves Origin: six bytes of payload in its larger variant rather than eight, so eight bytes with the tag rather than sixteen. That is a saving of width, not of a niche — `Option<Origin>` would cost nothing extra at either width, since the tag has spare values. It also turns an assumption into a bound the implementation must enforce: the arena holds everything written since the last flush, so a flush must be forced before it would outgrow 32 bits, and a single transaction larger than that, which no flush can split, must fail cleanly — which is what [the bound on transactions](#) requires.

22.1.6.1 Mandatory and elastic

Placement combines two streams: what the flush must write and what consolidation may add. **One stream must be written whole and the other may be cut short**, and that asymmetry is the only thing the packer needs to know about where a chunk came from. A commit that omits part of what the journal produced is not a commit. A survivor left behind costs a delay and nothing else. Description defragmentation’s rewrites belong to the first stream, since [they are chosen before packing](#), and once chosen they are writes like any other.

Neither stream is cut into pages before the other is visible, because pre-cutting the mandatory stream would freeze exactly the boundaries that the elastic stream exists to avoid. The elastic stream is instead drawn **on demand**: a victim is pulled into a page only at the moment that page has room its own content cannot use, and only as much of it as fits. That is what lets consolidation fill every page the flush writes to nearly full — cutting a fixed stream into pages would typically leave its last page sparse — without ever opening a page it cannot fill.

22.1.6.2 Slack is cheaper than a split — up to a point

Do not cut a chunk to make a page come out exactly full; cut only to keep a page from closing more than θ empty. In principle, the format would allow cutting Ref statements at arbitrary offsets and thus packing all emitted data pages exactly full. However, the reference implementation does not try to fit data pages *exactly* full and instead accepts a small amount of slack in each page. Filling data pages exactly would reduce page fragmentation at the cost of description fragmentation, with an important asymmetry: page fragmentation is *recoverable*, since the next consolidation to pick the page up re-packs it, whereas a split is a statement and a fragment-map entry that survive until [description defragmentation](#) pays them off — and description debt is the debt with no natural drain.

	costs	until
leaving slack	at most θ of a page, on every page but a flush’s last, and usually far less	the page is next consolidated

	costs	until
splitting to fill it	~10 bytes of address table, ~20 bytes of fragment map, and one more entry in every scan of that id	something rewrites the range as one Ref

Slack has no small bound of its own, which is what θ is for. Without it, what bounds slack is the smallest chunk still within the packer's reach, and that is only as good as the chunks are: close to the Inline threshold when small chunks are plentiful, and nothing like it when they are not. Chunks all just over half a page are the adversarial case — no two can ever share a page, so if we didn't ever cut them then every page would close half empty, whatever the look-ahead.

So a page that would close more than θ empty is filled instead, in the order that costs least:

1. with whole victims, which ride for free in a page that is being written anyway;
2. failing that, by cutting the flush's next chunk at the page boundary.

That makes the guarantee unconditional: **every data page a flush writes, except its last, is at least $1 - \theta$ full**, whatever the chunk sizes, at a cost of at most one cut per page — and in the adversarial case the cuts are exactly what halve the page count. The chunk cut is the next one in key order rather than the largest within reach, so its two halves stay adjacent in key order and in page order, and a tail shorter than the Inline threshold becomes an Inline rather than a chunk.

On a page that holds content of the flush's own, that content is what gets cut, not a victim's survivor. Both cuts cost exactly one extra statement. But the flush's own content is what the application is working on — the same [temporal update locality](#) that data-page scoring rests on — so its cut is the likelier to be overwritten soon and take the extra statement with it, whereas a survivor's cut would sit in cold data until description defragmentation found it. For the same reason a description defragmentation rewrite, though it travels with the flush's own chunks, is cut only when none of the genuine ones is within reach: it is cold by selection, since that is what its age term chooses for.

A page opened only for consolidation has no such content, and there a survivor is cut instead, its tail leading the next page. Refusing to would leave the adversarial file at half fill for good: every victim in it holds one chunk just over half a page, so no two victims could ever share a page, and consolidation could never improve it.

22.1.6.3 Packing in id order, with look-ahead

Pack in (id, offset) order and fit by looking ahead, rather than first-fit over a size-sorted list. First-fit-decreasing packs tightly and destroys id locality completely, which costs two things worth keeping: a smaller [reverse index](#), which holds one entry per (page, id) pair, and the chance that a later evacuation of the page relocates one id's bytes together and merges their fragments.

```
fn pack(mut chunks: Lookahead<Chunk>, elastic: &mut Elastic) -> Vec<Page> {
    let mut pages = Vec::new();
    if chunks.is_empty() { return pages; }           // no page is being written anyway
    let mut open = Page::new();
    loop {
        // 1. The first of the flush's own chunks that fits, in key order, within reach.
        //    They include the description defragmentation rewrites the share paid for.
        if let Some(c) = chunks.take_first_fit(open.room()) { open.place(c); continue; }
        // 2. None fits: free filling, with whatever fits, in the order that pays most.
        while let Some(e) = elastic.take_fitting(open.room(), open.number()) { open.place_all(e); }
        if chunks.is_empty() { break; }              // the flush's last page: short only if nothing fits
        // 3. Still more than  $\theta$  empty: cut the next chunk so the page closes full —
        //    a genuine one of the flush's own, unless only rewrites are within reach.
        if open.room() > THETA * MAX_PAGE_CONTENT {
            let (head, tail) = chunks.pop_next_to_cut().split_at(open.room());
            open.place(head);
            chunks.push_front(tail);                  // shorter than the `Inline` threshold: an `Inline`
        }
    }
}
```

```

    pages.push(mem::take(&mut open));
}
pages.push(open);
pages
}

```

Chunks longer than a page never reach pack: whole pages are cut off their front first and written on their own, [as the flush's placement does](#), since that is the one cut that is forced rather than chosen.

Step 2 fills the room with anything that fits and costs nothing, on every page the last included, in the order that pays most:

1. whole victims — the tail first, in [compaction mode](#) — since each frees a page;
2. description defragmentation candidates beyond the share, [re-checked](#) as the share's are, since each retires statements for no page write;
3. part of a victim too big to take whole — the survivors of as many of its statements as fit, a whole statement's at a time, so that nothing is cut — which frees no page yet but leaves the victim sparser, and cheaper to take later.

So the last page closes short only when nothing at all fits. Leaving it short on purpose, as a ready victim for the next flush, would not work anyway, since the age term would keep a page that young from being chosen.

Victims are pulled in while packing, not chosen before it and fitted afterwards. So there is no estimate of the page count to get wrong: a page closes when the flush's own content no longer fits it, the room it has left at that moment is known exactly, and whatever fills that room is chosen to fit it. Nor can a victim force a page open: step 2 moves only what fits the room, and a page that exists only for consolidation's sake is opened by [the budget loop](#), and only when its offer fills it. Slack in every page but the last is then at most θ of it, and usually far less, since step 2 fills the room with whatever victims fit before step 3 would cut; id order survives up to a reordering within the look-ahead.

22.1.6.4 One dirty set, and why statements are derived last

The cut states exactly the dirty set, and derives the statements that do so only once nothing will change the fragment map again: only the final map says which statements are needed, and [delta encoding](#) makes each one's length a property of its predecessor, so lengths cannot be known before the statements are.

```

/// Everything the flush in progress has taken, and so must state.
struct Dirty {
    ranges: IntervalSet<Key>, // merged on insert: overlapping or touching ranges fuse
    records: HashMap<Id, Record>, // per touched id, what the fragment map cannot say
}

```

Its ranges come from every source:

1. **the header's**, [taken whole](#) at the start of every flush — the header is rewritten every flush, so its content is this flush's to state as much as any new write;
2. the fold's, description defragmentation's, and every relocated survivor's;
3. the page rewrite's table victims', and the rotating window's.

Its records say how each touched id's size and existence changed, and whether a page the flush retires held its anchor or grow witness; the cut turns them into Shrink, Grow, and Tombstone statements by [the rules the mutation functions give](#).

Deriving is one pass over the merged ranges, and it merges across sources for free:

```

/// Every statement this flush writes, from the fragment map as it finally stands.
/// Each pending fragment falls in exactly one run, so it is stated exactly once.
fn derive(dirty: &Dirty) -> Vec<Statement> {
    let mut out = Vec::new();
    for id in dirty.ids_in_key_order() {
        // One statement per maximal run of pending fragments that state alike: zeros
        // with zeros, bytes with bytes that sit contiguously in one data page, and
    }
}

```

```

    // inline bytes with inline bytes up to the `Inline` threshold.
    for run in pending_runs(id, dirty.ranges.of(id)) { out.push(run.statement()); }
    // What the fragment map cannot say, merged in at its own key.
    out.insert_sorted(size_statement(id, &dirty.records[id], &out));
}
out
}

```

Two fragments from different sources — the flush’s own write and a survivor that pack placed right after it, say — become one Ref when they are adjacent both in the allocation and in the page. Inline fragments need no such luck, since the cut copies every inline payload into its new table page anyway: adjacent ones merge into one Inline whatever their origin.

The cut then lays the statements out in three steps, and every page but one comes out full:

1. **The header first, with the hottest statements it can hold.** When every statement fits, every statement goes there and no leaf is written, which is the steady state of every small file.
2. **The rest into leaves, in key order,** each leaf closed when the next statement does not fit, so that every leaf but the last is full to within one statement.
3. **The one page that still has room — the last leaf, or the header when there is no leaf — takes the fillers:** whole table victims that fit, then [the rotating window](#), which can stop at any byte and so fills the room exactly.

A leaf therefore exists only when the header is full, so the last leaf’s content could never have fit in the header instead, and no layout needs fewer pages. The one page that can end up more than a statement short — well under θ — is the one the fillers could not fill because nothing was left to take.

The header keeps the hottest statements, not the tail of the key order. The header is [the write buffer](#): a statement that stays in it is rewritten for free every flush, so an allocation the application keeps touching never leaves a statement to be shadowed in a leaf. Filling the header from the end of the sorted statements would evict a hot statement whenever it happened to sort early, and each of its later updates would then leave garbage in a leaf. Key order is right for the rest: the statements the header does not keep are cut into leaves in key order, which is what keeps leaves dense. Evicting the coldest *key-contiguous run*, rather than the coldest statements wherever they sit, gets both at once — heat decides that a run goes, and key order decides which.

Fillers cost only what they newly take. Both take more ranges into the dirty set, and the cut derives what they add; a range that was dirty already adds nothing.

Subadditivity of the encoded length is what makes fitting them cheap. $|\text{encode}(A \cup B)| \leq |\text{encode}(A)| + |\text{encode}(B)|$ for any two sets of statements that may share an epoch, because merging only ever brings each element’s predecessor closer and a varint’s length is monotone in the value it encodes. The one condition is that the union be conflict-free — [each fragment stated once](#) — since two statements claiming the same byte would need the offset cursor to step backwards, which the encoding cannot express. So a filler’s length, measured on its own, is a safe upper bound on what it adds: filling to that bound under-fills rather than overflows, and a second pass over the merged statements recovers the difference. (Subadditive is the word; convexity is a property of functions on a vector space, which a function of sets is not.)

Three details of the bookkeeping:

- Lengths are computed *with page boundaries*: each leaf’s first statement pays its full-width id and offset, since both cursors reset at a page boundary; a leaf spends one delimiter byte on its empty child list; and a statement that does not fit starts the next leaf.
- The header’s capacity is what remains after the file-header fields and its child list, and the child list depends on how many leaves this flush adds and unlinks, so step 1 works against an estimate and is corrected once: if the list turns out longer, the header gives up its coldest statements to the leaves.
- A child list that no longer fits the header at all is not a case to optimize, but [the depth change](#) the tree already provides: the whole list moves into one fresh interior page, which the header references instead.

Binding comes last. Only when every statement's page is known does the cut [bind](#) each one: a slab slot, its fragments — each pending fragment turning into the Bytes or ZeroExplicitly fragment that names it — its pins, and its framing, charged to its page with an inline payload per byte. A Shrink or Tombstone becomes the anchor then, and a Grow the witness.

22.1.6.5 Each fragment is stated once per epoch

Everything a flush writes shares one epoch, so no two of its statements may claim the same byte — and deriving the statements from the final fragment map is what guarantees it. Each pending fragment falls in exactly one run and each run becomes exactly one statement, whatever order the sources took their ranges in and however those ranges overlapped.

Were each source to create its statements as it went instead, a later source taking fragments from a statement an earlier one had already created would leave both claiming the same bytes, and a flush that assembles its output from several sources would do so in many ways:

- evacuation moving a survivor that the same flush overwrites, or that the header is carrying forward;
- the rotating window passing over a fragment that some other source had already restated;
- a page rewrite restating a Ref whose data page a later evacuation moves;
- a description defragmentation rewrite taking bytes from a statement the fold had just created.

Each would need a rule of its own, and the page-rewrite case would need the earlier statement rewritten after the fact. With statements derived at the cut, a later source just takes the fragments, and the earlier source's claim on them is simply gone.

Chunks, by contrast, are produced eagerly, when a source takes their fragments, **and pack fixes where their bytes go**. Both are safe because no source ever takes over bytes that another source of the same flush is writing, and nothing after pack moves a byte it placed:

- evacuation takes only fragments pointing into pages older than this flush, and a flush's own pages can never be its victims;
- page rewrites, the header, and the rotating window take fragments in place and never move bytes at all;
- description defragmentation's rewrites are chosen before pack, and [its age filter](#) keeps them off anything the flush is writing.

The one thing after pack that reads the layout is the derivation, which is exactly where it should: it is what merges adjacent chunks that pack placed contiguously into one Ref.

Two things follow. Coverage *releases* happen when a source takes a range, so victim selection sees the state the flush will leave with no provisional bookkeeping of its own; only the *charges* wait for locations — [at placement for bytes](#), and [at the cut for framing](#). And the fragment map now changes before the commit, so a flush that fails must unwind or [poison the session](#), which the reference implementation's poisoning already covers.

22.2 The budget

The design has one throttle — a per-flush **maintenance budget** in pages beyond the flush's own content — and one queue of offers competing for it, ranked by bytes reclaimed / bytes written. One further parameter, [the churn floor \$\lambda\$](#) , says when an offer is not worth taking at all.

Two sources compete for that budget, and they are one mechanism applied to two kinds of page:

1. **Evacuation** relocates a data victim's survivors, and reclaims dead data bytes.
2. **The page rewrite** restates a table victim's live content, and reclaims dead statement bytes.

The other two mechanisms stay out of the ranking, each for a reason of its own. **The rotating window** takes only the room the victims leave, because what it buys — density — is worth too little per byte to compete. **Description defragmentation** pays down a different debt: it reclaims almost no space, and what it retires is description, measured in statements and fragment-map memory rather than in file bytes, so it draws on [a reserved share](#) instead of competing.

Description defragmentation is worth naming as a mechanism of its own even so, because it is the only one that pays down [description fragmentation](#) and the only one that shrinks the in-memory fragment map. An allocation stippled with [data] [zeros] [data] ... patches costs statements forever until someone rewrites the region as one Ref — and the [Zero usage note](#) explicitly permits rewriting such a stipple as one Ref over a range whose gaps the consolidator fills with actual zeros, which is what makes the rewrite legal. Those bytes are being written anyway, so zeroing them costs nothing beyond the memset.

22.2.1 The structures behind the ranking

Every source that starts from a *page* is ranked from **bucket queues**, which is the structure that makes “give me a good victim” $O(1)$ instead of $O(\log P)$.

```
struct Buckets {
    lists: [Vec<PageNumber>; B],    // B is small; 16 is plenty
    at: HashMap<PageNumber, (u8, u32)>, // which bucket, and which slot in it
}
```

A page’s bucket is $\text{coverage}[p] * B / \text{capacity}$, so moving a page is: swap-remove it from its old list, push it onto the new one, and fix the at entry of whatever the swap moved. Three array writes and two map updates, no comparisons, no rebalancing — which matters because coverage changes on **every** fragment created or destroyed, not once per flush.

Each page kind gets its own instance. One shared instance would cost the $O(1)$ the structure exists for: data pages outnumber table pages by roughly the ratio of their contents, so `lowest_non_empty()` on a mixed structure would keep handing back data pages and the table side would scan past them. Both can use the same B, since the page rewrite is the table side’s primary mechanism and wants the same resolution evacuation does. Header pages belong in neither: a header is rewritten every flush, so it is never a victim, and its content is accounted for by being [taken and stated again by every flush](#) rather than by its coverage.

The rotating window needs no bucket structure at all, being driven by its cursor, and neither does the defragmentation it discovers.

Why bucketing is enough. A bucket queue answers “roughly the sparsest” rather than “the sparsest”, and the ranking is a heuristic whose inputs are estimates anyway. What it must not do is *miss* a good victim, and it cannot: a page in the sparsest non-empty bucket is within $1/B$ of the true minimum live fraction.

22.2.2 The budget loop

A flush and its consolidation run as one straight line:

```
fn flush(journal: Journal, budget: Budget) {
    // 1. Take: the header's content whole, then everything the fold changed. Every
    //    source from here on takes fragments and marks their range dirty, and none
    //    states anything; coverage falls as they take, so everything below sees the
    //    state this flush will leave. Description defragmentation's share is spent
    //    here as well: the best candidates earlier walks found, re-checked against
    //    the state the fold left, become chunks of the flush's own.
    let mut dirty = Dirty::new();
    take_header(&mut dirty);
    let fold = fold(journal, &mut dirty);
    let rewrites = defrag.choose(budget.defrag_share, &mut dirty);

    // 2. Data pages: the flush's own chunks in key order, with victims riding along
    //    wherever a page has room its own content cannot use. See `pack`.
    let mut data = pack(merge_by_key(fold.chunks, rewrites), &mut elastic);

    // 3. Budgeted pages, one at a time, to whichever page kind reclaims more per
    //    byte — and only a page the offer fills, since a page opened for a sliver
```

```

// would cost a whole page write for next to nothing.
for _ in 0..budget.pages {
    let Some(offer) = [evacuation.offer_page(), page_rewrite.offer_page()]
                        .into_iter().flatten()
                        .max_by_key(|o| o.reclaimed / o.written) else { break };
    if !offer.holds_tail() { // compaction mode's tail passes both floors
        if offer.reclaimed < budget.lambda * offer.written { break; } // the churn floor
        if offer.fill() < 1.0 - THETA { break; }
    }
    offer.take(&mut data, &mut dirty);
}

// 4. Address-table pages: derive the statements from the dirty set and cut them.
// The hottest stay in the header, the rest go to leaves in key order, and the
// last page's room goes to table victims that fit and then to the rotating
// window, whose walk also finds the next description defragmentation
// candidates. Binding comes last.
let table = cut(dirty, &mut page_rewrite, &mut rotating_window);

// 5. Unlink emptied pages, write, fsync, commit; then truncate a reusable tail.
#[cfg(debug_assertions)] fill_observer.record(&data, &table);
commit(data, table);
}

```

The two page kinds are combined, not traded off — but they do compete. Their victims fill one plan, and what they compete for is the one budget, a page at a time; each source's offers decline as it takes more, so one greedy pass over the two is the right amount of machinery.

Free filling happens inside steps 2 and 4, and step 3 is the only place a page is opened for consolidation's sake. It opens one only when the offer fills it to $1 - \theta$, which is what keeps a victim too many from costing a nearly empty page; only an offer holding [compaction mode's tail](#) is exempt. An offer is what the page would hold: whole victims first, by best fit, and then — if that leaves more than θ empty — one more victim with a survivor cut at the page boundary, exactly as pack cuts the flush's own chunk. A cut's tail leads the next page, which is therefore opened whatever its own offer, so the loop cuts nothing on its last page, and every chain of cuts it starts ends within the budget. The flush's data pages form one such chain: the last page pack returns may cut a victim's survivor too whenever the loop goes on to open another, so only the flush's very last data page can close short. Description defragmentation's reserved share is spent in step 1, before anything is packed, so its rewrites are packed in step 2 exactly as the flush's own chunks are — on a flush whose only data they are, too.

Debug builds observe how full the pages come out.

```

#[cfg(debug_assertions)]
#[derive(Default)]
struct FillObserver {
    header: Fill, // the header pages
    table: Fill,  // every other address-table page: leaves and interior pages
    data: Fill,   // data pages
}

#[cfg(debug_assertions)]
#[derive(Default)]
struct Fill {
    pages: u64, // written this session
    content: u64, // their `content_size`s summed; average fill is content / pages
    lowest: u16, // the lowest `content_size` written by the latest flush
}

```

It measures fill *at write time* — the packer's quality — which is a different number from the file's live fraction, the consolidator's quality, which the budget throttle reads. `record` also asserts what packing

promises: every data page of a flush but its last at least $1 - \theta$ full, and every leaf but the last full to within the length of the statement that did not fit. The header's fill is observed but not asserted, since a small file's header is legitimately almost empty.

22.2.2.1 The churn floor is a parameter, not an identity

The floor is a parameter λ — space freed per byte written — and its price is explicit. Cleaning pages at live fraction u costs $u / (1 - u)$ bytes written per byte of space freed, so a total write amplification of $1 / (1 - u)$: $2\times$ at $u = 1/2$, $5\times$ at 0.8 , $10\times$ at 0.9 .

But λ is the wrong knob for a target file fill. An average fill of 80–90 % does not mean cleaning pages that are 80–90 % live, and implementing it that way would buy the last few percent at five to ten times the write traffic. It means the distribution of live fractions is bimodal enough that most pages sit near full while the garbage concentrates in a few nearly empty ones — which is what [the age term](#) is for, and the whole reason LFS's cost-benefit cleaner beats plain greedy.

The two knobs therefore do different jobs, and only one of them is a floor:

- **λ is a churn floor.** It stops the cleaner grinding when there is nothing worth cleaning, and it belongs *low* — near 1, i.e. near $u = 1/2$ — precisely so that it is not what decides the file's fill.
- **The per-flush budget is the throttle**, and it is what converges the file on a target fill τ : measure $\text{live_bytes} / (\text{pages} \cdot C)$ after each commit and move the budget up while it sits below τ , down while above; the [consolidator state](#) carries the budget it has reached from one session to the next.

That also sharpens [the bound this buys](#): $\text{live_size} / \tau$ is reached by spending budget, and the floor's only job is to stop spending it on pages that cannot repay.

The known failure mode of the single ranked queue. Ratio-greedy ranking starves **costs that current ratios cannot see**, and one such cost remains in this design: description debt on a cold allocation — one that will be read forever and never written again — never looks urgent, because nothing about it is changing. It is taken out of the ranking rather than weighted inside it: [the rotating walk](#) discovers it, and a reserved share of the budget pays it. A table page whose live statements are scattered across the id space is not a second such cost, because the page rewrite selects by sparsity alone, to which scattering is invisible.

That is not a verdict against age terms. Age can do two different jobs, and only one of them is a starvation guard. In [data-page scoring](#) it is a *prediction*: a page written recently will likely get sparser by itself, so cleaning it now wastes the writes that relocate survivors about to die. That job is exactly right there, and the same prediction applies to defragmentation — a range patched recently will likely be patched again, which would waste the rewrite — which is why [defragmentation candidates are scored with age](#). What age would be wrong for is the other job, guaranteeing that a cost the ratio never sees is eventually paid: a weight large enough to force that is a tuning parameter standing in for a rotation, which guarantees it outright.

22.2.3 The complexity this incurs

Strictly logarithmic work *per flush* is unattainable, because a flush must at least record what it changed. The attainable form, and what this design achieves:

- $O((k + s) \cdot \log F)$ in-memory work for a flush dirtying k ranges and rewriting s statements under the budget;
- $O(1)$ victim selection per candidate;
- page writes bounded by dirty data pages + c + depth for budget c ;
- with the common small flush writing **no** address-table page beyond the header.

22.3 Two forms of consolidation

Consolidation runs inside every flush in two forms, which differ in [the price of the sink](#) and, following from that, in whether they are traded off against anything at all:

- **Free filling.** Victims, description defragmentation's remaining candidates, and parts of victims too big to take whole ride in the room of pages that are being written anyway — the flush's own data

pages, and the header or the last leaf — so they cost no page write and compete with nothing. It happens inside [packing](#) and inside [the cut](#), and it runs to exhaustion.

- **Budgeted pages.** Beyond that, pages that exist only to reclaim, offered one at a time and opened only when the offer fills them. These compete, under [the budget and the churn floor](#).

22.4 Victim selection for data pages

The classic greedy rule of log-structured file systems — **lowest live fraction first** — refined by age the way LFS and SSD flash translation layers refine it: prefer pages that are sparse *and* old.

The refinement rests on one assumption worth stating explicitly: **temporal update locality** — bytes written together tend to be rewritten together, and soon, or else not for a long time.

It transfers from the workload to pages mechanically, with no appeal to spatial locality in the logical data structures: a data page’s residents are exactly the content that one flush wrote together, so if the application is still working on whatever produced them, more of its residents will be shadowed shortly. Such a “hot” page gets sparser without help, and cleaning it early wastes writes on relocating survivors that are about to die anyway; a page whose live fraction has been stable for a long time is “cold”, and its survivors are the ones worth paying to move.

Co-written-implies-co-updated is the empirically load-bearing assumption behind LFS’s cost-benefit cleaner, and it **fails gracefully**: under a workload with no locality at all — uniform random updates — the age term carries no information and the policy degrades to plain greedy, which is the right fallback anyway.

22.4.1 Scoring a data page

Take LFS’s cost-benefit ratio unchanged, with $u = \text{coverage}[p] / \text{capacity}$ the live fraction and $\text{age} = \text{current_epoch} - \text{page_epoch}[p]$:

$$\text{score}(p) = (1 - u) \cdot \text{age} / (1 + u)$$

The numerator is what the eviction frees times how long it has been stable; the denominator is what it costs — one page read plus u of a page written back.

Evaluating it without scanning every page. The bucket gives u only to within $1/B$ and says nothing at all about age, so the next victim is the highest-scoring of K pages sampled from the sparsest non-empty bucket, for a small constant K — eight, say.

That keeps selection $O(1)$ and gets the age term back, and it is robust in the way that matters: the sampled set is already known to be sparse, so the worst outcome of a bad sample is cleaning a page slightly less old than the best one in the same bucket. Rotating the start of the sampling window across flushes stops the same unlucky prefix from being examined forever.

22.4.2 Victims are pulled one at a time

A victim is taken only when all of its survivors have somewhere to go in this flush. A victim is all-or-nothing, since relocating only some of a page’s survivors reclaims nothing, so the victim is the unit of choice and the room is the only constraint. In free filling that means the page’s room holds all of them; on a budgeted page, one survivor may be [cut at the boundary](#), its tail leading the next page — never on the budget’s last page, where the tail would have nowhere to go. Free filling makes one exception once nothing whole fits: [moving part of a victim](#) reclaims nothing yet, but in a page that is written anyway it costs nothing either, and it leaves the victim sparser.

A marginal victim frees $(1 - u_v) \cdot C$ bytes and writes $u_v \cdot C$. The $1 - u_v$ is not the victim’s empty fraction claimed as a gain; that space is unusable already and would be released regardless. It is what remains after paying for the survivors: all C of the victim’s bytes come back, and $u_v \cdot C$ of them are spent again at once, because the survivors take up as much space in their new page as they did in the victim. The two-epoch quarantine delays the release and the consumption alike, so it changes the timing and nothing else.

Sparsest first is exactly right, since the marginal ratio $(1 - u_v) / u_v$ falls as u_v rises, and the churn floor then reads $u_v \leq 1 / (1 + \lambda)$.

A free sink changes the arithmetic, not the order. A survivor landing in room that a page being written anyway would otherwise waste consumes nothing, so the victim's net gain is the whole C and its ratio $1 / u_v$, above 1 for every u_v : free filling takes any victim that fits, and the churn floor does not apply to it. That is the whole of the distinction between [the two forms](#).

```
impl Evacuation {
    /// The victim whose survivors fit `room` in page `filling`, evacuated, or None: in
    /// compaction mode the tail if it is a data page above `filling` and fits, otherwise
    /// the best-scoring sampled victim that fits. A victim that would overshoot is passed
    /// over for one that fits, never cut.
    fn take_fitting(&mut self, room: u32, filling: PageNumber) -> Option<Survivors> {
        // `tail()` is the highest non-journal live page, and None outside compaction mode.
        let tail = compaction.tail().filter(|&p| is_data(p) && p > filling);
        if let Some(top) = tail.filter(|&p| coverage[p] <= room) {
            return Some(self.evacuate(top));
        }
        let bucket = self.buckets.lowest_non_empty()?;
        let v = bucket.sample(K).filter(|&p| coverage[p] <= room).max_by_key(|&p| score(p))?;
        Some(self.evacuate(v)) // its survivors taken; see "Each fragment is stated once"
    }

    /// What one budgeted page would hold and reclaim, without taking anything: whole
    /// victims by best fit – the tail first, in compaction mode, whatever its fill – the
    /// churn floor applied to each but the tail, then, unless this is the budget's last
    /// page, one more victim cut at the boundary if the page would otherwise close more
    /// than  $\theta$  empty.
    fn offer_page(&self) -> Option<Offer> { ... }
}
```

A caveat on the arithmetic rather than on the policy: coverage counts [aliased bytes once per claimant](#), so u_v over-states both what an evacuation writes and what it reclaims, and the ratio above is a lower bound wherever aliasing is present.

22.5 Victim selection for address-table pages

Table pages are drained by the page rewrite, victim by victim, as data pages are evacuated, and the rotating window restates a key range for density's and defragmentation's sake in whatever room is left; [why the work divides that way](#) is above.

22.5.1 The page rewrite

Victims come from the address-table buckets — sparsest first, K sampled and scored as [data pages are](#), or the tail first [in compaction mode](#) — and a victim is taken only where all of its restatements fit: in the room of the cut's last page, which costs nothing, or on a budgeted leaf. Compaction mode's tail is the exception: it is taken whatever its size, and restatements that do not fit one leaf spill into the cut's next. Executing one is [rewrite_page](#): decode the victim, take the fragments its live statements still own so that the cut states them again, and record any anchor or grow witness that lived there for the cut to replace.

Its cost is those restatements' encoded bytes, estimated from the victim's coverage while it is only being offered, and known exactly once the cut has derived them — possibly less, since they merge with whatever else the flush states nearby; its benefit is the whole victim. So it is priced exactly as evacuation is, $(C - \text{written}) / \text{written}$, and the two compete in [the budget loop](#) on equal terms. [Coverage charges an inline payload per byte to the table page that carries it](#), so the estimate includes payloads without special care — which matters, since a page of 64-byte inlines is ~90 % payload and would look ten times sparser than it is if only framing counted.

A victim with children — an interior page — is rewritten the same way, and its `child_refs` move with it; see [unlinking](#).

22.5.2 The rotating window

```
/// One step of the rotation. Takes live fragments from the cursor on until `room`
/// bytes of encoding are used, walks on for description defragmentation's sake until
/// `WALK` fragments have been seen, and returns the candidates found.
fn rotating_window(room: u32, dirty: &mut Dirty) -> Vec<DefragCandidate> {
    let start = cursor;
    let mut stop = None; // where the taken range ends
    let mut encoded = 0;
    let mut defrag = KadanePerId::new(); // resets at every id boundary
    for (key, fragment) in fragments.range_wrapping(start).take(WALK) {
        defrag.push(key, fragment); // weights as in `defrag_candidates`
        cursor = key.next();
        if stop.is_some() { continue; }
        // `ZeroByDefault` has nothing to state, and a pending fragment is stated
        // already: neither adds to what the window costs.
        if fragment.statement().is_none() { continue; }
        encoded += re_encoded_size(fragment); // against the previous restatement
        if encoded > room { stop = Some(key); }
    }
    rewrite_key_range(start, stop.unwrap_or(cursor), dirty);
    defrag.finish()
}
```

room is an argument because the room is whatever the cut leaves, which varies from flush to flush; and the function returns what its walk found rather than a score, since nothing chooses between windows — there is only the one. The walk meets fragments the flush has already taken, whose statements the cut has not bound yet, so it weighs those by estimate; that is harmless, since a candidate's gain and age are [recomputed when it is chosen](#).

The cursor is a fragment-map key, (`Id`, `AllocationOffset`), so a window is a plain range of the structure it walks, with no special case at either end. An allocation whose live statements alone outgrow the room is simply resumed mid-allocation by the next window, where a cursor keyed by id alone would never get past it. Anchors need no position in the window: [rewrite_key_range](#) transfers an anchor only when the anchor's page is one the window restates from, which is a question about pages rather than about how much of the id the window covered.

A new session resumes the rotation where the last one left it, or else somewhere new. The cursor lives in memory, so the [consolidator state](#) carries it from one session to the next. Without a usable state, a session that started it at the beginning of the key space every time would, over many short sessions — a command-line tool that a script invokes over and over, say — only ever reach the low ids. So open then seeds it at the first statement of one table page, chosen by the governing header's CRC modulo the number of table pages: deterministic for a given file, so tests reproduce, and spread evenly over pages across sessions, which is spread evenly over statements to within how full the pages are.

22.5.3 Unlinking an emptied page

Nothing waits for a load. With framing, inline payloads, and child references all charged per byte, a table page whose last live content is restated elsewhere reaches zero coverage in the same flush, and [the two-generation quarantine](#) does the rest: fallback at the next commit, writable one commit after that. Its dead bytes then sit in it until it is overwritten, which is a statement about the bytes and not about the page's availability.

One step stands between zero coverage and reuse, and it is structural rather than an accounting overhang: **an emptied table page is still named by its parent**, so it stays reachable from the header, and [I2](#) forbids overwriting until a flush has also unlinked it. When the parent is the header, the flush that emptied the page unlinks it for free, since the header is written last. But when the cut's own fillers empty a leaf B

whose parent is an interior page I, unlinking B would take a fresh copy of I, and a new child reference in a header the cut may already have filled to the byte, so B stays linked — superseded, only not yet reusable — until the next flush takes it as a free victim before its cut.

The reference implementation writes shallow trees — the header is the only interior page until a file has accrued [on the order of a thousand leaves](#) — but it must not rely on that, since another writer may shape the tree differently. So:

- **The page table records each table page's parent**, four bytes a page, which [the tree property](#) makes a single page number.
- **Unlinking rewrites the parent** without the child's reference. At depth 1 the parent is the header, rewritten anyway; deeper, the parent is rewritten like any other table page, its other children carried over, and so is *its* parent, up to the header — the depth - 1 path copy of a [structural update](#).
- **A child reference is charged to its parent's coverage**, and released when the child is unlinked. Without that charge, a parent whose statements had all moved elsewhere would reach zero coverage while still holding the only reference to live children, and reusing it would orphan them — silently, since nothing would reach their statements at the next load.
- **A page rewritten while it has children carries them along**: into the header if its child list has room, which also flattens the tree, and otherwise into the page's replacement, with each moved child's parent entry updated.

Data pages need none of this, having no parent.

The one correctness obligation on any rewrite is [transferring the anchor](#).

22.6 Finding the referrers of a data page

This is an unresolved gap, and it is worth stating plainly rather than leaving implicit.

Rewriting an address-table page is self-describing — decode the victim, and its statements name the (id, offset) keys to look up — which is what lets [the page rewrite](#) drain table pages directly, and why address-table pages need nothing of what follows. A **data** page is opaque bytes with no ids in it, so decoding it tells you nothing, and the fragment map is keyed by (id, offset) rather than by address. Nothing in the design currently gets from a victim data page number to the Ref statements that must be re-pointed.

Four candidate answers, roughly in order of how well they fit:

- **A reverse page → ids index, maintained in memory**. To find the fragments referencing page P, sweep each listed id's range of the fragment map and keep the fragments pointing into P. Cheaper than an exact page → fragments index on both axes: one entry per (page, id) pair rather than per fragment, and insertion can be idempotent with pruning done lazily on use, so fragment destruction does nothing at all on the hot path. It is also **sound under splices**, which is the property that matters: a splice restates a statement with a new offset but the same id and address, so the index stays true through it. Refine it with an offset window per (page, id) — the bytes a page holds from one id span at most one page's worth of that id's offset space — and the sweep becomes proportional to the fragments actually in P rather than to the whole allocation. The window is a hint that is always wide enough and not necessarily tight; see [below](#) for the shape that holds it.

Three rules keep it correct, and the third is the one a splice breaks if it is stated the other way round:

1. **Widen on fragment creation**. Creating a Bytes fragment into P unions its range into the window for (P, id). $O(1)$, on the only path that can invalidate the index.
 2. **Do nothing on fragment destruction**. The index is then a superset, which costs a wasted scan and never a missed referrer.
 3. **Recompute the window only while using it**. A scan that visits (P, id) already learns the true extent of P's fragments in that id, so it can narrow the window then, and drop the entry when it finds none. Narrowing it at any *other* time is unsound: a splice moves bytes to offsets outside the recorded window, and it is rule 1 — not the window — that keeps the entry honest.
- **Make data-page consolidation allocation-driven**, the way defragmentation already is: start from an id whose bytes are spread over sparse pages rather than from the page. Needs no new structure

at all, at the cost of not being able to target a specific page — which does not cover the free-filling form, since that is page-driven by construction.

- **An exact page → fragments index.** 0(fragments in page) and exact, but a second index to maintain on every fragment created or destroyed.
- **LFS-style segment summaries** — a trailer in each data page listing (id, offset, page_offset, size) per resident piece. This is the classic answer, but it does **not** transfer directly here: LFS's summary is sound because a block's back-reference (inode, block#) is immutable, whereas a kladde splice re-points existing bytes to a new offset, so a written-once trailer produces false *negatives* — bytes judged dead that are live, which frees a referenced page. It can be repaired by letting coverage adjudicate: use the trailer only to locate live extents, verify each against the fragment map, and evacuate only if the verified bytes sum to coverage[P]; otherwise skip the victim.

22.6.1 The shape of the index

The first option is the current preference, but not in the shape the bullet sketches.

The index must be keyed by page alone, because the query is “which ids have bytes in P”, and a map keyed by (PageNumber, Id) cannot answer it without scanning every entry it holds. The per-(page, id) window is still the right *payload*; it was the outer key that was wrong.

```
struct ReverseIndex {  
    // Keyed by page, because that is the query. The inner list is kept sorted by id  
    // and is short: nothing below the `Inline` threshold ever reaches a data page,  
    // so a 4 KiB page holds at most ~64 ids and in practice a handful.  
    pages: HashMap<PageNumber, SmallVec<[(Id, Range<AllocationOffset>); 4]>>,  
}
```

22.7 Outer fragmentation

Placement decides whether holes form, truncation returns what reaches the end, and a compaction mode moves what never would.

22.7.1 Lowest page first

Reusable pages are handed out lowest first rather than in the order they became reusable, and the file grows only when none is left. Lowest first keeps new content away from the end of the file, so pages that die there stay dead and the tail drains of its own accord; a first-in-first-out pool would refill the tail as readily as any hole.

The cost is small. The pool is an ordered set — a BTreeSet<PageNumber>, or a bitmap with a cursor on its lowest set bit — rather than a queue, and [the quarantine rotation](#) is unchanged. It is an ordered set rather than a min-heap because it is used from both ends: allocation takes the lowest page, and truncation removes the highest ones, which a min-heap could only discard lazily, once they surfaced as its minimum. Lowest first does give up the contiguity that a run from the end of the file would offer a large write, but contiguity buys nothing: a Ref's payload is [bounded by one page's content](#), so no Ref spans pages, and adjacent pages need exactly as many statements as scattered ones. Journal pages come from the same pool, so the journal, too, starts in the lowest hole rather than at the end.

22.7.2 Truncation

After a commit, the file may shrink to end right after its highest live or fallback page — the [truncation rule](#). It is safe for the same reason reuse is: no header that recovery could choose reaches past that point, and a power cut that loses the truncation only leaves the file longer than necessary. The journal's last page does not count while it is empty, since a reader treats [a journal page past the end of the file](#) as empty too. The new end is always a page boundary, never inside the last page's padding: the file system would rewrite the block holding that page's end — zeroing the rest of it, and on some file systems again when the file grows — and a power cut can corrupt a block while it is being written.

Truncating is a cheap metadata operation, but shrinking and regrowing the file on every flush of a workload that oscillates is not, so the reference implementation truncates only once the reusable tail exceeds a threshold, and at close and open. At close, only fallback pages can keep the file from ending at its last page of content; once [the fsync at open](#) has retired the older world, they go too.

22.7.3 Compaction mode

Placement and truncation return the tail only if the pages there die by themselves, and cold, full pages never do. So while **holes** — pages below the highest live one that are reusable now, not in [quarantine](#) — exceed a share h of the file, the highest non-journal live page is offered to [the budget loop](#) ahead of every other victim, whatever its fill: evacuated if it is a data page, rewritten if it is a table page, with lowest-first placement landing its content in holes.

The mode needs no hysteresis, because switching in and out of it costs nothing. A tail page once moved stays moved, so leaving the mode loses no progress; and near h , where compaction lowers the share of holes and ordinary cleaning raises it again — every sparse page it frees below the tail is a new hole — a mode that alternates flush by flush merely splits the budget between the two debts.

An offer holding the tail passes the churn floor and the fill floor whatever else it holds, and a table tail is taken even when its restatements need more than one leaf, so it is the budget alone that paces the mode. The floors weigh what a page returns to the pool, and a tail victim returns more: a whole page to the file system. For a data page the exemption waives nothing the churn floor would catch, since moving it at fill u writes $u \cdot C$, a ratio of $1 / u$ that passes a floor near 1 even for a full page; the fill floor it can fail, when nothing else fills the page it moves into. A table page's restatements can take more than its coverage, since fragments that shadowing split need a statement each and restated statements lose their neighbours' delta encoding, so a nearly full table page fails the churn floor as well, and may not fit one budgeted leaf. A mass free makes one the tail as a matter of course: the flush that states the frees, and the one after it, cannot reuse the pages the frees release yet, so their leaves — the tombstones, and whatever they state `InLine` — extend the file. Were the tail held to the floors, the mode would stop at the first such page, and the budget loop with it, until the rotating window had drained the page a few statements a flush.

The return is realised only once every page above the victim has gone too, which is why the mode works strictly from the top down; journal pages are skipped, since the next flush moves the journal to the lowest hole by itself.

The tail moves only while a page that is reusable now lies below it. Right after a mass free, the pages it released sit in quarantine for two commits, so the lowest reusable page can lie above the tail, or there can be none; content moved then lands in pages the file grows by, which become the next tail, and the same content moves again. Holes count only reusable pages for a related reason: the pages the last commits released are working space, which the next flushes reuse by themselves, and counting them would keep a file whose flushes rewrite much of it in the mode for good.

Free filling takes part in the mode too, since a tail victim riding in a page the flush writes anyway costs nothing at all: while the mode lasts, each source's [take_fitting](#) offers the highest non-journal live page ahead of the sparsest, whenever that page is of its kind, lies above the page being filled, and its content fits the room. The tail wins that comparison because holes are what the file has too many of: freeing one more page for reuse gains little, while every tail page moved brings a truncation closer.

Finding the highest live page needs no structure of its own: a cached index, lowered past reusable pages as the tail empties and raised when a page above it is written, passes each page once per change of state, which is $O(1)$ amortised.

22.8 Pacing

Each flush consumes at least one reusable page, often a few, so consolidation must reclaim at least that many on average.

The situation where no good victim exists is the situation where pages are largely full — i.e. there is little garbage and the file is close to its live size — so “cleaning cannot keep up” and “the file is honestly this large” coincide, which is the benign coincidence.

The one obligation this leaves is [headroom](#), which is a requirement rather than a policy.

22.9 The bound this buys

Garbage accrues in proportion to write traffic and drains in proportion to consolidation effort, so the steady state is set by the budget: enforcing “consolidate until live fraction $\geq \tau$ ” bounds the file at $\text{live_size} / \tau$ for a chosen τ , at the price of the corresponding cleaning work. The bound is controllable and, in relative terms, independent of file size.

22.10 Constants still to be chosen

- The live-fraction target τ , and the feedback rule that moves [the budget](#) toward it.
- The churn floor λ , near 1 and not the knob that sets τ .
- Defragmentation’s price μ per byte written, and its reserved share of the budget.
- The per-flush page budget’s starting value and its bounds.
- The age weighting, in data-page scoring and in defragmentation’s ranking.
- The look-ahead depth L and the fill floor θ in [packing](#) — the first sets how much slack survives in the common case, the second bounds it in every case.
- The rotating walk’s length $WALK$, in fragments per flush.
- The bucket count, 16 for each page kind.
- The hole share h above which [compaction mode](#) runs, and the reusable tail that triggers truncation.
- The page size, 4 KiB versus 16 KiB, to be measured on macOS/iOS and Linux — a per-file header property either way.

Chapter 23

Consolidator state

What the reference implementation keeps in the [consolidator state](#), how it keeps writing it cheap, and how it tells whether what it finds there is still current.

23.1 Why it exists

Consolidation learns from many flushes, and a session that forgot what it had learned at every close would never get past the start. The damage concentrates in workloads of many short sessions, such as a command-line tool that a script invokes over and over, each time opening a file, flushing once, and exiting:

- [Description defragmentation's age](#) would have to be rebuilt from the pages that hold each allocation's content, which is exact only for content that has never moved. The header rewrites the inline payloads it holds every flush, so every allocation with one there would look freshly written at every open, and in a small file, whose allocations are mostly inline in the header, description defragmentation would practically never run.
- Description defragmentation would lose, at every close, the candidates the latest walk had found, since a candidate is executed a flush after it is found; a session of one flush would find candidates and never execute any.
- The [budget](#) would restart from its default at every open, and never converge on its target fill.
- The [rotating window](#) would restart wherever open seeds it, rather than where the last session left off.

23.2 What it holds

part	what	size
identification	the implementation, and the version of this layout	8 bytes
up_to_date	the epoch of the flush that last brought the state up to date	8 bytes
the budget	the per-flush budget the controller had reached	4 bytes
the window	the fragment-map key at which the latest flush's walk began	an (Id, AllocationOffset)
content ages	for each allocation, the flush that last wrote it	a snapshot, plus a byte or two per id written since

Everything a flush puts in the state is known before its data pages are packed, since it is written together with them: what the flush knew when it began, and what its fold learned. That is why the state records where the latest walk *began*: where a walk ends is decided at the cut, after packing.

23.2.1 Content ages

Content age is kept per allocation, the granularity `last_written` has in memory, because it is the one age the file does not already record.

- The other term of description defragmentation's age is a fragment's, the epoch of the statement that owns it, and every page records its epoch.
- Per page would be cheaper, but it would fail where it matters: the header holds the inline payloads of hot and cold allocations alike, so every allocation with one there would look as young as the youngest.
- Per fragment would cost an entry per statement, to cover a blind spot — a hot tail hiding a cold head in a large allocation — that the fragment term already covers.

Ages are kept as records, each naming an epoch and the ids last written in that flush, sorted and stored as varint deltas, which is a byte or two per id. A flush appends its record, which is one contiguous write; an array indexed by id would cost a statement per entry, since the ids one flush writes are scattered. An id's `last_written` is the highest epoch among the records that name it.

```
/// The content of the consolidator state, in this order.
struct ConsolidatorState {
    tag:      [u8; 8],           // the implementation, and the version of this layout
    up_to_date: Epoch,          // the flush that last brought the state up to date
    budget:   u32,              // the per-flush budget the controller had reached
    window:   (Id, AllocationOffset), // where the latest flush's walk began
    ages:     Vec<AgeRecord>,    // the snapshot, then one record per flush since
}

struct AgeRecord {
    epoch: Epoch,
    ids:   Vec<Id>,              // sorted, and stored as varint deltas
}
```

Once the appended records outgrow the snapshot, a flush rewrites everything as a fresh snapshot, keeping only each id's newest mention, one record per epoch. That keeps the state within about twice its compacted size, and costs at most two bytes rewritten per byte appended. Between snapshots, each appended record is described by a statement of its own, until [description defragmentation](#) merges them, as it would for any allocation written piecemeal.

A snapshot may leave allocations out, and an allocation without a record falls back to the youngest page that holds one of its fragments. That is the data page for bytes a Ref states, and the table page for an Inline's payload or a Zero; ZeroByDefault fragments are in no page and are left out, and an allocation with no other fragment falls back to the youngest page that holds one of its statements. Content is never younger than the page that holds it, so the fallback can only understate an age, which errs toward treating an allocation as recently written. It is exact for content that was written once and never moved. So a snapshot leaves out every allocation for which the fallback gives its `last_written` exactly, and keeps mainly two kinds: allocations whose bytes consolidation has moved since they were written, and those with inline content, which moves whenever its table page is rewritten — every flush, in the header. An allocation it left out and consolidation moves afterwards looks younger than it is until the next snapshot takes it in, which errs the same safe way.

23.2.2 The rotating window's position

The state records where the latest flush's walk began, and open walks that window again, read-only, before the rotation resumes where the walk ends. The walk at open finds again the description defragmentation candidates that the latest walk found, which no flush has executed yet, and hands them to the session's first flush. The previous session's last flush already restated as much of that window as its room allowed, so the first flush's own walk resumes where the one at open ended, as it would have within one session. Without a usable state, open [seeds the position](#) instead, and walks from there all the same.

23.2.3 The budget

The state records the **per-flush budget the controller had reached**, so that a session resumes the controller rather than restarting it.

23.2.4 What stays out

- **Description defragmentation's candidates**, which the walk at open finds again.
- **The eviction clock**, which open seeds from content ages: a fragment the header states starts with as many flushes untouched as its allocation has gone unwritten.
- **Whatever the file already records**: page epochs, each page's fill when it was written (its `content_size`), and every statement's page.

23.3 Writing it

Every flush brings the state up to date, together with its own writes. At the end of its fold, it appends its record and rewrites the fixed fields — `up_to_date` with its own epoch, the budget it is working with, and the position its walk will start from — which costs one contiguous append and one small range, a statement or two. The state is ordinary content, committed with the flush that wrote it, so a crash can never leave it ahead of or behind the file it describes.

The first flush of a file creates the state and names it in the header. So does a flush that finds no state it recognizes as its own: it replaces a foreign one, since the reference implementation has nowhere else to keep its own.

23.4 Checking it

A state is used entry by entry, as far as each entry can be shown current, because a writer that kept the state without maintaining it may have changed the file since.

- **If `up_to_date` is the governing header's epoch, everything in the state is current.** That is the common case, since the reference implementation brings the state up to date in every flush.
- **Otherwise, an allocation's age is current if no statement naming the allocation is newer than `up_to_date`,** since content cannot change without a new statement at the epoch of the flush that changed it. The test is conservative, since a restatement looks like a write; an allocation that fails it falls back to the youngest page that holds one of its fragments, as one without a record does.
- **The window's position and the budget are starting points either way:** at worst the rotation resumes somewhere other than where it would have, and the controller corrects a budget that no longer fits.
- **A state the reference implementation cannot parse is treated as absent,** like another implementation's. Nothing in it can affect content, so the worst a misread state can do is mislead the heuristics.

A flush that brings a stale state up to date rewrites it as a fresh snapshot first. Otherwise the records it could not vouch for would become current by default, the moment its own `up_to_date` covered the flushes they missed.

23.5 What the ripeness draft would add

The [ripeness draft](#) estimates how fast each page still drains from that page's own losses, which a session would otherwise have to observe afresh. It would add, for each page below $1 - \theta$, (`page`, `epoch`, `coverage`, `s0`, `s1`, `rate`, `at`, `fast`), appended as the fold drains pages, and the price κ its controller had reached. A page's entry is current if the page's epoch still matches, and the coverage the page has lost since the entry was written is a loss that happened in the gap.

Chapter 24

Transactions and batches

How application operations are grouped into the [transactions](#) the journal records.

Status: not implemented; the design is settled apart from the explicitly stated open questions.

Transactions are normative — they are the unit of atomicity an application builds on. **Batches are not:** they are an implementation’s mechanism for turning a long run of operations into reasonably sized transactions, and an implementation is free to offer something else or nothing at all. Both exist only to affect how mutations are *persisted*; the accompanying in-memory mutations are always immediate.

24.1 Three mechanisms over one journal

mechanism	effect on the journal	what it is for	when to use
transaction	wraps a sequence of operations in a single journal transaction, appended atomically when the caller commits	correctness: either all or none of the operations are persisted, so a crash can’t expose an invalid intermediate state	whenever intermediate state would be invalid
batch	cuts a long sequence of operations into reasonably sized subsequences, respecting nested transactions, and wraps each in a transaction	optimization: fewer I/O operations, while still bounding the journal’s size and any individual write	when many operations happen at once with valid intermediate states — a large import, a paste
bare operation	recorded as its own transaction	simplicity: changes persist immediately	single mutations, prototyping

Transactions enforce correctness. In complex data types, one semantic operation often produces several journal records — appending to a vector grows it and then writes the element — and interrupting between them leaves an invalid state: uninitialized data, or a leak. The same happens in application code, where a transition between two valid states sometimes passes through invalid ones.

Batches are an optimization, and a necessary one. Kladder’s promise to persist every mutation immediately carries a small unavoidable per-transaction overhead. That is usually worth it, but it adds up when an action triggers a large or unbounded burst — and for such bursts, per-operation durability buys little anyway. It is tempting to wrap the burst in a transaction instead, but **very large transactions perform poorly**, because the whole transaction is held in memory until it ends, and the journal, and therefore the file, must then temporarily grow in proportion to the number and size of the operations. A batch is the middle ground: the backend decides how to divide the sequence, respecting any explicit transaction inside it.

24.1.1 Recommendations

- For most application code, reach for a **batch or a bare operation**. Batches are cheaper for multiple operations, have no I/O overhead over a bare operation even for one, and cost a handful of instructions and no allocation. But remember that **batches delay persistence**: they are only guaranteed committed when ended, so holding one open for a long time defeats the point of immediate persistence.
- Use **transactions** when application logic requires atomicity, and keep them short. Needing a very long transaction almost always means either that the application's state transitions pass through too many invalid intermediate states, or that the wrong data type was chosen — a vector where a rope or an ordered tree belongs. When atomicity is genuinely required for a large mutation, it can usually be imposed at the application level instead: import into a structure that is not yet user-facing, then publish a pointer to it in one operation.
- For a long sequence where *some parts* need atomicity, use a **batch with a few smaller transactions inside**. This is the pattern the design is arranged around.

24.2 Nesting

Transactions and batches nest arbitrarily, so that a container's push can open a transaction without caring whether its caller already did. The backend flattens the nesting, because the journal only deals with a flat sequence of transactions.

↓ outer; inner →	transaction in a ...	batch in a ...
... transaction	only outer transaction survives	only the transaction survives (<i>shouldn't usually occur</i>)
... batch	both survive	only outer batch survives

- **Transaction in a transaction ⇒ outer**. The inner one is meaningless; its atomicity is already enforced. Expected often in composable code.
- **Transaction in a batch ⇒ both**. The inner transaction enforces atomicity, so no cutting point may lie inside it. Expected often.
- **Batch in a batch ⇒ outer**. The inner one is meaningless; deferral is already in effect. Fine, but not useful.
- **Batch in a transaction ⇒ the transaction**. The inner batch is meaningless, since the transaction already forces the operations to commit as one group. A potential code smell: batches are for large or unbounded sequences, transactions should be short, so this combination implies a long chain of state transitions with only invalid intermediate states.

24.3 The buffering state machine

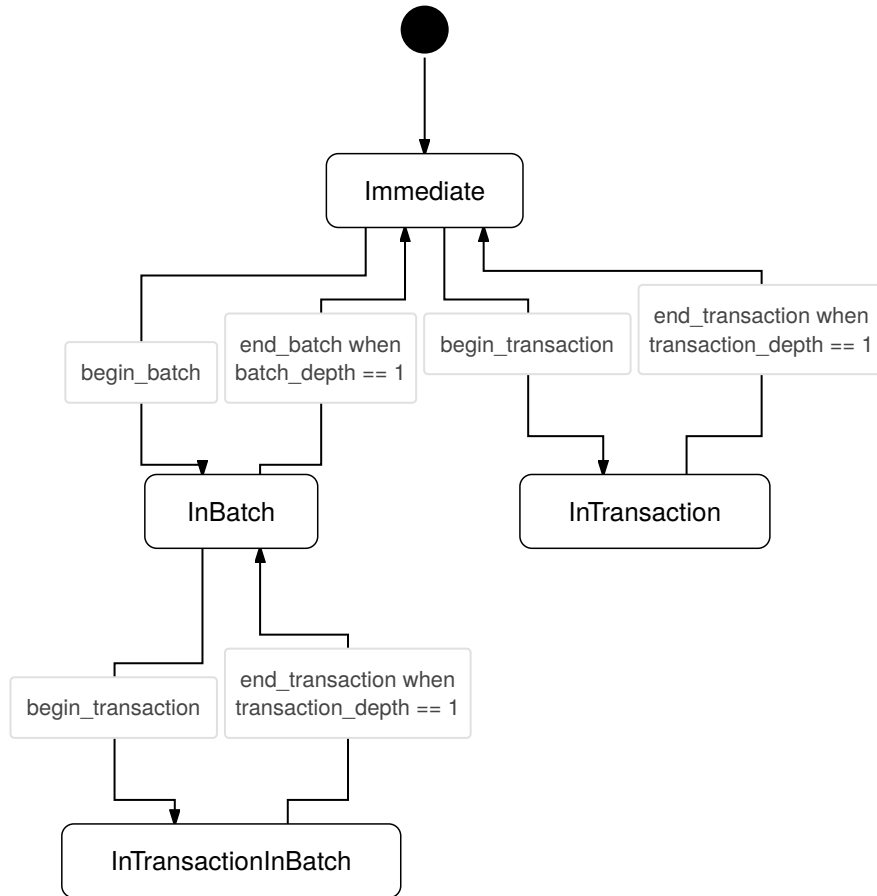
The backend keeps an in-memory buffer ops of serialized operations, not all of which are necessarily in the file yet, and a state:

```
enum TransactionState {
    Immediate,
    InBatch           { batch_depth: NonZero, committed_cursor: usize },
    InTransaction     { transaction_depth: NonZero, batch_depth: u32,
                      committed_cursor: usize },
    InTransactionInBatch { transaction_depth: NonZero, batch_depth: NonZero,
                      committed_cursor: usize, ready_cursor: usize },
}
```

transaction_depth and batch_depth count begins minus ends, with an implied zero in any variant where they are absent. Two cursors, committed_cursor ≤ ready_cursor ≤ ops.len(), divide the operations since the last flush into three spans:

- committed := ops[..committed_cursor] — written to the on-storage journal;

- `ready := ops[committed_cursor..ready_cursor]` — pending operations of the current batch, ready to be appended;
- `transaction := ops[ready_cursor..]` — pending operations of an ongoing transaction.



Why four states and not three. **InTransaction** carries a `batch_depth` that may be positive, so it overlaps with **InTransactionInBatch** in depth alone. What distinguishes them is **which scope was opened first**. **InTransactionInBatch** is *batch outside, transaction inside* — the useful nesting, where ending the transaction returns to **InBatch** and persistence stays deferred. **InTransaction** { `batch_depth > 0` } is *transaction outside, batch inside* — the code-smell nesting, where the batch is subsumed and its depth is tracked only so that its end can be matched.

24.3.1 Transitions in detail

The error cells are all the same mistake: ending one scope while a scope nested inside it is still open — *interleaving* rather than nesting. An implementation should make that unrepresentable at the API level rather than checking for it.

	Immediate	InBatch	InTransaction	InTransactionInBatch
transaction_depth	0 (implicit)	0 (implicit)	> 0	> 0
batch_depth	0 (implicit)	> 0	any	> 0
committed_cursor	<code>ops.len()</code> (implicit)	explicit	explicit	explicit
ready_cursor	<code>ops.len()</code> (implicit)	<code>ops.len()</code> (implicit)	<code>committed_cursor</code> (implicit)	explicit

	Immediate	InBatch	InTransaction	InTransactionInBatch
begin_transaction()	InTransaction { t: 1, b: 0 }	InTransactionInBatch { t: 1, .. }	transaction_depth += 1	transaction_depth += 1
end_transaction()	error	error	-1; at 0: Immediate if batch_depth == 0, else error	-1; at 0: InBatch
begin_batch()	InBatch { b: 1 }	batch_depth += 1	batch_depth += 1	batch_depth += 1
end_batch()	error	-1; at 0: Immediate	batch_depth -= 1	-1; at 0: error

24.4 Flush triggers

Append whole transactions, and flush once the journal segment has outgrown its page budget.

The budget is a trigger, not a capacity. The journal is a [chain of pages](#), so a transaction larger than what is left of the budget — or larger than the whole budget — needs no special case: it is appended whole, spanning as many pages as it needs, and the flush that follows folds it.

The rules, all applied at operation boundaries:

- **Recording an operation in Immediate, and InTransaction → Immediate:** append the operation, or transaction, to the journal as one transaction; then flush if the segment is over budget.
- **Recording an operation in InBatch:** add it to ready; if ready has reached the batch size, append ready as one transaction, then flush if the segment is over budget.
- **InTransactionInBatch → InBatch:** transaction joins ready, and the same check follows as when recording an operation in InBatch — which is why a batch is never cut inside a transaction.
- **InBatch → Immediate:** append ready as one transaction, then flush if the segment is over budget.
- **An explicit flush():** append ready, which lies outside any open transaction, then flush.
- **Opening a file whose recovered journal is non-empty:** flush before the first append, as [the journal requires](#).

The batch size decides how much a batch may hold back from the journal, trading the per-transaction overhead against what an application crash can lose; it is independent of the segment budget.

The budget bounds the file, not memory. An open transaction is held in memory in full, whatever the budget, because nothing can [fold an unfinished transaction](#); a huge transaction costs memory in proportion to its size, and journal pages in proportion too once it is appended.

24.4.1 Consequence for write-phase geometry

Every automatic flush runs right after an append, with ready and transaction empty, so it folds exactly the operations the write-phase geometry has seen, and the geometry can simply be cleared.

An explicit flush() inside a transaction is the exception. The open transaction's operations stay buffered past committed_cursor, so the geometry no longer describes the same point in the sequence as the fold, and it must be **repopulated** from those operations after the flush rather than cleared.

24.5 Checkpoint versus commit

Decided: a checkpoint never splits a commit — see [the journal](#).

A *checkpoint* is a flush triggered by resource pressure, and a *commit* is a transaction boundary the recovered state may snap to. The open question was whether a checkpoint could fold part of an unfinished transaction, to bound the memory the transaction holds. It cannot: the header it committed would describe a state inside the transaction, and undoing that part on recovery would need a mechanism the format does not have. So every flush folds complete transactions only, and the memory an open transaction holds is bounded only by [keeping transactions short](#).

What remains is that application authors need never call `flush()`: the operation that pushes the journal over its budget flushes automatically. Two consequences follow:

Flush needs only shared access to the backend. The auto-checkpoint fires from inside a mutating call, which holds no exclusive reference to the whole structure. This is safe — callers hold ids and offsets, never addresses, and writes resolve locations at write time, so relocation under a live mutation is transparent. Note that it is a *reentrant* call, so no write path may hold an interior borrow across a callback.

The trigger belongs at operation boundaries, not at append time. A check on append can fire between two records of one logical mutation. An operation is the natural atomic unit, being the smallest thing an application author perceives as one change.

24.6 Open questions

- The segment's page budget and the batch size: whether each is fixed or scales with the file, and whether either should adapt to an application whose transactions routinely exceed it.

Chapter 25

Related work

Where kladde’s storage design comes from, and where it deliberately parts company.

None of the load-bearing pieces is novel, which is the point: never-overwrite plus cleaning is Rosenblum’s log-structured file system, the alternating header commit is LMDB’s meta pages, copy-on-write with checksums is ZFS, WAFL and btrfs, and extents-with-escalation is ext4, XFS and NTFS. Their failure modes are understood and bounded, which is exactly what the [bespoke machinery they replaced](#) could not claim.

What *is* different from all of them is the premise: **kladde reads in bulk and looks nothing up on disk**. Almost every divergence below is a consequence of that one fact, and every one of them cuts toward simplicity.

system	shared ideas	where kladde diverges, and why
LSM trees (LevelDB, RocksDB)	The header-as-write-buffer is a memtable; eviction is an L0 flush; ratio-driven consolidation is compaction; cold/hot separation at eviction.	LSMs keep runs sorted and levelled so point lookups work without full residency; kladde is fully resident after the bulk read, so it needs no sort order, no levels, no bloom filters — one tier of dense, unordered leaves.
LFS and SSD FTLs	Never overwrite; live-fraction victim selection; cost-benefit (sparse <i>and</i> old) cleaning.	Their cleaning unit is a large fixed segment; kladde’s is a page, and its leaves can be re-cut freely because the directory keeps no order.
LMDB	Copy-on-write pages, alternating root, page-reuse quarantine by transaction age.	LMDB’s B-tree is sorted because readers descend it in the mapped file; kladde never descends, so it trades away the ordering along with its ~25 % slack and its rebalancing writes. It also needs two fsyncs per commit where kladde needs one , because kladde can let the header lag one write behind and lean on journal replay.
btrfs, ZFS	Copy-on-write everything, checksums everywhere; btrfs <i>inline extents</i> and ZFS <i>bonus buffers</i> store small data directly in metadata — the precedent for Inline as a first-class citizen.	Both maintain on-disk search structures for partial access to data far larger than memory; kladde’s bulk-read premise removes the requirement those structures exist to serve.

system	shared ideas	where kladde diverges, and why
ext4, XFS	Extent-shaped Refs; small-file inlining (<code>inline_data</code>); description debt fought by defragmentation.	In-place journaling and fixed metadata locations rather than copy-on-write shadowing.
MVCC stores (PostgreSQL)	Epoch shadowing is tuple versioning; tombstone lifetime is dead-tuple visibility; consolidation is <code>VACUUM</code> .	MVCC retains versions for concurrent readers; kladde retains them only across the commit boundary, so its vacuum needs no visibility horizon beyond the two-header rule.

One lesson taken from the same literature rather than from a system: PostgreSQL's transaction-id wraparound "freezing" is why [epochs are 64-bit](#). A 32-bit counter would require actively rewriting laggard pages before the counter caught up with them, which is an operational burden PostgreSQL carries only because its format predates the lesson.

Chapter 26

kladde-rs

The Rust implementation of kladde, and the reference implementation of the [specification](#).

Everything here is **specific to Rust** and should be expected to differ in a port. The [file format](#) and the [algorithms](#) are shared; the trait shapes, the mutation-capture mechanism, the macros, and the memory layout are not.

26.1 Two audiences

If you want to use kladde in an application, start with the [tutorial](#). It assumes you know Rust and nothing about kladde, and works up from a first program to writing your own persistable types.

If you want to work on kladde, or port it, read the [specification](#) and [implementation notes](#) first — they carry the parts that are not about Rust — and come back here for the rest.

26.2 What it is

A Rust workspace providing durable data structures: containers and derived types whose mutations are recorded durably as they happen.

```
use kladde::Kladde;
use kladde::Persistable;
use kladde_types::{PersistableString, PersistableVec};

#[derive(Persistable)]
struct Notes {
    title: PersistableString,
    lines: PersistableVec<PersistableString>,
}

let mut notes = Kladde::new(Notes {
    title: PersistableString::from("today"),
    lines: PersistableVec::new(),
});

notes.guard().lines_mut().push(PersistableString::from("wrote a doc"))?;
```

The push updates the in-memory vector and appends the change to the journal before it returns. There is no save call. `Kladde::new` keeps its file in memory, which suits examples and tests; `Kladde::create` and `Kladde::open` do the same on a real file.

26.3 The central problem

Rust has no way to intercept a field assignment.

Every comparable system relies on exactly such a hook: Python has `__setattr__`, Smalltalk has message dispatch, Swift has property wrappers. Those systems make persistence syntactically invisible — `self.foo = bar` just works — because the language hands them a place to stand.

Rust hands them nothing. So `kladde-rs` makes the recording point **explicit and unforgeable** instead: the only mutating API is the one that records, and it is reached through a `Guard`.

Everything idiosyncratic about this layer follows from that one constraint, and it is the single largest thing a port should expect to redesign. A Python implementation should use property hooks and have no guards at all.

26.4 The documents

document	contents
Crate layout	the workspace, the dependency direction, and where the seam falls
Pointers	the two handle types, and the ownership discipline they enforce
Persistable and guards	the central traits, why mutation needs a guard, and why it returns a <code>Result</code>
Containers	how the built-in containers are laid out and why
The derive macro	what it generates, and why guards are generated per type
Freeing	the <code>free</code> hook, and where ownership ends
Transactions	the <code>RAII</code> types, and what happens on unwind
Schema binding	how a Rust type declares its descriptor, and the check at open
The store	the Rust-specific half of the storage layer: interior mutability, I/O, and the in-memory structures
Tutorial	for application authors

26.5 Status

A first prototype of the whole design exists; nothing is frozen. Where the implementation departs from the [implementation notes](#), or found them unclear, `implementation-notes.md` in the `kladde-rs` repository says so.

area	status
Schema descriptors, encoding, fingerprints	implemented and specified
Schema binding to Rust types, fingerprint check at open	implemented
Containers, derive macro, guards	implemented
File format, address table, journal, recovery	implemented
Flush and consolidation	implemented
Transactions and batches	implemented, with the behaviour on unwind
Freeing	implemented as a type-driven hook
Schema evolution	designed, not built

Chapter 27

Tutorial

For people who want to *use* kladde in a Rust application. Assumes ordinary Rust knowledge and no prior exposure to kladde.

1. [Getting started](#) — your first durable value, and the mental model.
2. [The built-in containers](#) — vector, hash map, string, blob.
3. [Deriving your own types](#) — structs and enums.
4. [Compared to serde](#) — when you want kladde and when you want serde, and why they are not competitors.
5. [Durability and flushing](#) — what is guaranteed when, what a crash costs you, and transactions.
6. [Writing a custom `Persistable`](#) (*advanced*) — opaque types, and implementing the trait by hand.

27.1 Is kladde the right tool?

Kladde is a good fit when **all** of these hold:

- you have a long-lived, mutable data structure that must survive a crash;
- you read it far more often than you write it, and you want reads to cost what in-memory reads cost;
- your mutations are small and frequent relative to the size of the whole structure;
- one process owns the file.

It is a poor fit when:

- you want to *exchange* data with other systems — that is `serde`'s job, and kladde's format is a heap, not a document;
- you need concurrent writers, or readers in other processes;
- your data is append-only and never mutated in place — a log file is simpler and faster;
- your working set does not fit in memory. Kladde keeps the whole structure resident by design.

The comparison that most often matters in practice is with “just serialize the whole thing on every change,” which is what most applications do. That is $O(\text{total size})$ per mutation. Kladde is $O(\text{size of the change})$, at the cost of a more constrained type vocabulary.

27.2 Status warning

Kladde is early: the file format is a draft, and a file written today may not open with a later version.

Chapter 28

Getting started

28.1 The mental model

A durable data structure lives in two places at once.

In memory, it is an ordinary Rust value: a `PersistableVec<T>` really does hold a `Vec<T>`, and reading from it is a plain memory access. Nothing about reading touches the file.

In the file, it is a compact binary representation that lags slightly behind, plus a *journal* of changes not yet folded into it. A mutation appends to the journal and then updates the in-memory value, before it returns.

So: reads are as fast as the plain in-memory equivalent, writes cost one small append, and the bulk representation is brought up to date periodically rather than on every change.

The name comes from this shape. A *Kladde* is a merchant's rough day-book — transactions scribbled down in order as they happen — later transcribed into the clean main ledger.

28.2 Your first durable value

```
use kladde::{Kladde, Persistable};
use kladde_types::{PersistableString, PersistableVec};

#[derive(Persistable)]
struct Journal {
    owner: PersistableString,
    entries: PersistableVec<PersistableString>,
}

fn main() -> kladde::Result<()> {
    let path = std::env::temp_dir().join("getting-started.kladde");
    let mut journal = Kladde::create(
        &path,
        Journal {
            owner: PersistableString::from("ada"),
            entries: PersistableVec::new(),
        },
    )?;

    journal
        .guard()
        .entries_mut()
        .push(PersistableString::from("first entry"))?;
```

```
println!("{}", journal.get().entries.len());
journal.close()?;

let journal = Kladde::<Journal>::open(&path)?;
assert_eq!(journal.get().entries.len(), 1);
Ok(())
}
```

Four things are worth noticing.

Kladde<T> is the root. It pairs your root value with the file that backs it. Everything reachable from it is durable; a value you construct outside it is not, until you store it into something that is.

Reading goes through get(), which hands you &T — the plain value. From there you use ordinary methods: len(), iter(), indexing. No guard, no file access, no cost beyond the read itself.

Writing goes through guard(), which hands you a JournalGuard. Every field of a derived type gets a _mut() accessor on its guard, which reborrows the same backend one level deeper. So journal.guard().entries_mut() is a PersistableVecGuard, and push on it both records the change and mutates the vector. It returns a Result, because recording is I/O.

open reads it all back. close folds the journal into the file first, but it does not have to: a file closed by a crash opens just the same, with every mutation that returned Ok.

Kladde::new(value) does the same as create with a file that lives only in memory, which is handy for tests and for trying things out.

28.3 Why the guard exists

In Python, journal.entries.append(x) could persist automatically, because Python lets a library intercept attribute access. Rust does not. There is no hook that fires when you assign to a field.

The guard is the substitute. It is an RAII token — exactly the relationship MutexGuard has to Mutex — that proves you have exclusive, recording access, and it carries two things a plain &mut could not: the backend to record into, and the *location* in the file that this particular field occupies.

This is why you cannot mutate a durable value through a plain &mut. There is no such method. Persistence is not something you can forget, because the only mutating API is the one that records.

28.4 Nesting

Guards compose to arbitrary depth, and each level reborrows the same backend:

```
let mut guard = journal.guard();
let mut entries = guard.entries_mut();
entries.push(PersistableString::from("second"))?;
entries.push(PersistableString::from("third"))?;
```

Or in one chain, when you only need one mutation:

```
journal.guard().owner_mut().set("grace")?;
```

Each _mut() borrows the guard it is called on, so a guard hands out one field's guard at a time. To mutate several fields together, parts() hands out a guard for every field at once, in a struct named after yours:

```
let mut guard = journal.guard();
let JournalParts { mut owner, mut entries } = guard.parts();
owner.set("grace")?;
entries.push(PersistableString::from("handed over"))?;
```

Hold a guard only as long as you need it. It borrows the root, so nothing else can touch the structure while it is alive.

Chapter 29

The built-in containers

kladde-types provides four durable containers. They are hand-implemented against the persistence layer rather than derived, the same way std's collections hand-write raw pointer manipulation internally.

29.1 A default library, not a privileged one

kladde-types is not part of the machinery, and nothing depends on it. It is a collection of types most applications turn out to want, offered so that you do not have to write them yourself.

Everything in it is built on the same public Persistable/Guard surface available to any crate. There is no internal kladde magic here that a third-party library — or your own application — could not use equally well, and third-party libraries of general-purpose durable types are welcome.

29.2 PersistableVec<T>

A growable sequence. T must be Persistable, and nothing more — no Clone, no Serialize.

```
let mut guard = journal.guard();
let mut v = guard.entries_mut();
v.push(PersistableString::from("a"))?;
v.insert(0, PersistableString::from("b"))?;
v.get_mut(1).unwrap().push_str("c"); // the guard of one element
let first = v.remove(0)?; // yours now, allocations and all
v.push(first)?; // so it moves without copying its bytes
v.delete(0)?; // removes and frees
```

Reading needs no guard: PersistableVec<T> dereferences to [T], so len, indexing and iter work on the in-memory elements.

Layout. A pointer inline, plus a separate allocation holding a dense array of fixed-size element slots — the same shape as Vec<T>'s in memory. The length is that allocation's size divided by the element size, so it is never stored separately.

Cost. A push grows the allocation and writes the new slot, a pop shrinks it, and an insertion or removal in the middle is one splice, which shifts the tail exactly as Vec does.

Removal. remove and pop hand the element back with everything it owns, so you can store it elsewhere; delete and clear free it. An element you take out and then drop leaks its allocations in the file, the way a value passed to std::mem::forget leaks its memory.

29.3 PersistableString

Growable text.

```
let mut owner = guard.owner_mut();
owner.set("ada"?;
owner.push_str(" lovelace"?;
```

It is a thin wrapper around `PersistableVec<u8>`, which is more interesting than it sounds. Plain `String` deliberately has **no** `Persistable` implementation, and that absence is the enforcement mechanism: a struct field typed `String` simply fails to compile, rather than silently persisting nothing. `PersistableString` exists because a durable string needs a field of its own to remember which allocation holds its bytes, and `String` has nowhere to put one.

So this does not compile, and that is the point:

```
#[derive(Persistable)]
struct Journal {
    owner: String, // the trait bound `String: Persistable` is not satisfied
}
```

29.4 PersistableHashMap<K, V>

A key-value map. `K: Eq + Hash + Persistable, V: Persistable`.

```
let mut contacts = guard.contacts_mut();
let ada = PersistableString::from("ada");
contacts.insert(PersistableString::from("ada"), contact)?;
contacts.get_mut(&ada).unwrap().email_mut().set("ada@example.com"?;
contacts.delete(&ada)?; // removes and frees; `remove` hands the value back
```

Reading — `get`, `contains_key`, `iter`, `len` — needs no guard. Inserting under a key that is already present overwrites its value in place and returns the old one, as `HashMap::insert` does.

Layout. An array of fixed-size slots, each a one-byte liveness tag followed by a `(K, V)` pair. The on-disk form does *not* support key lookup at all — that is what the in-memory map is for — so it needs no hashing structure on disk.

Removal is tombstoning: the tag is cleared in place and nothing else moves, and a later insertion reuses the slot. This is what lets each key be stored exactly once in memory, and is why `K: Clone` is not required.

Keys are immutable. No guard is ever handed out for a key; `Persistable` on `K` governs only how its bytes are read and written.

29.5 PersistableBlob<T>

An escape hatch for foreign types.

```
#[derive(Persistable)]
struct Config {
    window: PersistableBlob<WindowGeometry>, // a foreign serde type
}
```

Wraps any `T: Serialize + DeserializeOwned + Default` as an opaque, serialized payload with its own allocation. Use it when a type is not yours to change and cannot get its own `Persistable` implementation.

It is deliberately the worst option. A blob is rewritten in full on every change, which forfeits kladde's central advantage; it is opaque to [tooling](#); and it drags in a serialization dependency. Prefer a derived type or a hand-written implementation whenever you can. It is gated behind `kladde-types'` `serde` feature — the only thing in the workspace that needs `serde` at all.

29.6 What is missing

A **rope**, for large text with efficient middle-insertion, is designed but not built.

Tuples of up to twelve components are persistable, laid out like a tuple struct, and their guard's parts() hands out a guard per component. There is no durable Option or Box as such — a derived enum covers Option's role.

29.7 Choosing

you have	use
a sequence	PersistableVec<T>
text	PersistableString
a lookup table	PersistableHashMap<K, V>
a few values that belong together	a tuple, or a derived struct
your own struct or enum	<code>#[derive(Persistable)]</code>
a foreign type you cannot change	PersistableBlob<T>, reluctantly
a foreign type whose layout you know	a hand-written impl

Chapter 30

Deriving your own types

`#[derive(Persistable)]` turns your own struct or enum into a durable type, as long as every persisted field is itself `Persistable`.

30.1 What to import

Two crates: `kladde` for the machinery, `kladde-types` for the containers. Neither is on `crates.io` yet, so depend on the repository:

```
[dependencies]
kladde = { git = "https://github.com/kladde-dev/kladde-rs" }
kladde-types = { git = "https://github.com/kladde-dev/kladde-rs" }
```

You do **not** need `kladde-derive` or `kladde-persist`. The macro is re-exported from `kladde` behind its `derive` feature, which is on by default, and generated code is rooted at `::kladde`, which re-exports every item it names. That is deliberate: if you had to add `kladde-persist` yourself, a version disagreeing with `kladde`'s would give you two distinct copies of the `Persistable` trait and an error message that prints the expected and found types identically.

Here is a complete program:

```
use kladde::{Kladde, Persistable};
use kladde_types::{PersistableString, PersistableVec};

#[derive(Persistable)]
enum PhoneNumber {
    Mobile(PersistableString),
    Landline(PersistableString),
}

#[derive(Persistable)]
struct Contact {
    email: PersistableString,
    phones: PersistableVec<PhoneNumber>,
    starred: bool,
}

fn main() -> kladde::Result<()> {
    let mut contact = Kladde::new(Contact {
        email: PersistableString::from("ada@example.com"),
        phones: PersistableVec::new(),
        starred: false,
    });
}
```

```

let mut guard = contact.guard();
guard.starred_mut().set(true)?;
guard
    .phones_mut()
    .push(PhoneNumber::Mobile(PersistableString::from("555-0100")))?;
Ok(())
}

```

The `Persistable` in that use is the derive *macro*. It shares its name with the `Persistable` *trait*, which is legal and deliberate — they live in different namespaces, exactly as `serde`’s `Serialize` does. If you need to name the trait explicitly in the same file, import it under an alias.

30.2 Structs

The macro generates a guard type — `ContactGuard` — with a `_mut()` accessor per field, and a `set` that replaces the whole value:

```

let mut contact = /* ... a ContactGuard ... */;
contact.starred_mut().set(true)?;
contact.phones_mut().push(number)?;

```

It also generates a `ContactParts` struct, and a `parts()` method on the guard that returns the guards of all fields at once, for when you need more than one at a time. A tuple struct’s accessors are named by position — `field_0_mut()`, `field_1_mut()` — and its `parts()` returns a tuple struct.

A struct owns no allocation of its own. Its fields are laid out consecutively in whatever allocation contains it, so its inline size is just the sum of its fields’ inline sizes, and each field’s offset is the sum of the earlier ones’. Those offsets are compile-time constants, which is why field access costs nothing at runtime.

That includes primitive fields. `bool`, `i32`, `char` and friends all implement `Persistable`, specifically so the macro can treat every field uniformly rather than special-casing leaves. So do tuples of up to twelve `Persistable` types, laid out like a tuple struct.

30.3 Newtypes

A single-field struct marked `#[kladde(transparent)]` is persisted exactly as its field — the same bytes and the same [fingerprint](#) — so wrapping a field in a newtype changes nothing in the file:

```

#[derive(Persistable)]
#[kladde(transparent)]
struct Meters(u32);

fn walk(distance: &mut Kladde<Meters>) -> kladde::Result<()> {
    distance.guard().get_mut().set(42)
}

```

Its guard’s `get_mut()` returns the field’s guard.

30.4 Enums

An enum is a discriminant followed by the selected variant’s fields, each variant laid out like a struct in its own right. Its inline size is the discriminant width plus the largest variant’s field sum — so, like a struct, it owns no allocation. Positional fields are named by their position: `Mobile(PersistableString)` has a field called `"0"`.

To mutate the current variant in place, match on the guard’s `parts()`. It hands out the guards of the current variant’s fields, in an enum named after yours, `PhoneNumberParts`, with the same variants:

```

use kladde::{Kladde, Persistable};
use kladde_types::PersistableString;

#[derive(Persistable)]
enum PhoneNumber {
    Mobile(PersistableString),
    Landline(PersistableString),
}

fn main() -> kladde::Result<()> {
    let mut number = Kladde::new(PhoneNumber::Mobile(PersistableString::from("555-01")));

    let mut guard = number.guard();
    match guard.parts() {
        PhoneNumberParts::Mobile(mut digits) => digits.push_str("00"),
        PhoneNumberParts::Landline(mut digits) => digits.set("555-0199"),
    }

    // To switch to another variant, replace the whole value.
    guard.set(PhoneNumber::Landline(PersistableString::from("555-0123")))?;
    assert!(matches!(number.get(), PhoneNumber::Landline(digits) if digits == "555-0123"));
    Ok(())
}

```

Each field guard writes inside its field, so the variant stays what it was. `set` replaces the whole value, which is how the variant changes: it stores the new value, then frees whatever the old one owned. `parts()` borrows the guard, so the compiler stops you from calling `set` while you still hold a field's guard.

The discriminant is Rust's own: the one you write (`Mobile = 1`), or else one more than the previous variant's, counting from 0. On file it takes the smallest of 1, 2, 4 or 8 bytes that holds the largest discriminant, unless an integer `#[repr]` such as `#[repr(u16)]` fixes the width; negative discriminants and signed or pointer-sized reprs are not supported. Explicit values on variants with fields need a `#[repr]` anyway, as Rust requires, and this one also makes the discriminant 4 bytes wide where 1 would do:

```

#[derive(Persistable)]
#[repr(u32)]
enum PhoneNumber {
    Mobile(PersistableString) = 1,
    Landline(PersistableString) = 2,
}

```

30.5 Generics

Type parameters work, and each gets a `Persistable` bound:

```

#[derive(Persistable)]
struct Labelled<T> {
    label: PersistableString,
    value: T,
}

```

Lifetime and `const` parameters are not supported.

30.6 What the derive needs

Every persisted field must be `Persistable`. In practice that means:

- primitives — fine as-is;

- kladde containers — fine as-is;
- your own derived types — fine, and they nest arbitrarily;
- `String`, `Vec<T>`, `HashMap<K, V>` — **not** `Persistable`. Use the kladde equivalents. This is deliberate: the compile error is the mechanism that stops you from silently persisting nothing.

30.7 Non-persisted fields

Sometimes a struct has a field that should not be stored — a cache, a handle, something derived. The `derive` macro has no attribute for this yet; the workaround is a [hand-written implementation](#) that reads and writes only the fields you want persisted.

Doing so does **not** make your type opaque to the format. A hand-written implementation that writes exactly the bytes of a struct with two fields *has* that struct's [descriptor](#). The presence of a third, non-persisted field in your Rust source is invisible.

30.8 Layout stability

A few things you should know before you ship a file format built on a derived type.

Field order is layout. Reordering fields in the source changes every subsequent field's offset, and therefore changes the type's [fingerprint](#), and `Kladde::open` refuses a file whose fingerprint differs from your type's. Once [evolution](#) lands, reordering will be non-breaking, because fields are identified by name.

Variant order is layout only through implicit discriminants. A variant without an explicit discriminant takes the next number, so inserting or reordering variants rennumbers the ones after them, and changes the fingerprint. With [explicit discriminants](#), reordering changes nothing, since enum variants are canonicalized by discriminant value.

The discriminant's width is layout too, unless you pin it. Without a `#[repr]`, adding a 257th variant, or giving one a discriminant of 256 or more, widens the discriminant of every value of the type, which moves every field behind it in every struct that contains it, and changes the fingerprint. An integer `#[repr]`, such as `#[repr(u16)]`, fixes the width up front.

Renaming a type is free. A struct's or enum's own name is not fingerprinted. Renaming a *field* or a *variant* is not free, since those are the identities evolution reconciles by.

Chapter 31

Compared to serde

The most common question about kladde is how it relates to serde. The short answer: they solve different problems and compose badly, and if you are choosing between them you are probably choosing between “exchange” and “storage.”

31.1 The core difference

Serde serializes a value. You call `to_vec(&value)`, you get bytes, you write them somewhere. The cost is proportional to the size of the whole value, every time.

Kladde stores a structure and records changes to it. You mutate it, and the cost is proportional to the size of the change.

For a 100 MB structure with a one-field edit:

	serde	kladde
cost of one small edit	rewrite 100 MB	append tens of bytes
cost of a read	free (already in memory)	free (already in memory)
cost of opening	parse 100 MB	read the compact form, replay the journal
crash mid-edit	lose everything since the last write	lose nothing acknowledged

The interesting column is the first one, and it is the whole reason kladde exists.

31.2 What serde does better

Interchange. Serde targets many formats — JSON, MessagePack, CBOR, YAML — and other systems can read them. A kladde file is a heap with an embedded schema. Another kladde implementation can read it; nothing else can.

Type coverage. Serde works on essentially any Rust type, including `String`, `Vec<T>`, `HashMap`, and third-party types with derives. Kladde needs its own container types.

Maturity. Serde is one of the most-used crates in the ecosystem. Kladde is early.

Zero setup. `#[derive(Serialize)]` and you are done. Kladde asks you to restructure your types around durable containers.

31.3 What kladde does better

Incremental durability. The one thing serde structurally cannot do. There is no serde-based design where a small edit to a large structure costs less than rewriting the structure, short of building an incre-

mental format yourself — which is what kladde is.

Crash safety with no explicit save. With *serde* you choose when to write, and everything since the last write is at risk. With kladde, a mutation that has returned is durable.

In-place mutation of stored data. A kladde file is randomly addressable and mutable. A *serde* document is a byte stream you rewrite.

Language-independent tooling. A kladde file carries its own [schema](#), so a generic tool can inspect and even compact it. A postcard blob is opaque without the writing program.

31.4 They are not exclusive

`PersistableBlob<T>` wraps any *serde* type as an opaque payload inside a kladde structure.

This is the right escape hatch for a foreign type you cannot change, and the wrong default for anything else: a blob is rewritten in full on every change, so it forfeits exactly the property you came for. Use it at the leaves, for things that change rarely.

31.5 Choosing

Use ***serde*** when the data leaves your process: config files people edit, API payloads, anything another program reads.

Use ***kladde*** when the data *is* your application's state: it lives across runs, it is large relative to each change, and losing it to a crash is unacceptable.

Use **both** when your state is mostly kladde-shaped but contains a few foreign leaves.

31.6 An honest caveat

Serde is a safe default and kladde is not yet. If your structure is small enough that rewriting it is cheap — under a few megabytes, say, edited a few times a second — *serde* is simpler, better tested, and fast enough. Kladde starts winning when rewriting starts hurting.

Chapter 32

Durability and flushing

What is guaranteed when, and what a crash actually costs you.

32.1 The two-stage model

A mutation does two things before it returns.

1. It appends records describing the byte-level change to the **journal**.
2. It updates the in-memory value.

Separately and later, the journal is **folded**: the records are applied to the file's compact form, written to fresh pages, and a new journal segment is started.

Durability comes from the journal, not from the fold. A mutation that has returned survives an application crash whether or not a fold has happened, because its record is in the file — the operating system writes it out even if your process dies. A *power* cut is the case where the fold matters; see [below](#).

32.2 What a crash costs

After an application crash: nothing that returned 0k. If push returned 0k, the element is in the file, and `Kladde::open` finds it.

After a power cut: at most what came after the last flush. Journal appends are not forced to disk one by one, so a power cut can take the ones the operating system had not written out yet. Everything folded by a completed flush survives, and what survives of the rest is always a prefix: never a later mutation without the earlier ones.

A partial mutation is never visible. Each mutation, and each transaction, is recorded so that recovery replays all of it or none of it, and the built-in containers order their records so that every state in between is valid anyway.

A crash during a fold costs the fold, not the data. The fold never overwrites live data in place; it writes to unused space and commits with one small atomic write. A crash before that commit leaves the old state intact and the half-written new data inert, and the journal still holds everything, so the next open simply redoes the fold.

32.3 Flushing

You do not normally have to think about this.

Flushing is **automatic**: once the journal outgrows its budget, the mutation that pushed it over folds it before returning. Application authors mutate state and never call anything.

An explicit `flush()` remains available for the cases where you want the fold to happen now — before a long idle period, say, since a flush is also what makes everything before it survive a power cut. `close()`

flushes and then truncates the file after its last live page, which does not shrink it to its live pages: free pages below that one stay in the file. They go only as later flushes consolidate, each moving a bounded amount of live content down; space freed by one flush becomes available from the flush after it on. To reclaim more of it before closing, call `flush()` a few times first; a freshly created file, for instance, needs one or two:

```
journal.flush()?;
journal.flush()?;
journal.close()?;
```

What flushing is *not* for: making your data survive an application crash. The journal already did that.

32.4 When the disk fails

A failed `fsync` means the operating system may have dropped writes it had already accepted, so `kladde` **poisons** the `Kladde`: every later call returns `Error::Poisoned`. Drop it and open the file again; it then holds every mutation that returned `Ok`.

32.5 Transactions

Every mutation is atomic on its own. When several belong together — money leaving one account must arrive in another — make them one transaction, and a crash keeps all of them or none:

```
use kladde::{Kladde, Persistable};

#[derive(Persistable)]
struct Accounts {
    checking: i64,
    savings: i64,
}

fn main() -> kladde::Result<()> {
    let mut accounts = Kladde::new(Accounts { checking: 100, savings: 0 });

    let mut tx = accounts.transaction();
    let mut guard = tx.guard();
    let AccountsParts { mut checking, mut savings } = guard.parts();
    checking.set(70)?;
    savings.set(30)?;
    tx.commit()?;

    assert_eq!(accounts.get().savings, 30);
    Ok(())
}
```

Each mutation still changes the in-memory value when it is made, and `commit` appends them to the journal together. Dropping a transaction commits it too, unless a panic is unwinding: then its mutations are discarded, since they may be half a change, and the `Kladde` is poisoned, since the in-memory value already holds them. There is no way to cancel a transaction, for the same reason. [Transactions](#) has the details.

32.6 Long runs of mutations

A single transaction is held in memory until it ends, because the fold only ever folds complete transactions. So keep transactions short, and use a batch for a long run of mutations that are each valid on their own:

```
let mut batch = log.batch();
for reading in 0..100_000 {
    batch.guard().push(reading)?;
}
batch.end()?;
```

A batch appends its mutations to the journal in a few large pieces rather than one by one, which saves writes, and the automatic flush can fold between the pieces. It is not a transaction: a crash can keep some pieces and lose the later ones, though never part of a mutation, nor of a transaction opened inside the batch with `batch.transaction()`.

Chapter 33

Writing a custom Persistable

Advanced. Most types should be [derived](#) or built from the [containers](#). Reach for a hand-written implementation when neither fits.

33.1 When you need one

Three cases.

A type with non-persisted fields — a struct carrying a cache or a handle that should not be stored. The derive macro has no attribute for this yet, so you write an implementation that reads and writes only the fields you want.

A layout you control precisely — you know your type's bytes should be, say, a u32 length followed by a payload, and you want that exact layout rather than what the derive would produce.

A container — something with its own allocation and a non-trivial internal structure. This is what the built-in containers do, and it is the hardest case.

If your type is a foreign one you cannot change, you probably want [PersistableBlob<T>](#) instead, and to accept its costs.

33.2 The trait

```
pub trait Persistable<P: PointerRepr = Pointer>: Sized {
    const INLINE_SIZE: usize;

    fn store<B: WriteBackend<Pointer = P>>(
        &mut self, backend: &B, location: Location<P, B::Size>,
    ) -> Result<(), Error>;

    fn load<B: ReadBackend<Pointer = P>>(
        backend: &mut B, location: Location<P, B::Size>,
    ) -> Result<Self, Error>;

    fn free<B: WriteBackend<Pointer = P>>(&mut self, backend: &B) -> Result<(), Error> {
        Ok(())
    }
}
```

Plus a `Guard` type and a `guard()` method for the mutation side, and `describe_local` for the schema.

INLINE_SIZE is how many bytes your value occupies *inline*, in whatever allocation contains it. For a scalar, its own width. For a type that owns a separate allocation, the size of its fixed header — not the size of its contents. This must be a constant, because it is what makes sibling fields' offsets statically computable.

Location is where your value's inline bytes live: an anchor pointer plus a byte offset within it. It is threaded down the same chain that carries the backend.

store takes &mut self, which surprises people. A value that owns an allocation may not have one yet — a `PersistableVec` built by `from_iter` holds real content but no pointer, because nothing gave it a backend — and `store` is where that allocation happens the first time, so it must record the new pointer in `self`.

load takes &mut B, not `&B`. Loads are sequential, so a single exclusive borrow reborrowed down the recursion suffices, and it lets the read path hand out a real seekable cursor.

free releases whatever your value owns; the default, for a type that owns nothing, does nothing.

33.3 What to import

One crate. `kladde` re-exports everything a hand-written impl names, so you never add `kladde-persist` yourself:

```
[dependencies]
kladde = { git = "https://github.com/kladde-dev/kladde-rs" }
```

If you are writing a *library* on top of `kladde-persist` and have no reason to pull the facade in, depend on `kladde-persist` directly and point the macro at it instead — see [the `derive` macro's path resolution](#).

33.4 A complete example

A fixed-size type owning no allocation of its own. This is everything the trait requires: the inline size, a guard, the two halves of the round trip, and the descriptor.

```
use kladde::{
    Error, Field, Guard, Location, Persistable, PointerRepr, ReadBackend, SchemaBuilder,
    TypeDescriptor, WriteBackend,
};
use std::io::Read;

pub struct Rgb {
    r: u8,
    g: u8,
    b: u8,
}

pub struct RgbGuard<'s, B: WriteBackend> {
    inner: &'s mut Rgb,
    backend: &'s B,
    location: Location<B::Pointer, B::Size>,
}

impl<'s, B: WriteBackend> RgbGuard<'s, B> {
    pub fn set(&mut self, mut value: Rgb) -> Result<(), Error> {
        <Rgb as Persistable<B::Pointer>>::store(&mut value, self.backend, self.location)?;
        *self.inner = value;
        Ok(())
    }
}

impl<'s, B: WriteBackend> Guard for RgbGuard<'s, B> {
    type Persistable = Rgb;
    type Backend = B;
    fn as_persistable(&self) -> &Rgb {
        self.inner
    }
}
```

```

    }
    fn as_persistable_mut(&mut self) -> &mut Rgb {
        self.inner
    }
    fn backend(&self) -> &B {
        self.backend
    }
}

impl<P: PointerRepr> Persistable<P> for Rgb {
    const INLINE_SIZE: usize = 3;

    type Guard<'s, B: WriteBackend<Pointer = P>>
        = RgbGuard<'s, B>
    where
        B: 's;

    fn guard<'s, B: WriteBackend<Pointer = P>>(
        &'s mut self,
        backend: &'s B,
        location: Location<P, B::Size>,
    ) -> RgbGuard<'s, B> {
        RgbGuard { inner: self, backend, location }
    }

    fn store<B: WriteBackend<Pointer = P>>(
        &mut self,
        backend: &B,
        location: Location<P, B::Size>,
    ) -> Result<(), Error> {
        backend.write(location.anchor, location.offset, &[self.r, self.g, self.b])
    }

    fn load<B: ReadBackend<Pointer = P>>(
        backend: &mut B,
        location: Location<P, B::Size>,
    ) -> Result<Self, Error> {
        let mut buf = [0u8; 3];
        backend.read_at(location.anchor, location.offset)?.read_exact(&mut buf)?;
        Ok(Rgb { r: buf[0], g: buf[1], b: buf[2] })
    }

    fn describe_local(builder: &mut SchemaBuilder) -> TypeDescriptor {
        TypeDescriptor::Struct {
            name: "Rgb".into(),
            fields: ["r", "g", "b"]
                .into_iter()
                .map(|name| Field {
                    name: name.into(),
                    ty: <u8 as Persistable<P>>::describe(builder),
                })
                .collect(),
        }
    }
}

```

Four things worth noting.

INLINE_SIZE is 3 because store writes exactly three bytes. Every offset computed by a containing struct

depends on that number being right.

The guard is a separate type, generated for you by the derive macro but written out here. It holds the value, the backend and the location, and its mutating methods do both halves in order: record the bytes, then update the in-memory value, so that a failed append leaves the value as it was.

describe_local declares what the bytes are, not what the Rust type is. `Rgb` writes three consecutive `u8`s, so it declares a `Struct` of three `u8` fields — even though the implementation is hand-written. Declare `Opaque` only when the representation genuinely is not decomposable; see [Your descriptor](#).

The impl is generic over `P: PointerRepr`, so `Rgb` works at any pointer width. A type that *holds* a pointer is written for one `P`.

33.5 Owning an allocation

If your type owns content, it holds an `Option<UniquePointer>` — `None` until first stored — and its inline representation is that pointer. The pattern:

- in store, allocate if `None` — `backend.alloc(size)` — and write the content;
- write the pointer itself *last*, since it is what makes the content reachable;
- in free, free what the content owns, then the allocation itself;
- for a whole-value set on its guard, call `kladde::replace`, which stores the new value and then frees the old one, in one transaction.

That ordering is not stylistic. It is the [ordering discipline](#): a crash between the content write and the pointer write must leave a valid, if stale, state, and publishing the pointer first would leave it pointing at content that was never written. The rule generalizes to *prepare new state* → *one publishing write* → *clean up what it replaced*.

33.6 Your descriptor

A hand-written implementation must also declare its [type descriptor](#), and the rule is easy to get wrong:

Declare the descriptor that matches **the bytes you actually read and write**, not the shape of your Rust type.

A struct with a non-persisted cache field declares the struct *without* that field. Declare `Opaque` only when your representation genuinely is not decomposable into fields and variants — a length-prefixed blob, an externally serialized payload. “Hand-written” and “opaque” are different axes, and conflating them makes your type needlessly invisible to [tooling](#).

33.7 The obligations you are taking on

A hand-written implementation is trusted, not checked. You are promising:

- `INLINE_SIZE` matches what store writes and load reads;
- store and load are exact inverses;
- writes within one mutation are ordered so every prefix is valid;
- free releases exactly what the value owns;
- your declared descriptor matches your actual bytes.

None of these are enforced by the compiler. This is the same bargain `std`’s collections make with `unsafe`, and the reason the derive macro is the right default.

Chapter 34

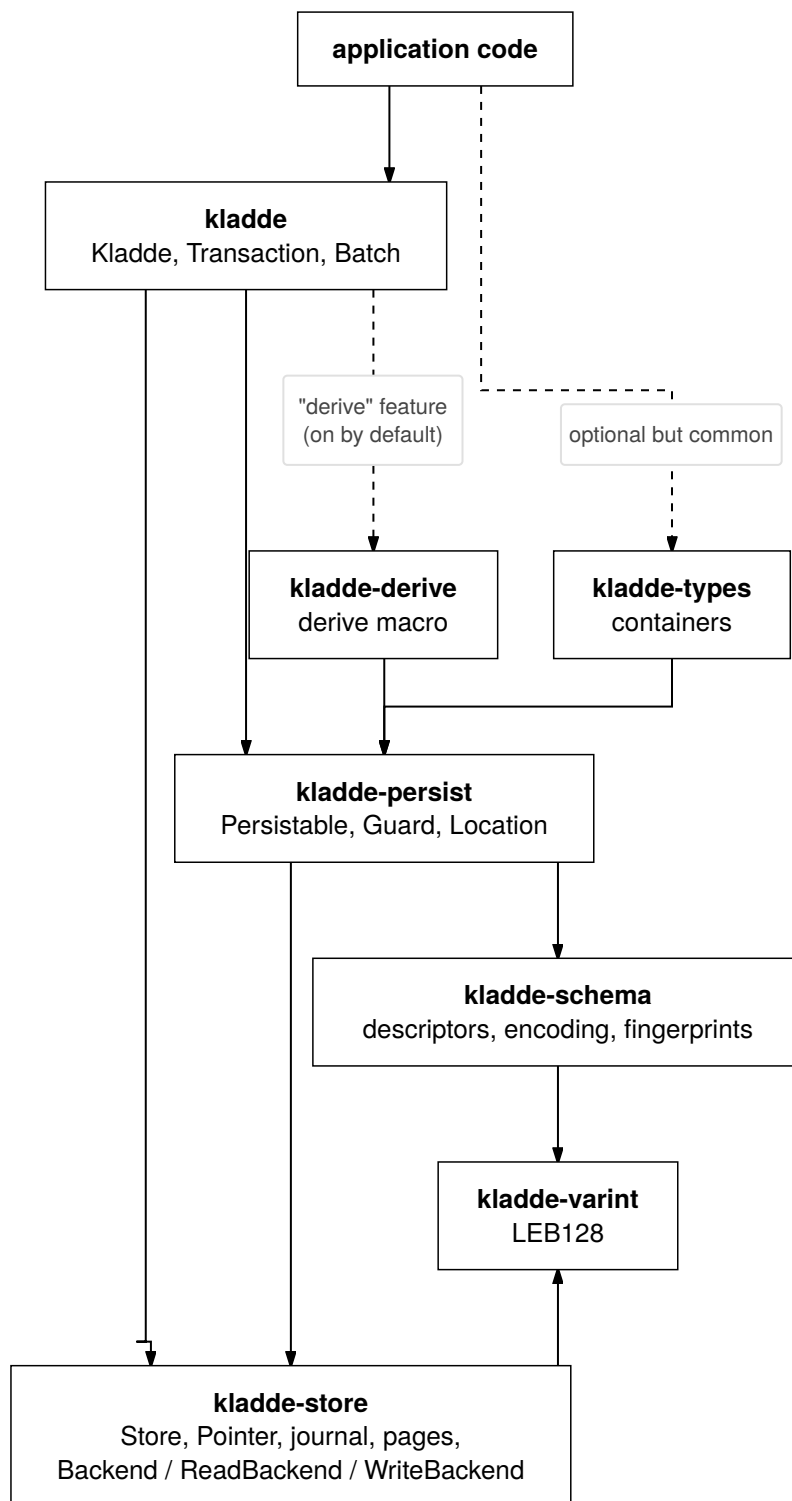
Crate layout

The workspace, what depends on what, and where the language-neutral half ends.

34.1 The crates

crate	role
kladde-store	the storage layer: the file, pages, the address table, the journal, the flush, and consolidation, behind the Backend traits. Type-agnostic; knows nothing about serialization.
kladde-persist	the serialization layer: <code>Persistable</code> , <code>Guard</code> , <code>Location</code> , the scalar guards, and the schema binding.
kladde-schema	type descriptors, their canonical encoding, and fingerprints. Depends on nothing but <code>kladde-varint</code> .
kladde-varint	LEB128 varints.
kladde-types	the built-in containers: vector, hash map, string, blob. The default collection, not a layer — it uses only the public API any library could.
kladde-derive	the <code>#[derive(Persistable)]</code> macro. Reached through <code>kladde</code> 's <code>derive</code> feature; applications do not depend on it directly.
kladde	the application-facing entry point: creating and opening files, the root value, transactions and batches. Re-exports the <code>derive</code> macro and everything a <code>Persistable</code> impl names, so an application needs this crate and nothing below it.

The dependency direction is strictly downward, and `kladde-store` deliberately depends on nothing but `kladde-varint` — it is usable on its own as a persistent store of byte allocations, independently of anything above it.



34.2 The parts

34.2.1 kladde-store — the storage layer

Owns the file. It implements everything [the implementation notes](#) describe, and adds only what Rust needs: the Backend traits the layer above calls, the Store that implements them, and a Storage trait with two implementations, a real file and a byte vector in memory. See [The store](#) for the Rust-specific half.

34.2.2 kladde-persist — the serialization layer

Where types meet bytes. Persistable, its Guard counterpart, Location, and the guards of the scalar types. Knows about the store’s traits; knows nothing about any particular container or derived type.

34.2.3 kladde-schema — type descriptors

The Rust implementation of the [language-independent schema model](#). It depends on neither the store nor Persistable, deliberately, because it must be portable and testable on its own; the binding between a Rust type and its descriptor lives one layer up, in kladde-persist. See [Schema binding](#).

34.2.4 kladde-types and kladde-derive — the type vocabulary

The built-in containers, and the macro that turns user types into backed ones. Both build on kladde-persist, and neither depends on the other.

The macro is re-exported from kladde, not from kladde-types, so that an application can use the containers without the macro machinery and vice versa — and because generated code is rooted at `::kladde`, which is where the items it names are re-exported from.

34.2.5 kladde — the entry point

Kladde<T> pairs a root value with a Store, and is what application code holds. It writes the root’s descriptor table and fingerprint into the file at creation, checks the fingerprint at open, and hands out guards, transactions, and batches.

34.3 The seam

For anyone porting kladde, one boundary matters more than the rest.

Below kladde-persist, the design is language-neutral. The address table, the journal, the flush, the consolidation policy, and the schema model all translate more or less directly; they are documented in [Specification](#) and [Implementation](#) rather than here.

At and above kladde-persist, the design is Rust-specific and should be redone.

- Persistable/Guard exists because Rust cannot intercept a field assignment.
- The `&self write / &mut self read` split exists because Rust’s borrow checker can then enforce “no reading stale data mid-write” for free.
- `INLINE_SIZE` as an associated constant exists because Rust can compute field offsets at compile time. A language without compile-time constant folding will compute offsets some other way — probably once per type at startup, which is fine, but it changes the shape of the code.

A Python port should keep the storage layer almost verbatim and throw away the guards entirely. A C++ one might use proxy objects with `operator=`; a Java one might use bytecode enhancement, which is what db4o did.

34.4 Cross-cutting concerns

Crash consistency is a property of the journal and the flush, but every container must uphold the [ordering discipline](#) for it to hold, and no type system enforces it.

Freeing is a type-driven hook that spans the persistence layer and the store; see [Freeing](#).

Schema evolution touches the load path, the mutation path, and the flush, because a value read at a foreign layout must not then be mutated at native offsets; see [the specification](#). Until it exists, a file whose root fingerprint differs from the application's is refused at open.

Chapter 35

Pointers

The handle types, what each is for, and the ownership discipline they enforce.

35.1 Two types, one idea

type	copyable?	owns?
Pointer	yes	no
UniquePointer	no	yes

Pointer is the serialized, at-rest form: an allocation's id and nothing else. It is Copy, it is what a Location anchors to, and it is what gets written into a file. Its field is a NonZeroU32, so Option<Pointer> gets the null niche in memory for free, and the [on-file null encoding](#) — all-zero bytes — is sound rather than merely conventional.

UniquePointer is the owned handle: single-owner, not Copy. Holding one *is* the claim to the allocation, and it is what `resize`, `splice`, and `free` take, so that only the owner can reshape or release an allocation. It is untyped: an allocation holds bytes, and the type that owns the handle knows what they mean.

35.2 Why exactly one copyable type

If a pointer can be freely copied, a moved allocation may have arbitrarily many live references to it, which requires a registry just to find them all. With exactly one owned handle per allocation, ownership is local: nothing needs finding, because the serialized value is a stable id and the in-memory owner is unique by construction.

Pointer is copyable because it is *not* a claim — it is an id-like value, useful for anchoring a Location and for serialization, and it confers no rights.

35.3 Ownership

Exactly one owning handle per allocation, at all times.

This is a format-level invariant, not a Rust convention: [the value graph is a tree of ownership](#), and a tool may rely on it.

Ownership **transfers** rather than duplicating. Storing a value that already owns allocations writes only its handles, so moving a value from one container to another moves its content without copying a byte.

Application code never constructs or holds a bare owned handle. Only library code does — the containers and generated code — so this is an invariant of a handful of types rather than something every mutating method must remember.

35.3.1 The one hole

`Persistable::load` reconstructs an owned handle from the `Pointer` it reads, which means it **can mint a second owner**. It is needed, since loading a value has to recover the handle from the serialized id, and there is no way to do that without constructing one. It is a load-time operation only, and the single-owner property is therefore conventional at that one point rather than enforced.

35.4 Width

The format [fixes ids at 32 bit](#), and so does `Pointer`. `Pointer` is nevertheless generic over a `Word`, defaulted to `u32`, and `Persistable` is generic over the pointer type, so that a per-file width, should one ever arrive, is a type argument rather than a rewrite.

Chapter 36

Persistable and guards

The central traits, why mutation needs a guard, and why it returns a Result.

36.1 Persistable

```
pub trait Persistable<P: PointerRepr = Pointer>: Sized {
    const INLINE_SIZE: usize;

    type Guard<'s, B: WriteBackend<Pointer = P>>: Guard<Persistable = Self, Backend = B>;
    fn guard<'s, B: WriteBackend<Pointer = P>>(
        &'s mut self, backend: &'s B, location: Location<P, B::Size>,
    ) -> Self::Guard<'s, B>;

    fn store<B: WriteBackend<Pointer = P>>(
        &mut self, backend: &B, location: Location<P, B::Size>,
    ) -> Result<(), Error>;
    fn load<B: ReadBackend<Pointer = P>>(
        backend: &mut B, location: Location<P, B::Size>,
    ) -> Result<Self, Error>;
    fn free<B: WriteBackend<Pointer = P>>(&mut self, backend: &B) -> Result<(), Error> {
        Ok(())
    }

    fn describe_local(builder: &mut SchemaBuilder) -> TypeDescriptor;
}
```

A `Persistable` is a plain, in-memory value that *has the right shape* to be persisted, but is not itself tied to any backend. Reading it is ordinary reading.

36.1.1 INLINE_SIZE

Every type has a fixed-size **inline representation** — what a containing struct reserves for it, regardless of how much variable-length content it may own elsewhere:

- a scalar: its own bytes;
- a derived struct: the sum of its fields’;
- a derived enum: a discriminant of 1, 2, 4 or 8 bytes plus the largest variant;
- an owning type: a fixed header, *not* the size of its content.

Having *some* fixed inline size is what makes sibling fields’ offsets statically computable, and computing them at compile time is what makes field access free.

36.1.2 Location

```
struct Location<P, S> { anchor: P, offset: S }
```

Where a value's inline bytes live: the nearest ancestor that owns a real allocation, plus a byte offset within it. A deeply nested leaf does not know or care what its ancestor's allocation holds, which is why the anchor is the type-erased `Pointer`. `Location` is threaded down the same reborrow chain that carries the backend.

36.1.3 Why `store` takes `&mut self`

A value that owns an allocation may not have one yet. A vector built by `from_iter` holds real content but no pointer, because nothing gave it a backend, and `store` is where that allocation happens the first time. If `store` took `&self`, it could still allocate and write correctly — but the caller's copy would stay stuck believing it has no allocation, breaking any guard obtained from it afterwards.

36.1.4 Why `load` takes `&mut B`

Loads are **sequential** — one field or element after another — so a single exclusive borrow reborrowed down the recursion suffices, and the read path can hand out a real seekable cursor with no interior mutability at all. As a free side effect, the borrow checker forbids loading while any guard is alive, since a guard holds a shared borrow.

36.1.5 `free`

Releases the allocations a value owns, recursively; see [Freeing](#).

36.2 Guards

```
pub trait Guard {  
    type Persistable;  
    type Backend;  
    fn as_persistable(&self) -> &Self::Persistable;  
    fn as_persistable_mut(&mut self) -> &mut Self::Persistable;  
    fn backend(&self) -> &Self::Backend;  
}
```

A guard borrows both the value and a backend for a lifetime — exactly the relationship `MutexGuard` has to `Mutex`, and the naming is deliberate. It is an RAII token proving exclusive, recording access.

Every mutating method on a guard does two things: it emits the records the mutation requires, and it mutates the in-memory value. **Mutating methods never return anything for the caller to separately apply.** Persistence is not optional or forgettable by construction.

Non-mutating access uses the plain value; guards implement `Deref`, so read-only methods stay available inside a mutating context.

36.2.1 Mutations return a `Result`

A mutation is durable once it returns, which it can promise only because it appends to the journal before returning — and an append is I/O, which can fail. So every mutating method returns `Result<_, Error>`, and the order inside it is fixed: **record first, then mutate memory**. A mutation that fails leaves the in-memory value as it was, and the file at worst holding an allocation that nothing reaches yet, which the [ordering discipline](#) permits.

A failed `fsync` [ends the session](#), so after one the store is **poisoned**: every later call returns `Error::Poisoned`, and the application drops the `Kladde` and reopens the file, which then holds every mutation that returned `Ok`.

36.2.2 Guards are generated per type

Not one shared generic wrapper, and the reason is Rust’s orphan rules.

A single `Guard<T, B>` defined in the persistence crate would be a *foreign* type from any downstream crate’s point of view, and Rust forbids inherent `impl` blocks on foreign types outright, regardless of what their parameters are filled with. So a library author could not write ordinary methods on their own guard.

Per-type generation means `MyTypeGuard` is local to the crate that declared `MyType`, and ordinary `impl` blocks just work.

36.2.3 The write/read asymmetry

`WriteBackend`’s methods take `&self`; `ReadBackend`’s take `&mut self`.

Write is `&self` because a parent guard holds `&B` and hands each nested field guard the *same* `&B` by reborrow, which is what lets sibling guards coexist. The interior mutability this requires lives in the concrete backend; see [The store](#).

Read is `&mut self` for the reason above: sequential access, real cursors, and the free staleness guarantee.

The two traits are also split like `Read/Write` in `std`: `ReadBackend` has *no* write or allocation surface at all, so `&mut impl ReadBackend` genuinely cannot mutate.

36.3 Why an automatic flush cannot tear a mutation

The store flushes by itself once its journal outgrows its budget, and it does so [from inside the mutating call](#) that pushed it over — while a guard is alive. That is safe for two reasons, one per layer.

The flush touches no `Persistable` value. It folds journal records into pages; it never reads or writes an in-memory value, so the guard’s borrow of the root is irrelevant to it, and the store’s own interior borrow is released before the flush takes it again.

A flush lands only on a transaction boundary. It runs after an append, and an append is a whole transaction — a bare operation, a committed transaction, or a batch’s ready run — so every state a flush commits is one the [ordering discipline](#) already requires to be valid.

36.4 The chain, end to end

```
notes.guard()           // NotesGuard - anchor = root allocation, offset 0
    .lines_mut()         // PersistableVecGuard - same anchor, offset 4
    .push(value)?;       // appends the record, then mutates memory
```

Each `_mut()` reborrows the same backend and extends the location by that field’s static offset. The caller supplies a backend exactly once, at the outermost guard.

36.5 What this costs

- **Verbosity.** `x.guard().field_mut().set(v)?` where other languages write `x.field = v`.
- **Borrow contention.** A guard borrows the root, so nothing else can touch the structure while it lives.
- **No partial borrows.** You cannot hold guards on two different fields at once, because each reborrows the root.

The first is inherent to Rust. The third is a real limitation with no current answer.

Chapter 37

Containers

How the built-in containers are laid out, and the design choices behind each.

The containers are **hand-implemented** against `Persistable`/`Guard`/`UniquePointer`, not generated. This is the same division `std` makes: its collections hand-write raw pointer manipulation internally, and the derive macro targets application-level types composed from already-persistable fields.

37.1 `PersistableVec<T>`

Inline: the content allocation's `Pointer`, all-zero while there is none. **Content:** one allocation holding a dense array of fixed-size element slots, element `i` at `i × T::INLINE_SIZE`.

The length is not stored. It is the content allocation's size divided by the element size, since the store already knows every allocation's size. That halves the inline footprint and removes a field that could drift out of step with the allocation — and it makes a push **one record**: a `Write` past the end [grows its allocation](#), so writing the new element and publishing the new length are the same atomic step. The price is that a zero-sized element type is unrepresentable, and `PersistableVec` refuses one.

A push costs one `Write`, a pop one `Resize`, and a removal from the middle one `Ssplice`, which the [fold](#) turns into a statement edit when the vector was written in the same flush.

`T` needs only `Persistable` — no `Clone`, no `Serialize`. That falls out of storing a fixed-size inline representation rather than going through a serializer: a guard method serializes a value into the record while only *borrowing* it.

Removal transfers ownership. `remove` and `pop` return the element with its allocations intact, so it can be pushed elsewhere; `delete` removes and [frees](#) it.

37.2 `PersistableString`

A thin wrapper around `PersistableVec<u8>`.

The interesting part is why it exists at all. Plain `String` deliberately has **no** `Persistable` implementation, and that absence is the enforcement mechanism: a struct field typed `String` fails to compile rather than silently persisting nothing. `String` cannot implement the trait because it has nowhere to keep the pointer to its own content allocation; `PersistableString` is a library struct with room for one.

37.3 `PersistableHashMap<K, V>`

Inline: the slot array's `Pointer`. **Content:** an array of fixed-size slots, each a one-byte liveness tag followed by a `(K, V)` pair.

The design turns on one observation: **the on-disk form does not need to support lookup**. That is what the in-memory map is for, so the content is a flat slot array with no hashing structure at all.

Removal is tombstoning — clear the tag in place, move nothing — and an insertion reuses the lowest tombstoned slot before it grows the array. Two consequences:

- No “which key is at slot n ” reverse lookup is needed, so each key is stored **exactly once** in memory, in a single map from key to slot-and-value. Hence `K: Clone` is not required.
- The array’s length is a **capacity**, the highest slot in use, rather than a live count; reusing tombstones keeps it at the high-water mark of live entries rather than of entries ever inserted.

Keys are immutable. No guard is ever handed out for a key, so `Persistable` on `K` governs only how its bytes are read and written.

37.4 `PersistableBlob<T>`

Wraps any `T`: `Serialize` + `DeserializeOwned` + `Default` as an opaque payload with its own allocation.

The escape hatch for foreign types that can get neither their own wrapper nor a derive. It is deliberately the least attractive option: rewritten in full on every change, opaque to [tooling](#), and the only thing in the workspace that needs a serialization dependency at all — hence the feature gate.

37.5 What is missing

A rope, for large text with efficient middle insertion. Designed in outline, not built. It is the container that would most benefit from the [fold’s zero-I/O splice](#).

Chapter 38

The derive macro

What `#[derive(Persistable)]` generates, and why.

38.1 What it generates

For a struct or enum, three things:

1. a `Persistable` implementation — `INLINE_SIZE`, `store`, `load`, `free`, and the schema descriptor;
2. a **guard type**, `MyTypeGuard`, with a `_mut()` accessor per field and a whole-value set;
3. a `Deref` implementation on the guard, so read-only methods stay available — and no `DerefMut`, which would hand out a `&mut` whose changes no journal record describes.

```
#[derive(Persistable)]
struct Point { x: i32, y: PersistableString }
```

generates, in essence:

```
impl<'s, B: WriteBackend> PointGuard<'s, B> {
    fn x_mut(&mut self) -> I32Guard<'_, B> {
        self.inner.x.guard(self.backend, self.location)
    }
    fn y_mut(&mut self) -> PersistableStringGuard<'_, B> {
        self.inner.y.guard(self.backend, self.location + 4)
    }
}
```

Each accessor reborrows the **same** backend and extends the location by that field's static offset — the sum of every earlier field's `INLINE_SIZE`. `y`'s offset is 4 regardless of how long its string currently is, because a string's inline representation is a fixed header.

38.2 The aggregate records nothing of its own

A struct owns no storage beyond its fields, so its guard has nothing to write. Every accessor just reborrows into the field's own guard, recursively, and `free` recurses into every field. This is why the macro is small: it computes offsets and forwards; all the actual work is in the leaf types.

38.3 Primitives are `Persistable` too

`i32`, `bool`, `char` and friends have implementations, specifically so the macro can treat **every** field uniformly rather than special-casing leaves. Those implementations must live in the crate that *defines* `Persistable`, because `impl Persistable for i32` is `impl ForeignTrait for ForeignType` from anywhere else, which the orphan rules forbid; the same reasoning forces the scalar guard types into that crate.

38.4 Enums

An enum is a discriminant followed by the selected variant's fields, each variant laid out like a struct in its own right, based past the discriminant. So, like a struct, an enum owns no allocation, and its `INLINE_SIZE` is the discriminant plus the largest variant's field sum. Positional fields are named by position — "0", "1" — which keeps the schema model uniform.

The discriminant is as wide as its values need, unless an integer `#[repr]` fixes its width. `#[repr(u8)]`, `u16`, `u32` or `u64` sets the width; without one, it is the smallest of 1, 2, 4 and 8 bytes that holds the largest discriminant value, the widths the [Enum descriptor](#) allows. A signed or pointer-sized `repr`, and a negative discriminant, do not compile, since descriptors store discriminants unsigned and `size` has no fixed width. Without a `repr`, the width is a layout cliff: a 257th variant, or a discriminant of 256 or more, widens every value of the type and moves every field behind it in the structs that contain it. A type whose layout has to stay put pins the width with a `repr`.

The guard's `parts()` hands out the guards of the current variant's fields, as a generated enum `{Enum}Parts` with the same variants, so a field is mutated in place by matching:

```
match guard.parts() {
  ShapeParts::Circle(mut radius) => radius.set(2)?,
  ShapeParts::Rectangle { mut width, .. } => width.set(3)?,
  ShapeParts::Origin => {}
}
```

Each field guard is based at the field's offset past the discriminant and writes inside its field only, so the discriminant cannot change underneath it. And `parts()` borrows the guard, so `set` cannot switch the variant while a field guard is alive. `set` is how the variant changes: it stores the new value, then frees the old one. An enum without any fields gets no `parts()`, since it has nothing to mutate in place.

38.5 Generics and transparency

Type parameters are supported: each gets a `Persistable` bound, the heuristic `#[derive(Debug)]` uses. Lifetime and `const` parameters are rejected.

`#[kladde(transparent)]` on a single-field struct persists it exactly as its field — same bytes, same descriptor, same fingerprint — and gives its guard one `get_mut()` returning the field's guard, as `#[serde(transparent)]` does.

38.6 Path resolution

Generated code names items by absolute path, because a macro cannot know what the user called their imports. It roots them at `::kladde`, and `kladde` re-exports every item generated code or a hand-written `impl` names, so an application depends on `kladde` and nothing below it. That matters for more than convenience: if the facade did *not* re-export them, an author would add `kladde-persist` themselves, and a version that disagreed with the facade's would produce two distinct copies of the `Persistable` trait and an error that prints the expected and found types identically.

A library built directly on `kladde-persist` has no reason to pull the facade in, and redirects the macro instead, as `serde`'s `#[serde(crate = "...")]` does:

```
#[derive(Persistable)]
#[kladde(crate = "kladde_persist")]
struct Node { /* ... */ }
```

38.7 What is missing

Field attributes. There is no `#[kladde(skip)]` for a non-persisted field, no `#[kladde(id = ...)]` for rename-robust identity, and no way to declare a default for a field added later. The last two become necessary when [evolution](#) lands.

38.8 Testing compile failures

The macro's *compile errors* are part of its contract — a `String` field must fail to compile, and that is the mechanism preventing silent data loss. These are tested with `rustdoc compile_fail` doctests rather than a separate harness, which keeps the expected-failure cases next to the documentation that explains them.

Chapter 39

Freeing

Recursive reclamation of backed memory: a type-driven hook, and the points where ownership ends.

39.1 The problem

An allocation must be released when the value that owns it goes away — overwritten, or removed from a container and not kept.

The obvious mechanism is unavailable. An owned handle **cannot free itself** from its own destructor, because it holds no reference to the backend. It deliberately holds none: a handle is meant to be small, and embedding a backend reference in every one would inflate every container and every derived struct. So freeing has to be driven by something that *does* have the backend in hand, which is a guard.

39.2 The mechanism

`Persistable::free(&mut self, backend)` releases everything a value owns, and guards call it where ownership ends.

- **Derive-generated** for structs and enums: recurse into each field, or into the current variant's.
- **Hand-written** for the containers: free every element, then the content allocation.
- **A no-op** for scalars, which own nothing.

The walk follows **owning pointers only**, which is trivial here because the type knows which of its fields own. That is the decisive advantage of putting the hook on the type rather than deriving it from the schema.

39.2.1 Ordering

The free walk must respect the [ordering discipline](#): **publish, then free**. A set on any guard stores the new value — writing the pointer that publishes it — and only then frees the old one. A crash in between leaks, which is recoverable, rather than dangling, which is not.

39.3 Where ownership ends

Overwriting a value with `set` frees the old one.

Removing from a container comes in two flavours, because taking a value *out* is not the same as discarding it:

- **transfer** — remove and pop return the value with its allocations intact, so it can be stored elsewhere without copying a byte;
- **delete** — delete and clear free what they remove.

A transferred value that is then dropped without being stored anywhere leaks its allocations in the file, exactly as a `Box` leaked with `std::mem::forget` leaks memory. Nothing can prevent that without a destructor that knows the backend, which is the thing handles deliberately lack.

39.4 Partial initialization

No container has slots it has not written: a vector's length is its allocation's size, so every slot below it was written, and a hash map records each slot's liveness in a tag byte. So a free walk never recurses into uninitialized bytes, and the format needs no notion of them.

39.5 The schema-level alternative

A tool that does not link the application cannot use a type-driven hook, so a language-independent garbage collector or leak checker would need the schema to say which fields are owning pointers. That is what the reserved `Pointer descriptor kind` is for, and it stays deferred until such a tool exists.

Chapter 40

Transactions

The Rust surface over the [transaction and batch machinery](#): RAII types, and what happens when a panic unwinds through one.

40.1 RAII types

The store's `begin_transaction`, `end_transaction`, `begin_batch` and `end_batch` are **private**, because they are dangerous: it is very easy to forget an end call, which prevents every subsequent mutation from ever being persisted.

The public API provides them through RAII types instead:

- `Kladde::transaction()` returns a `Transaction`, and `Transaction::commit(self)` ends it;
- `Kladde::batch()` returns a `Batch`, and `Batch::end(self)` ends it;
- `Batch::transaction()` opens a transaction inside the batch — the nesting [the design is arranged around](#).

Each offers `guard()`, so mutations go through the scope rather than around it, and `commit` and `end` return a `Result`, since ending a scope appends to the journal. Because the scope object borrows the `Kladde` mutably and an inner scope borrows the outer one, a scope cannot be ended while a scope nested inside it is still open: the interleaving errors of the [state table](#) do not compile.

Dropping a scope without ending it ends it, and a failure to end it there — where no error can be returned — poisons the store, since the in-memory mutations have then already happened and the journal does not hold them.

There is no `Transaction::cancel`. Cancelling the transaction's effect on the *file* would be possible, but its effects on the in-memory representation are always immediate, and redoing them would mean reloading the entire affected value from the file.

40.2 Behaviour on unwind

Drop branches on `std::thread::panicking()`, because the two scopes differ:

- **Batch: end normally, no poisoning.** Every transaction boundary inside the batch was, by the ordering discipline, a valid state. Committing the completed ones loses nothing, and discarding them would throw away durability the application already believes it has.
- **Transaction: discard the in-flight operations and poison the store.** Committing a half-built transaction would persist exactly the invalid intermediate state that transactions exist to prevent. But not committing leaves the file and the in-memory representation disagreeing, for the same reason there is no `cancel`, and poisoning is what stops that divergence from being silently observable.

“Poisoned” means *every subsequent store call returns an error*, not *panic* — panicking during unwind aborts the process. Recovery is to drop the `Kladde` and reopen the file, which then holds the last completed

transaction. So a panic loses the in-flight transaction, a crash loses the in-flight transaction, and both leave the file at the last valid state.

40.3 Open questions

- **Is a long-lived batch a footgun?** Batches delay persistence until they are ended, so holding one across a long computation quietly defeats kladde's main feature. Whether the API makes that tempting in common usage patterns is worth checking before it is frozen.

Chapter 41

Schema binding

The Rust binding of the [language-independent schema model](#), and the check it performs at open.

41.1 Two layers

The split is deliberate and worth preserving in a port.

kladde-schema — the model. The descriptor types, the canonical encoding, the fingerprint computation, and a SHA-256 implementation. It has **no** connection to the store, to `Persistable`, or to any Rust type system machinery, which is what makes it portable and testable on its own: the conformance vectors for [encoding](#) and [fingerprints](#) exercise this layer and nothing else.

The binding — where Rust types meet the model. A `Persistable` type declares its own descriptor in `describe_local`, and the `derive` macro generates that declaration.

41.2 The binding

A type contributes its descriptor by describing itself into a `SchemaBuilder`, which interns descriptors into a table and hands back references:

- **Recursion terminates.** A type that refers to itself reserves its table slot, keyed by its `TypeId`, before describing its fields.
- **Descriptors are deduplicated** by type, so a type used in twenty places occupies one table entry.
- **The table is built once**, not per value.

41.3 What a type declares

A struct declares a `Struct` descriptor with its fields in declaration order; an enum declares an `Enum` with its variants canonically ordered by discriminant.

A hand-written implementation declares **whatever descriptor matches the bytes it actually reads and writes** — which is the rule that most often trips people up. A hand-written implementation of a plain field-sum declares `Struct`, not `Opaque`; “hand-written” and “opaque” are different axes, see [the guiding principle](#).

The containers declare `Opaque`, carrying their library name, type name, version, inline size, and element types as parameters. Whether they *should* — as opposed to declaring structural descriptors so that tools can walk them — is [an open format question](#).

41.4 At create and at open

`Kladde::create` encodes the root type's descriptor table into an allocation that the header's `schema_table` names, and writes the root's fingerprint into the header. `Kladde::open` compares the header's fingerprint with the application's and **fails closed** on a mismatch, with an error carrying both fingerprints. It never parses the descriptor table on the matching path, which is the [fast path](#) the fingerprint exists for.

Resolution — reading at a writer's offsets — does not exist yet, so a mismatch cannot be bridged; see [Evolution](#).

41.5 Design invariants

Worth keeping in mind when changing anything here:

- **The fingerprint must not depend on table layout.** References are encoded structurally, so any renumbering or reordering produces the same hash; a test permutes the table and asserts equality.
- **A type's own name is not fingerprinted;** its fields' and variants' names are.
- **Determinism over everything.** No hash-map iteration order, no addresses, nothing incidental may reach the hash.
- **Encoding and fingerprinting are separate functions** over the same model. They share the canonical order but not the code, because the fingerprint deliberately omits things the encoding includes.

Chapter 42

The store

The Rust-specific half of the [storage layer](#): interior mutability, file I/O, and the concrete types of the [in-memory structures](#).

None of this changes what the structures *mean*; it is about realising them in Rust without making them awkward.

42.1 Interior mutability

Store keeps all of its state behind one `RefCell`, and borrows it for the length of one call. `WriteBackend`'s methods take `&self` so that sibling guards can share a backend, which forces interior mutability somewhere, and one cell is enough: no store method calls out into user code, so no borrow is ever held across a callback, and a single dynamic borrow per call costs nothing measurable next to the journal append it guards.

The one reentrant path is the [automatic flush](#), which runs inside the mutating call that pushed the journal over its budget. It runs after that call's own borrow has ended, and takes the cell again for itself.

Store is therefore `Send` but not `Sync`, which matches the [single-writer scope](#) of the format.

42.2 Storage

The store reads and writes pages through a `Storage` trait with two implementations.

FileStorage is a `std::fs::File` driven by positional reads and writes, so no call depends on a shared file cursor, and synced with `sync_data`, which covers the file-size change a growth makes. It extends the file with `set_len`, since the standard library offers no `fallocate`, and that is the [ungraceful form](#) of detecting a full disk: a disk that fills up surfaces at the flush's `fsync`, which ends the session, rather than as an error the flush could return.

MemoryStorage is a byte vector, used by `Kladde::new` and by tests. It also keeps the image as of the last sync apart from the writes since, so a test can simulate a power cut by discarding any subset of those writes and opening what is left — the [flush-interruption test](#) the conformance suite calls for.

42.3 The mirror

Address-table pages stay resident as the [implementation notes recommend](#), each a boxed page-sized array keyed by page number. Data pages are read through a small cache while a file is loaded, and dropped afterwards; a flush that needs their bytes again, to relocate survivors or to fold a `Copy`, reads them from the file.

42.4 The in-memory structures

42.4.1 Keys

An `(id, offset)` key is packed into one `u64`, `id` in the high half, so the fragment map is a `BTreeMap<u64, Fragment>` whose order is exactly the `(id, offset)` order the format needs, and a range over one `id` is a range of integers.

42.4.2 Fragments

`Fragment` is an ordinary Rust enum of twelve bytes, with a pending fragment carrying an index into the flush's table of pending entries.

A packed ten-byte representation is available if memory ever matters: the discriminant can be stolen from the page-offset field, using two values that a real offset can never take because [the page framing](#) puts a CRC there. It is not used, because it costs unsafe reads of `MaybeUninit` fields on every access for two bytes per fragment.

42.4.3 The statement slab

The language-agnostic shape is a slab of `{ page_or_next, pins }` plus framing lengths. Rust realises it as **two parallel arrays**, `Vec<StatementRecord>` of eight-byte records and `Vec<u8>` of framing lengths, rather than one packed record. A nine-byte packed record would need `#[repr(packed)]`, where taking a reference to `pins` — the field touched on every fragment gained or lost — is undefined behaviour; splitting gets the same nine bytes per slot with natural alignment, and reads `framing_len` only when a statement dies.

`StatementRef` is a `NonZeroU32`, and slot 0 is never handed out, so `Option<StatementRef>` is four bytes rather than eight: that saves twelve bytes per allocation across the allocation map's anchor and `grow_witness` and the recyclable set's tombstone.

`pins` is `u32` rather than `u16`, because a `Shrink` or `Tombstone` wins the gaps *between* newer statements anywhere in `[n, size)`, so its fragment pins are bounded by the allocation's fragment count and nothing smaller.

42.4.4 Maps keyed by id or page

The allocation map is a `HashMap<u32, AllocationMeta>` with a multiplicative hasher rather than `std`'s `SipHash`, whose protection against HashDoS an id-keyed map built from the file does not need. The page table is a `Vec<PageInfo>` indexed by page number, since [lowest-first placement](#) keeps page numbers dense.

42.5 Where the integer widths come from

Every width below is forced by [the specification's bounds](#), not chosen:

type	width	because
Pointer	<code>NonZeroU32</code>	allocation ids are 32 bit; zero is reserved so <code>Option<Pointer></code> is free
page offset	<code>u16</code>	page sizes are at most 64 KiB
allocation offset	<code>u32</code>	allocation sizes are below 2^{32}
page number	<code>u32</code>	page numbers are 32 bit
<code>StatementRef</code>	<code>NonZeroU32</code>	at most $2^{32} - 1$ statements in a file, and zero is the free niche
framing length	<code>u8</code>	a statement's framing is at most 21 bytes

Chapter 43

Evaluation

Measurements of [kladde-rs](#) under realistic workloads, and what they say about the design in [Implementation](#).

Each page names the commit it measured and the commands that reproduce it. Its figures are drawn by `tools/plot-evaluation.py` from the tables in `data/`, which are kept, gzipped, so that a figure can be redrawn without rerunning the benchmark.

The numbers come from one machine. Ratios, such as the file's size over its live size or the bytes written per byte the application wrote, carry over to other machines far better than times do.

43.1 The evaluations

page	what it measures
Consolidation under load	the design in Consolidation , on six workloads: space and write amplification, compaction after a mass free, the growth of the description, flush times, and variants of the churn floor, the target fill, and defragmentation's share
Cleaning by ripeness	the draft 's policy, implemented on a branch of kladde-rs , against the design above on the same workloads: the trade-off between space and writes, the controller that sets the price of space, what the logarithm in its threshold contributes, whether it keeps content that never changes apart from content that does, and what keeping the estimates costs
Ripeness with a static share	the draft 's later estimate, a draining share over a static one fitted to each page's losses, against the single rate per page above: how often pages earn a static share, and what keeping the fit costs
Bayesian ripeness	the Bayesian draft 's posterior per chunk, started from the file's empirical prior and decided by the expected gain or by the option to wait, against both estimates above: what it finds static, how it cleans, how its prior fares, and what it costs to keep and to rank

Chapter 44

Consolidation under load

Consolidation keeps a file at 1.5 to 1.7 times its live size under sustained overwrites, where the same file grows to 6 times without it, for about one more byte written per byte the application writes. After three quarters of a file are freed at once, it gives the space back within 16 flushes. Flushes that do this take 6 to 25 ms at the median.

Two of its constants do less than [the design](#) intends. The target fill $\tau = 0.8$ is out of reach under the churn floor $\lambda = 1$, so the budget sits at its cap and λ alone sets the fill, to within a percent of what a one-line model predicts. And description defragmentation's reserved page per flush merges next to nothing under appends: sixteen pages cut the statements by a third.

The benchmarks also found four defects, all fixed in the measured commit.

44.1 What was measured

Six workloads, and variants of three of them, ran on real files, each until the application had written eight times the file's live size, with a flush after every 1000 operations.

workload	live size	what the application does	what it exercises
uniform	1, 8 and 64 MiB	writes of 1 to 511 bytes at random offsets of random 16 KiB allocations	evacuation and the page rewrite
skewed	1, 8 and 64 MiB	the same, with 90 % of the writes going to 10 % of the allocations	scoring, with hot and cold content mixed in pages
append	16 MiB	512 logs, appended to in records of 16 to 240 bytes and truncated to nothing once a record would take one past 64 KiB	description fragmentation, and mass frees when the logs rotate, nearly in step
churn	16 MiB	values of 16 to 128 bytes, or one in five of 1 to 8 KiB, each operation replacing one with a new allocation, rewriting one whole, or patching 1 to 64 bytes of one	allocation turnover, id recycling, and Inline statements
shrink	64 MiB	allocations of 64 KiB, three quarters of them freed at once, then 400 flushes of 100 patches of 1 to 64 bytes each	compaction mode and truncation
typed	16 MiB	a <code>Kladde<PersistableHashMap<u64, PersistableString>></code> of strings of 50 to 500 bytes: 40 % of the operations replace a string, 30 % append to one, 20 % insert one, 10 % remove one	the typed layers

workload	live size	what the application does	what it exercises
variants	8 and 16 MiB	uniform and skewed at 8 MiB with λ of 0.5 and 2, or τ of 0.6 and 0.5; append with 4 and 16 pages of defragmentation share	the constants

The *live size* is the sum of the allocations' sizes, and every ratio below is taken against it. A workload first fills its file, which is not measured, and then records one row per flush: the file's pages by kind and their coverage, and everything written since the fill. The horizontal axis of most figures is the application's bytes written since the fill, over the live size. After the last flush, each run reopens its file and compares every byte with a model of what the application wrote.

Times include each flush's fsync. Journal appends are not fsynced, as [the journal's specification](#) prescribes: an operation survives a crash of the application once its call returns, and a power cut once the next flush has returned.

The runs measured [kladde-rs](#) at commit cb34576, built with --release by Rust 1.97.1, with the default [options](#) unless a variant says otherwise, on an Intel Core i7-1165G7 with 16 GB of memory under Linux 7.0, in a development container whose files live on an overlay file system. Identical configurations differed by up to 20 % in median flush time from one run to the next, so only differences well beyond that are worth reading into the times.

44.2 Space

Consolidation holds overwritten files at about 1.5 times their live size once they are much larger than what a few flushes write; without it, uniform overwrites grow the file sixfold.

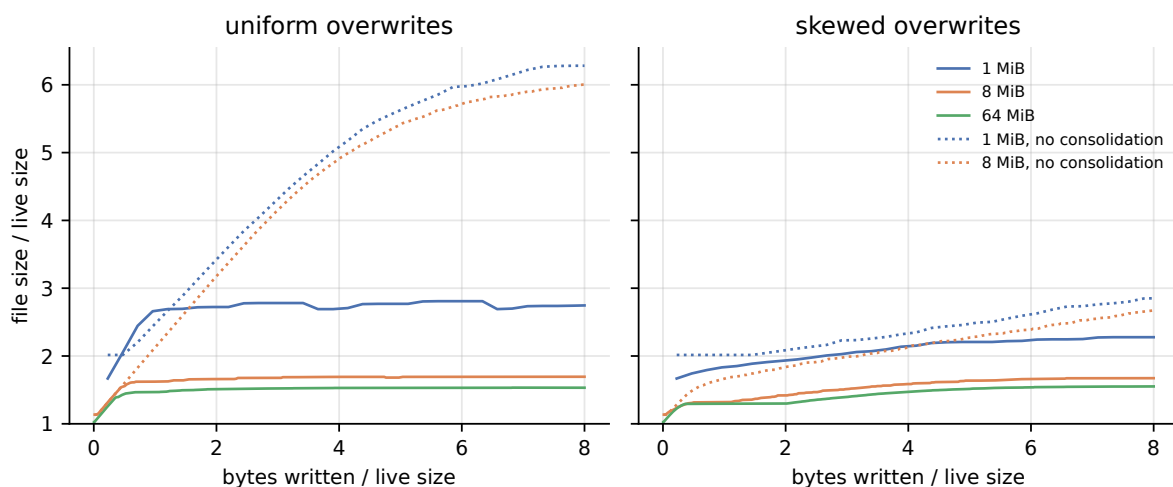


Figure 44.1: File size over live size as the overwrites accumulate, with and without consolidation.

file size / live size, at the end	1 MiB	8 MiB	64 MiB
uniform, consolidating	2.75	1.69	1.53
uniform, without consolidation	6.28	6.00	—
skewed, consolidating	2.28	1.67	1.55
skewed, without consolidation	2.85	2.67	—

Without consolidation, a data page is reused only once every byte on it has been overwritten. Under uniform overwrites that almost never happens, and data pages end up 18 % live; under skewed ones, hot pages do empty by themselves, and the file grows more slowly.

In the 64 MiB uniform file, data pages take 1.35 times the live size, at 73 % live, address-table pages 0.15 times, at 59 % live, and reusable pages 0.03 times. Under uniform overwrites, the data pages settle within half a live size of writes and the table pages within two. Under skewed ones, both keep losing fill for most of the run, as they would if pages froze above the churn floor once their hot content had died, which is [the problem the ripeness draft describes](#).

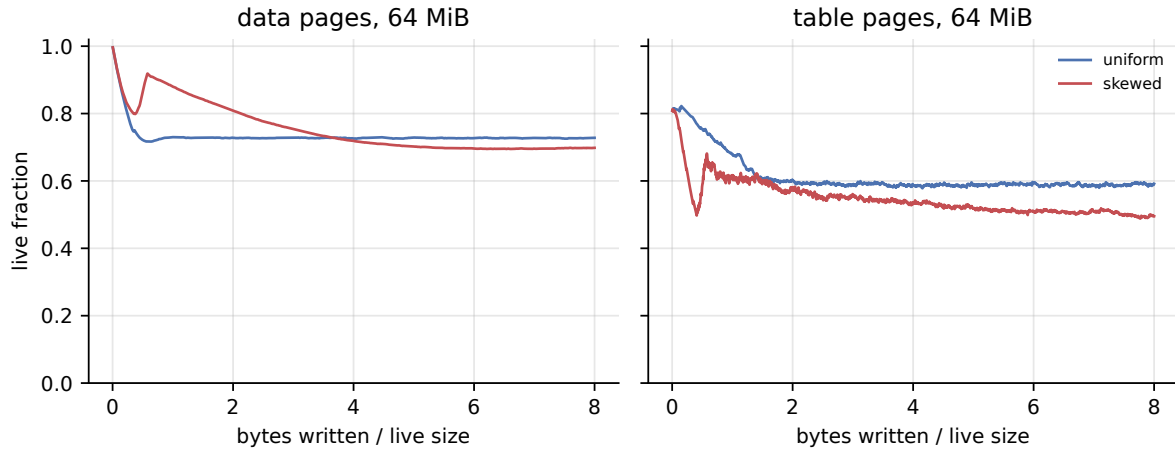


Figure 44.2: The live fraction of data pages and of address-table pages, in the 64 MiB files.

Small files pay for the quarantine: a file keeps three to four flushes' worth of written pages free, whatever its size. A page that a commit stops referencing can be reused only after the next commit, and the reusable pages that the next flush will write are free as well. The 1 MiB uniform file keeps 340 pages free and writes 104 pages per flush, the 8 MiB one 430 and 130, and the 64 MiB one 490 and 138. In the 1 MiB file, where one flush writes 40 % of the live size in pages, that is 1.33 times the live size; at 64 MiB it is 3 %.

44.3 The churn floor sets the fill, not the target

With the default constants, the budget climbs to its cap of 256 pages within 1.3 live sizes of writes and stays there in every overwrite run, but flushes open only 8 to 59 pages of it: the churn floor admits no more victims.

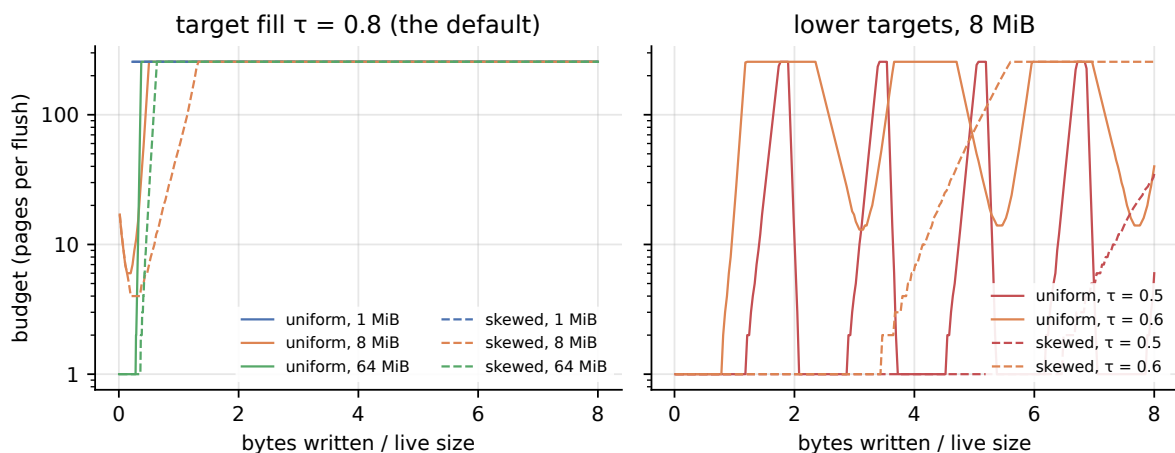


Figure 44.3: The budget, in pages per flush, with the default target fill (left) and with lower targets (right).

The fill the file reaches is then the churn floor's. A victim may be at most $1 / (1 + \lambda)$ live, so under uniform overwrites every data page is written full, loses its content at a steady rate, and is cleaned once

it reaches that fill. A page whose live fraction decays exponentially from 1 to $1 / (1 + \lambda)$ spends its time at an average fill of $\lambda / ((1 + \lambda) \cdot \ln(1 + \lambda))$, which matches the measurements to within about a percent:

churn floor λ	data-page fill predicted	measured, uniform	measured, skewed
0.5	0.822	0.825	0.819
1 (the default)	0.721	0.733	0.707
2	0.607	0.603	0.615

This is the opposite of the division of labour that [the design](#) intends, in which λ stays low enough never to decide the fill and the budget converges the file on τ . At $\lambda = 1$, the 8 MiB files reach a whole-file fill of 0.63, table pages and free pages included, well short of $\tau = 0.8$.

The controller does reach targets that the churn floor allows, but by swinging between its bounds rather than settling. With $\tau = 0.6$, the 8 MiB files end at whole-file fills of 0.58 (uniform) and 0.59 (skewed), and with $\tau = 0.5$ at 0.53 and 0.51. Under uniform overwrites, the budget meanwhile swings between one page and the cap, a swing taking one to two live sizes of writes. Multiplying the budget by $\exp(4 \cdot (\tau - \text{fill}))$ after every flush makes the controller an integral one, and the fill it measures, the whole file's, answers the budget only once new writes reuse the pages cleaning frees, so it overshoots.

To trade space for writes, λ is the better knob: a lower target moves the file along a worse curve, since its budget swings.

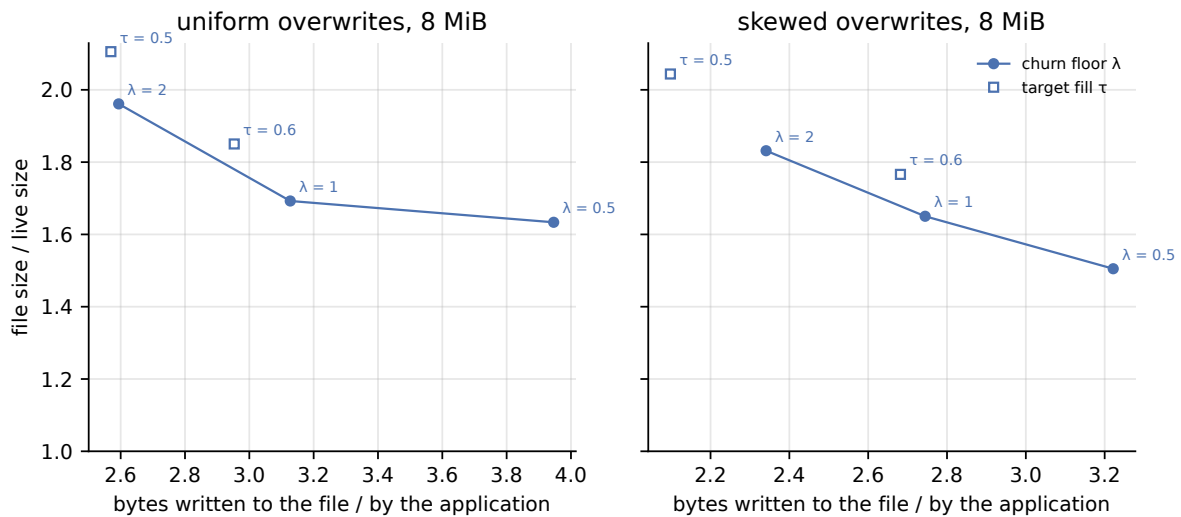


Figure 44.4: File size against bytes written in the steady state of the 8 MiB overwrite runs, the second half of each, for three churn floors and two lower targets.

The figure compares steady states: the file size averaged over the second half of each run, and the bytes written during that half per byte the application wrote. Under uniform overwrites, $\lambda = 2$ writes about as much as $\tau = 0.5$ (2.59 against 2.57 bytes per byte) and leaves a file 7 % smaller (1.96 against 2.11 times its live size). Under skewed ones, $\tau = 0.6$ leaves a file of 1.77 times its live size for 2.68 bytes per byte, 5 % larger than the churn floor's curve passes at those writes. While the swinging budget is low, the file grows, and what cleaning frees afterwards are holes, which the file keeps until new writes fill them.

44.4 Writes

Each byte the application writes costs 2.2 to 3.2 bytes written to the file: one in the journal, one in a data page, and up to one more for consolidation.

The journal takes 1.02 to 1.10 bytes per byte for the raw store, more the smaller the writes, since each record carries a header, and 1.12 through the typed layers. Consolidation adds 0.9 bytes per byte under



Figure 44.5: Where the bytes written to the file went, per byte the application wrote, over each whole run.

uniform overwrites, 0.6 to 0.8 under skewed ones, 0.7 under appends, 0.6 in the typed workload, and 0.1 under churn, whose replaced values free whole allocations. Address-table pages add 0.02 to 0.23, headers less than 0.04. Without consolidation, the same workloads write 1.5 to 2.2 bytes per byte: the 8 MiB uniform file 2.13, against 3.07 with it.

The uniform figure is what the churn floor predicts. Cleaning a victim at a live fraction u writes $u / (1 - u)$ bytes per byte it frees, 1 at $u = 1/2$, and under uniform overwrites nearly all garbage is freed that way, since pages seldom empty by themselves.

44.5 Mass frees

After three quarters of a 64 MiB file are freed at once, compaction mode gives the space back within 16 flushes; without it, the file never shrinks.

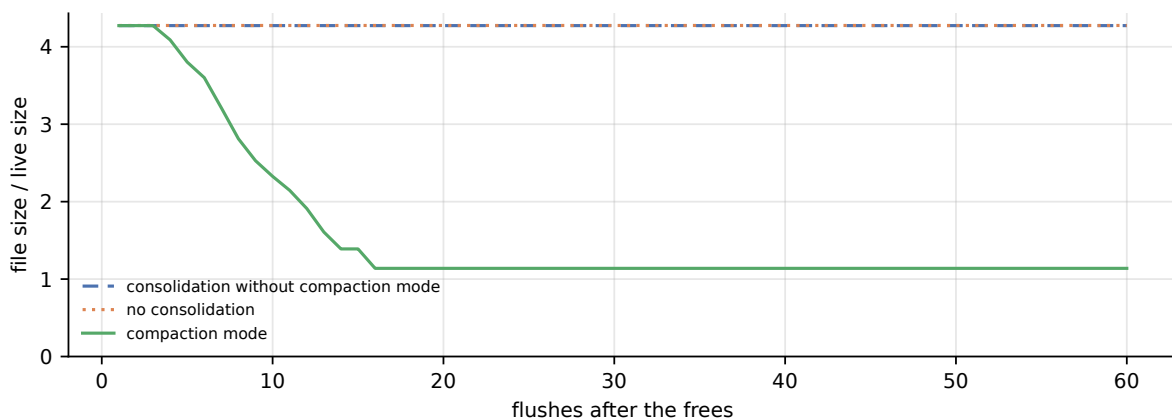


Figure 44.6: File size over live size after three quarters of the file were freed.

The file went from 4.27 to 1.14 times its live size, moving 3,221 data pages to return about 12,300, in 12 flushes in compaction mode and 12 truncations; over the remaining 385 flushes it grew by 2 %, as the patches added statements. Consolidation without compaction mode moves nothing here: after the frees every page is either full or empty, so no page is a victim, and the holes stay holes until the application writes as much again.

The append workload’s rotations, which free most of the file within a few flushes, are absorbed the same way. Without consolidation, the file peaks at 8.8, 6.2 and 5.5 times its live size as the logs rotate, and it has not shrunk back below 1.3 when the next rotation comes. With it, the peaks are 2.9, 2.3 and 2.3, and the file is back below 1.3 within half a live size of appends.

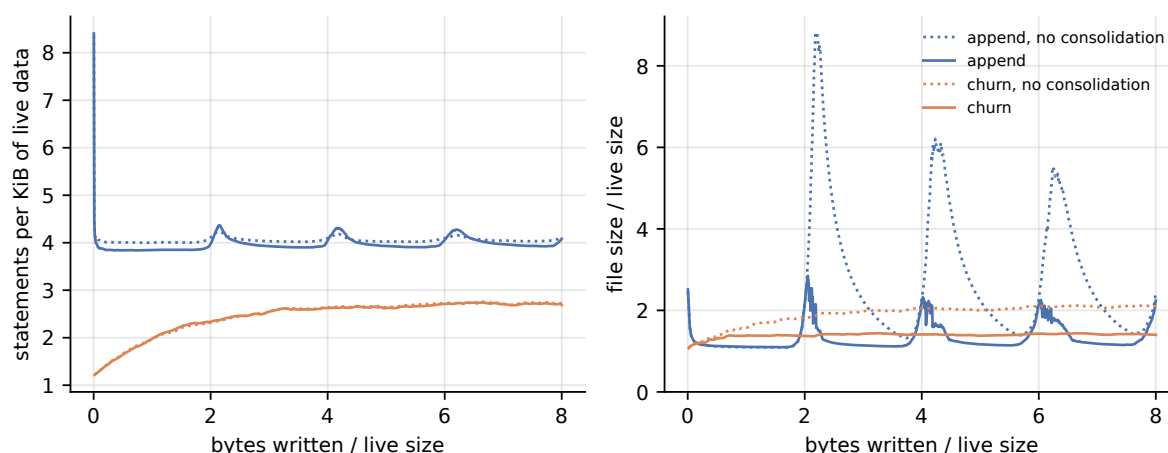


Figure 44.7: Statements per KiB of live data, and file size over live size, under appends and churn.

44.6 The description

Description defragmentation’s reserved page per flush is too small to matter under appends; sixteen pages cut the statements by a third, for a sixth more writes.

defragmentation share, pages per flush	statements per KiB of live data	file size / live size	bytes written per byte
none: consolidation off	4.06	2.80	2.21
1 (the default)	4.01	1.38	2.93
4	3.72	1.41	3.06
16	2.57	1.38	3.43

The statements and the file size are means over the second half of each run, since both swing with the rotations; the bytes written are over the whole run. Each flush appends about 125 KiB across 512 logs, two records per log on average, which the flush states as about one statement per log. A share of one page rewrites 4 KiB per flush, 3 % of what was appended, and that is what it did. What merging buys here is a smaller description, which is what the fragment map in memory holds, and not a smaller file: the share leaves the file’s size where consolidation put it.

Small patches accumulate as statements of their own under every setting measured. In the churn workload, statements grow from 1.2 to 2.7 per KiB of live data, with and without consolidation alike. In the shrink workload’s 400 flushes of 100 patches, statements grow from 4,833 to 45,837 and address-table pages from 44 to 439: every patch stays a statement, Inline since it is at most 64 bytes. That is the design working as intended for the writes, which cost the patch and not the page it lands in, but the description has no counterweight at this rate.

44.7 Time

A flush takes 6 to 25 ms at the median here, roughly in proportion to the pages it writes, and flushes take most of the time: the operations between them cost 1.3 to 3.2 μ s each.

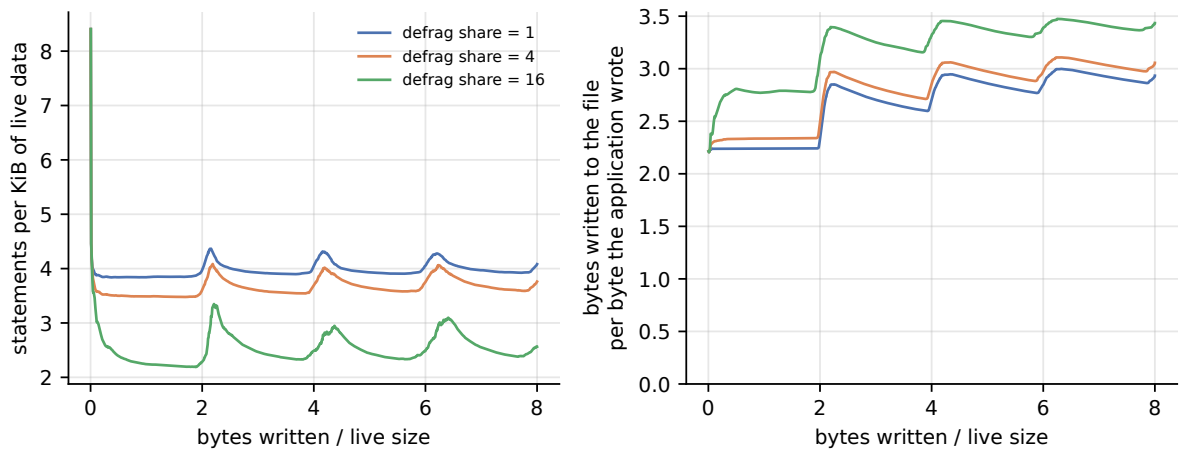


Figure 44.8: Statements per KiB of live data, and bytes written per byte, for three shares of description defragmentation.

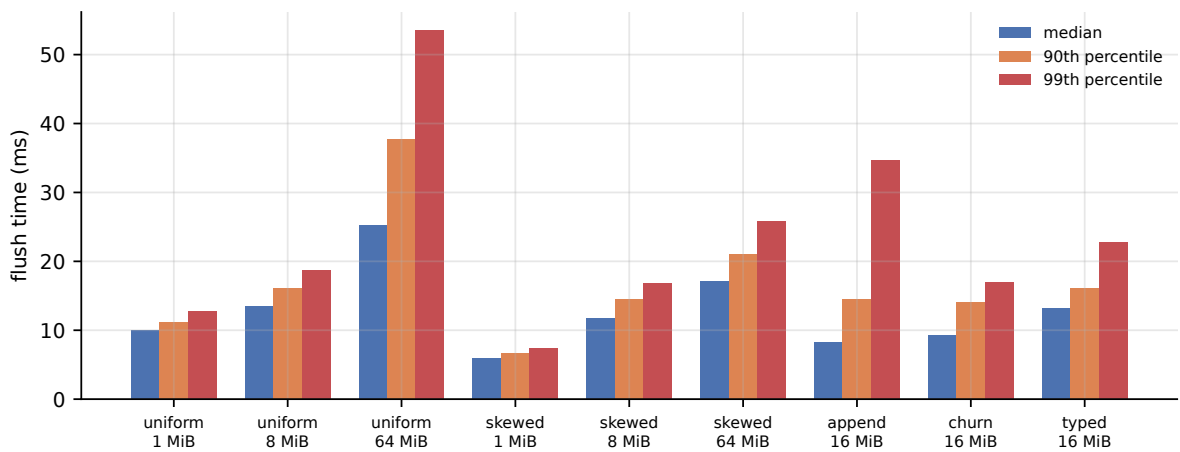


Figure 44.9: Flush times, including the fsync: median, 90th and 99th percentile.

workload, consolidating	pages written per flush	mean flush time	operations per second, flushes included
uniform, 1 MiB	104	10.4 ms	84,000
uniform, 8 MiB	130	14.1 ms	64,000
uniform, 64 MiB	138	29.6 ms	32,000
skewed, 64 MiB	131	18.2 ms	50,000
append, 16 MiB	64	10.1 ms	87,000
churn, 16 MiB	192	10.3 ms	110,000
typed, 16 MiB	82	14.5 ms	57,000

The figures are over the second half of each run. The uniform flush doubles its time per page written between 8 and 64 MiB, from 108 to 214 μ s, at about the same pages per flush. These runs do not say why; the page-table scans a flush makes, over some 25,000 entries, are far too cheap to account for 15 ms. The 99th-percentile flush of the 64 MiB uniform file takes 54 ms.

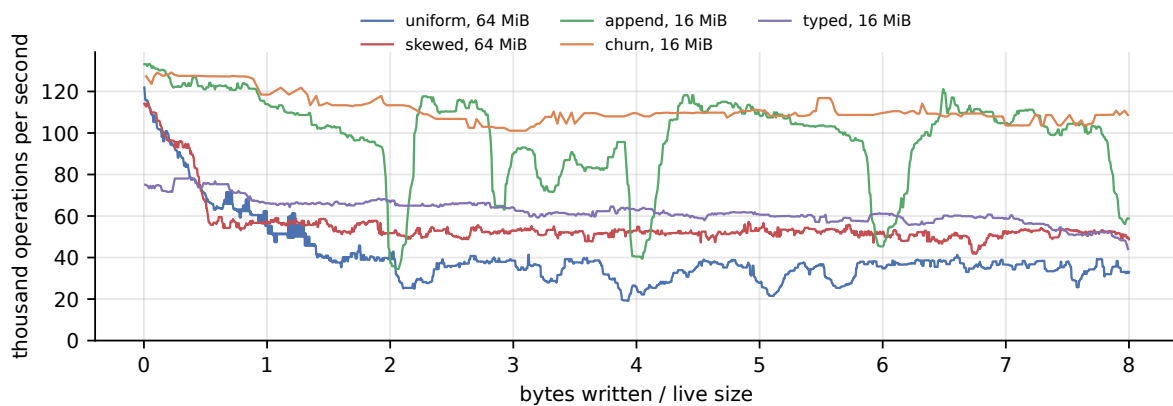


Figure 44.10: Operations per second, flushes included, over the median of 21 flushes.

Throughput dips while the append workload's logs rotate, when consolidation evacuates what the rotation left sparse.

44.8 The typed layers

Through the typed layers, a hash map of strings costs what the raw store does per byte: 1.53 times its live size in space, and 2.97 bytes written per byte. Its journal takes 1.12 bytes per byte, the framing of its records included. It holds 9.1 statements per KiB of live data, since each of its 137,000 strings is an allocation of its own. A flush takes 13 ms at the median, and the map sustains 57,000 operations per second, flushes included.

44.9 Defects the benchmarks found

All four are fixed in the measured commit, and each has a test of its own.

1. **Recycled ids resurrected old content** (fixed in 58dcfa0). A tombstone whose own page a flush dropped was released as if it had outlived its purpose, so after a reopen, older statements of a freed and recycled id could show through; the churn workload's check caught it.
2. **Storing a new string journaled it byte by byte** (fixed in 8a2f7cb), which made the typed workload's journal 2.8 times as large as its data.
3. **Compaction mode stalled on a full table page at the end of the file** (fixed in 4182b3f). The re-statements of a table page are estimated at 1.25 times its coverage, so a tail page more than 80 % full failed the churn floor, from which [the design](#) means the tail to be exempt: in the shrink workload,

a page of Inline patches at the end of the file held compaction back for 180 flushes. Before the fix, the file was still 3.2 times its live size after 400 flushes.

4. **Compaction mode ran almost permanently in small files** (fixed in 26dcc08), because it counted pages in quarantine as holes, although they are working space that the next flushes reuse by themselves. In the 1 MiB uniform file it ran in 30 of 33 flushes, and the file ended 6 % larger than it does with the fix.

implementation-notes.md in kladde-rs records the first, under “Flush”; [Compaction mode](#) now states the rules the last two fixes follow.

44.10 What to change

- **Make the constants agree.** Either lower τ to what λ allows, which for $\lambda = 1$ is a whole-file fill of about 0.63, or lower λ to what τ needs; or replace both with a threshold per page, as [the ripeness draft](#) proposes, which [Cleaning by ripeness](#) measures. As they stand, the controller only ever raises the budget to its cap, and the cap bounds a flush’s work only because the churn floor runs out of victims first.
- **Measure what the controller acts on.** [Cleaning by ripeness](#) moves a price with the same kind of controller, and its price cycled the same way while it measured the whole file; measured over the data pages and leaves, which answer within the flush, it held within a few percent. The budget would likely settle the same way; holes are compaction mode’s to return anyway.
- **Scale description defragmentation with the rate at which statements accrue**, rather than reserving one page per flush: appends and patches both outrun it by an order of magnitude.
- **Find out why flushes slow down as files grow**, with a profile of the 64 MiB uniform run.
- **Measure small files with fewer operations per flush**, since with 1000 their free space is set by the quarantine rather than by consolidation.

44.11 Reproducing

In kladde-rs, at the measured commit, run every workload and write one table per workload to a directory:

```
cargo run --release -p kladde-bench -- /tmp/kladde-bench
```

In kladde-docs, keep the tables gzipped next to this page and draw the figures from them, with matplotlib installed:

```
for f in /tmp/kladde-bench/*.csv; do
    gzip -9 -n -c "$f" > content/evaluation/data/consolidation/$(basename "$f").gz
done
tools/plot-evaluation.py --summary --out content/evaluation/figures/consolidation \
    main=content/evaluation/data/consolidation
```

--summary also prints the end state of every run as a table. The whole benchmark takes about four minutes here.

Chapter 45

Cleaning by ripeness

Updated for the draft's clarifications, as of kladde-docs commit **b33a836**: the branch now measures a page's fill relative to the fill survivors are packed at, which moved no result of 8 MiB or more by more than 1 % in file size or 2 % in writes. The clarified draft ranks a page by $h(u/u_0)$, with $u_0 = 1 - \theta$, and never cleans a page at or above u_0 ; the first version ranked pages by $h(u)$, with $1 - \theta$ only as a cap. The controller absorbs the difference: it settles the price of space 15 to 50 % higher, since $h(u/u_0)$ lies below $h(u)$ at every fill, and the trade-off curves of the two versions lie within 1 % of each other. The branch also carries the rates that fresh content starts from across sessions now, since the clarified draft rules out estimates that start from nothing; these runs, one session each, do not exercise it. The ablation that the clarified draft proposes, ranking without the logarithm, writes 3 to 6 % more for the same file size under skewed overwrites, and 1 to 4 % more under uniform ones: see [Without the logarithm](#). This update's runs shared their host with other work, so the flush times on this page are still the first version's; its tables are in kladde-docs commit 08febdf.

Cleaning by ripeness, as [the draft](#) proposes, keeps a file under skewed overwrites 10 to 12 % smaller than the current design for the same writes, or writes 4 to 20 % less for the same file size; under uniform overwrites the two are level, as the draft predicts. At the default settings of each, the 8 and 64 MiB skewed files come out 5 to 6 % smaller for 10 to 11 % fewer writes. It gains where the draft says it should: under skew, [the current design's](#) data pages lose fill for most of a run as pages freeze above the churn floor, and ripeness's hold at 0.76. Where hot, cool, and never-changing cold allocations [share pages at random](#), it gains about as much, a file 4 to 5 % smaller for 9 % fewer writes, since main's data pages freeze there too; neither policy separates the cold content from the rest.

It took one change to the draft to work: the controller that sets the price of space must measure the fill of the data pages and leaves, not of the whole file, or the price swings over decades. Three costs remain. Recording every page's estimate in the consolidator state takes 1 to 7 % of the writes. Under churn the file comes out 19 % larger. And after a mass free the price takes 13 to 19 flushes to climb back from where calm phases left it, so the file peaks higher.

45.1 What was measured

The [same benchmark](#) ran on the ripeness branch of kladde-rs, and on main. Main is at commit cb34576, as on [the consolidation page](#). Its tables here are from a run made back to back with the first version of the branch, so that their times compare; a run of main for this update reproduced every number in them but the times, as the benchmark is deterministic but for its times. The branch is at 362210a, which follows the draft as of kladde-docs b33a836; the first version, at f7d4eab, ranked pages by their absolute fill. Two more runs of the branch, at the same commit, measure what the consolidator state costs, by leaving it out, and what the logarithm contributes, by ranking pages without it, as the draft's ablation does. The mixed workload ran later, on the branch at 799c661, and on main from the branch main-mixed at 7646408; both commits add it to kladde-bench and change nothing else. The experiments this page cites besides, on the controller and on churn, ran on the first version, changed for the purpose, and are not part of the committed code.

The branch fills in what the draft leaves open with a forgetting rate $\beta = 0.1$ per flush, a floor $R_{\text{MIN}} = 10^{-4}$ and a starting price $\kappa = 0.01$ (the draft’s own examples), a cursor that gives up pages older than $W = 8$ flushes, and a budget of 256 pages per flush, which is now a fixed cap. Its controller aims the data pages and leaves at a fill $\tau = 0.75$ by default, where main aims the whole file at 0.8 and never gets there. `implementation-notes.md` in `kladde-rs` lists every departure from the draft.

Where main’s page compares policies, it uses **steady states**: the file size averaged over the second half of each run, and the bytes written during that half per byte the application wrote. Both controllers take a live size or more of writes to settle, which end-of-run values would mix in.

45.2 The trade-off

Under skewed overwrites, ripeness’s curve lies below main’s everywhere they overlap; under uniform ones, the two curves cross, within 3 % of each other.

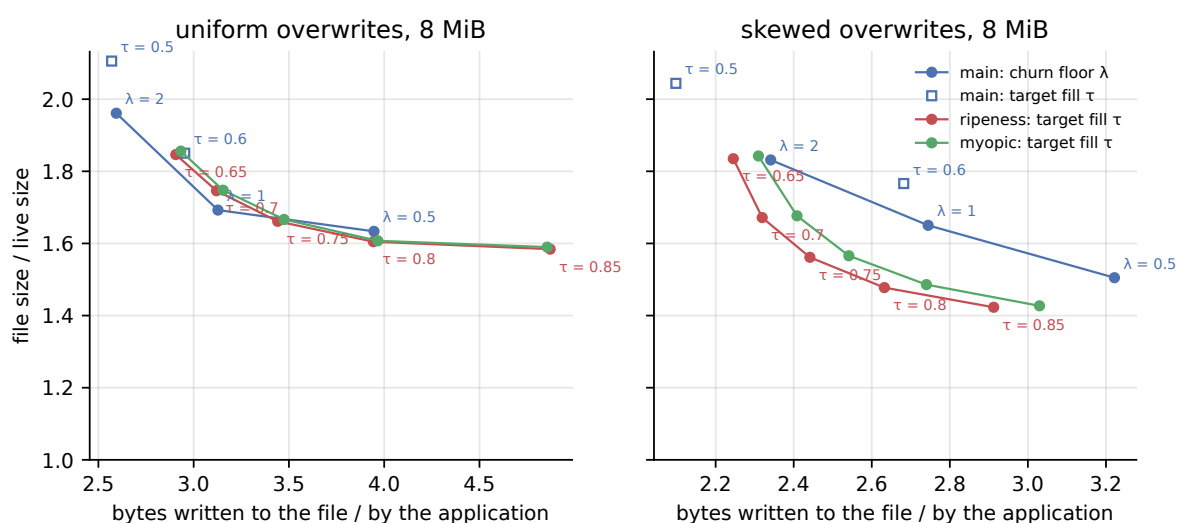


Figure 45.1: File size against bytes written in the steady state of the 8 MiB overwrite runs: main’s churn floors and targets, ripeness’s targets, and the same targets ranked without the logarithm (“myopic”), labelled where ripeness’s are.

8 MiB, steady state	main	ripeness, same writes	ripeness, same file size
skewed, $\lambda = 2$	1.83 times the live size, 2.34 bytes per byte	1.65 (−10 %)	2.25 bytes per byte (−4 %)
skewed, $\lambda = 1$	1.65, 2.74	1.46 (−12 %)	2.34 (−15 %)
skewed, $\lambda = 0.5$	1.51, 3.22	below 1.42	2.57 (−20 %)
uniform, $\lambda = 1$	1.69, 3.13	1.74 (+3 %)	3.32 (+6 %)
uniform, $\lambda = 0.5$	1.63, 3.95	1.61 (−2 %)	3.69 (−7 %)

The ripeness columns interpolate between its runs at targets 0.65, 0.7, 0.75, 0.8, and 0.85.

Skew is where ripeness gains, and for the reason the draft gives. The churn floor cleans a page only once it is half empty; a page whose hot content has died and whose cold content stays freezes above that, and keeps its garbage. In main’s 64 MiB skewed run, data pages lose fill for most of the run, down to 0.70; in ripeness’s, they hold at 0.76, since a frozen page, draining at the floor rate, is ripe at a fill a draining page never is.

Under uniform overwrites every page drains alike, and ripeness reduces to a single fill threshold, like the churn floor. What remains between the curves is noise: a page’s estimate comes from its own sporadic losses, so pages that drain alike get different estimates, and a threshold that differs page by page costs a little at light cleaning.

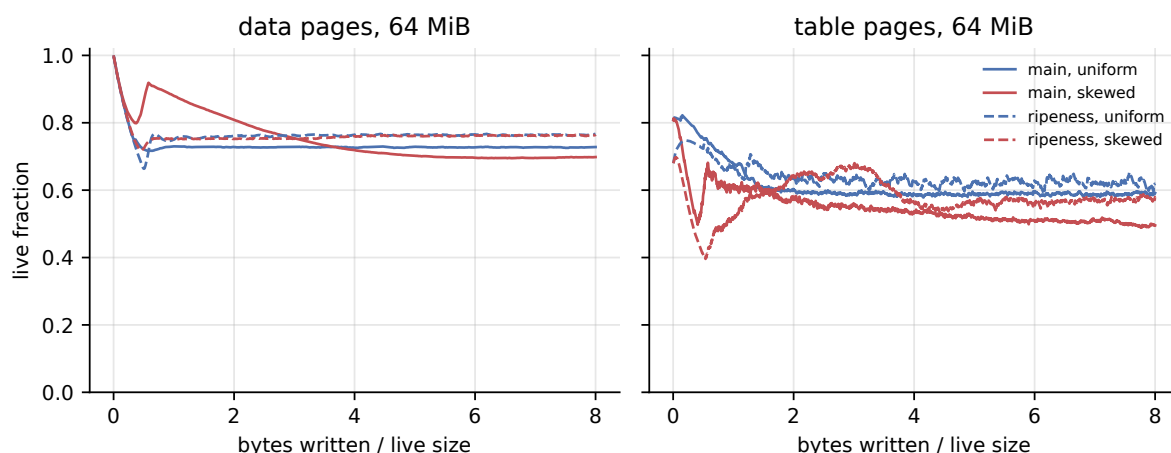


Figure 45.2: The live fraction of data pages and of address-table pages, in the 64 MiB files.

45.3 The controller

Measuring the whole file, the price controller swung by two to three decades; measuring the data pages and leaves, it holds within about 10 %. The draft's controller moves κ by $\exp(\text{gain} \cdot (\tau - \text{fill}))$ after every flush, as main's moves its budget. The first version's controller measured the whole file, like main's. But cleaning does not shrink the file at once: the pages it frees stay free until new writes reuse them, so the whole file's fill answers a change of price only many flushes later, and an integral controller on a lagging measure cycles. At 8 MiB and a target of 0.5, the price swung between $2 \cdot 10^{-4}$ and 0.09, a swing per one to two live sizes of writes, and at targets of 0.5 to 0.6 the files ended up 8 to 20 % larger than a fixed price gave for the same writes. The data pages and leaves, which cleaning acts on, answer within the flush. Measured there, the price holds within about 10 % of its median over the second half of every overwrite run of 8 MiB or more; in the first version, gains of 0.5 and 2 gave the same results, and the steady states fell on the curve that fixed prices traced. The holes the measure leaves out are compaction mode's to return anyway. [The consolidation page](#) suggests the same change for main's budget.

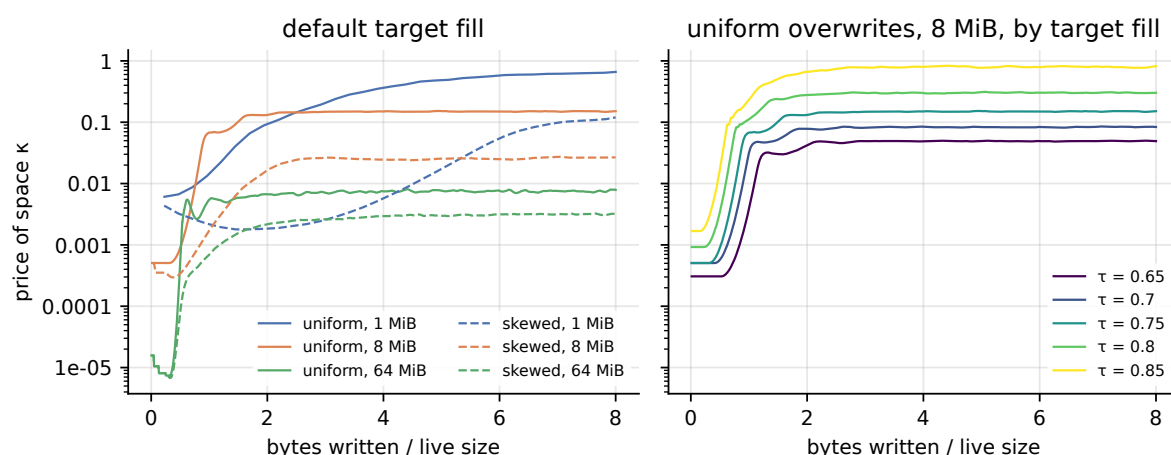


Figure 45.3: The price of space: the default runs (left), and the uniform 8 MiB runs by target (right).

The price also has to hold where moving it can change nothing. A file starts full, so without that the first flushes drove the price to its floor, and it took one to two live sizes of writes to climb back. The branch holds the price after a flush that cleaned nothing while the fill is above target, and after one that spent its whole budget while the fill is below; the clarified draft asks for the second.

That is not enough after a mass free. Between the append workload's rotations, the live pages sit above

target, and the price drifts down to $3 \cdot 10^{-5}$ even so, since small moves by free filling count as cleaning. When the logs rotate, the price needs 13 to 19 flushes to climb three decades, and the file peaks at 4.3, 3.5 and 2.9 times its live size, where main's peaks at 2.9, 2.3 and 2.3.

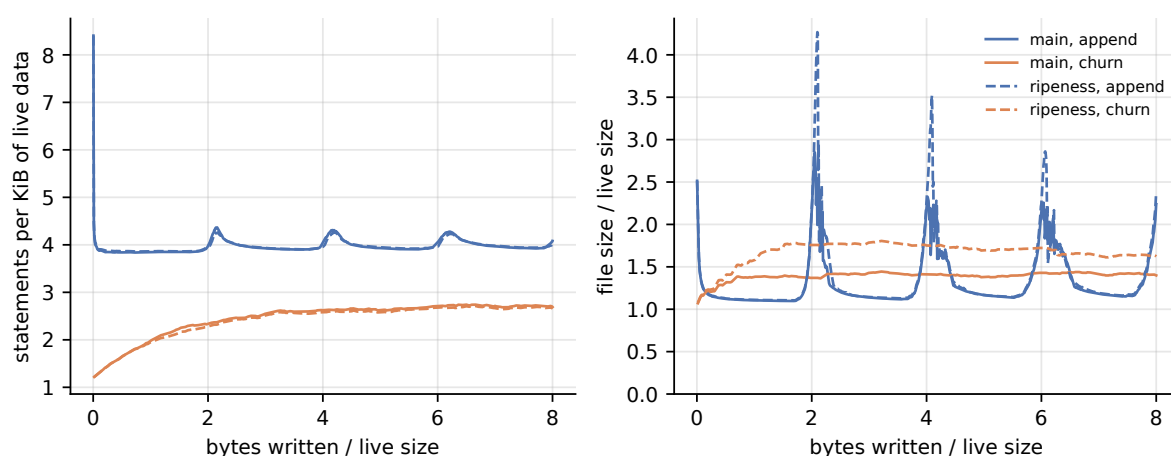


Figure 45.4: Statements per KiB of live data, and file size over live size, under appends and churn.

45.4 At the default settings

Ripeness at its default target writes less than main at its default under skewed overwrites and appends, and more under uniform overwrites, where its target lies further along the same curve.

steady state, file size / live size and bytes written per byte	main	ripeness	change
uniform, 8 MiB	1.69, 3.13	1.66, 3.44	−2 %, +10 %
uniform, 64 MiB	1.53, 3.26	1.48, 3.99	−4 %, +23 %
skewed, 8 MiB	1.65, 2.74	1.56, 2.44	−5 %, −11 %
skewed, 64 MiB	1.53, 3.15	1.43, 2.84	−6 %, −10 %
append, 16 MiB	1.38, 3.14	1.44, 2.96	+4 %, −6 %
churn, 16 MiB	1.41, 2.18	1.68, 2.12	+19 %, −3 %
typed, 16 MiB	1.53, 3.04	1.46, 3.81	−5 %, +25 %

The 1 MiB files are dominated by the quarantine, as on [the consolidation page](#), and their 33 flushes are too few for either controller to settle; the skewed one comes out 11 % larger with ripeness.

Under churn, ripeness's file is 19 % larger, and neither a higher target nor more generous free filling closes the gap. Main's churn run keeps data pages 91 % full, taking half of its victims for free in the room of pages the flush writes anyway, and moves 14 MB of survivors against ripeness's 4 MB. In the first version, at a target of 0.85, ripeness's data pages reached 0.88, but its file still kept a third of the live size in free pages, against main's quarter, and ended 9 % larger for about the same writes. In experiments on the first version, letting free filling take any ripe victim that fits, or any victim at all, changed nothing, or made things worse; why churn leaves ripeness with more free pages remains open.

After three quarters of a file are freed, compaction mode returns the space as fast as on main, but stops earlier. The file reaches 1.29 times its live size within 14 flushes, where main's reaches 1.14 within 16. Both stop once holes fall below a quarter of the file; ripeness's run crossed that line with 20 % holes left, main's overshot to 11 % in its last step.

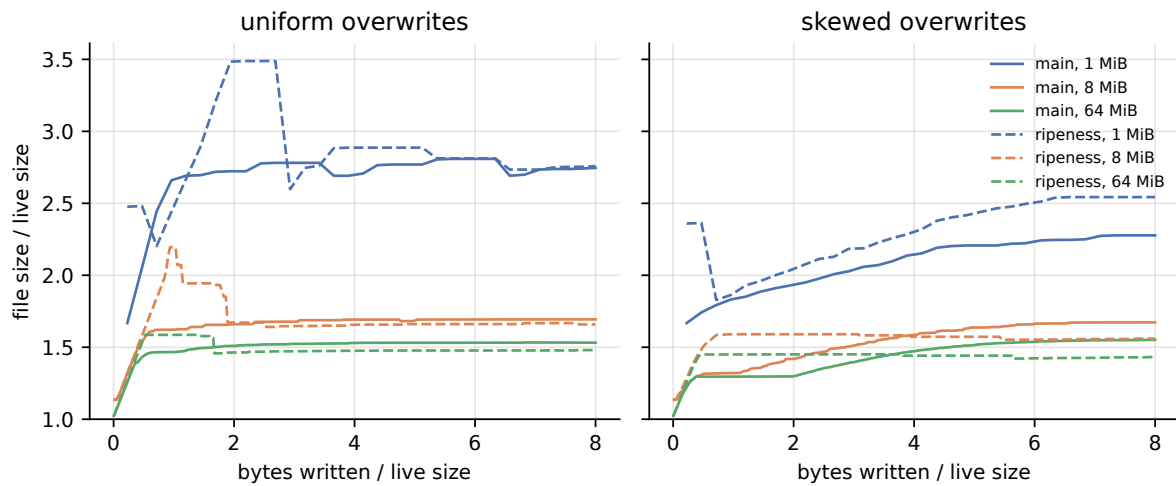


Figure 45.5: File size over live size as the overwrites accumulate.

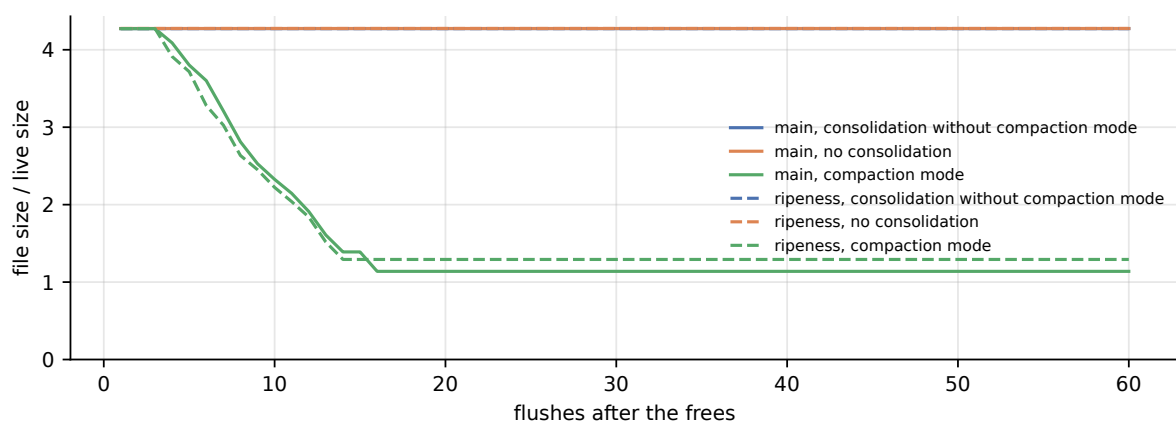


Figure 45.6: File size over live size after three quarters of the file were freed.

45.5 Hot, cool, and cold allocations on shared pages

When hot, cool, and cold allocations share pages at random, ripeness gains over main about as much as under skewed overwrites, a file 4 to 5 % smaller for 9 % fewer writes, but neither policy separates the cold content from the rest. kladde-bench's mixed workload is skewed with each allocation's class drawn at random. A tenth of the allocations are hot and take nine tenths of the writes; the other writes go to any allocation alike, as under skewed overwrites, except that the 45 % of allocations that are cold never change, so writes drawn for them are not made. The allocations are 1 KiB, four to a page, so that the pages written at the start mix classes, 91 % of the live data at 8 MiB. Skewed overwrites mix far less: their allocations of 16 KiB fill pages of their own, and their hot allocations hold the lowest ids, so a flush, which packs in key order, writes them apart from the rest. Each row of the workload's table records how much of the live data, and how much of the cold data, lies on pages that mix classes.

steady state, file size / live size and bytes written per byte	main	ripeness	change
mixed, 8 MiB	1.64, 2.70	1.56, 2.47	−5 %, −9 %
mixed, 64 MiB	1.51, 3.15	1.44, 2.85	−4 %, −9 %

Under skewed overwrites, ripeness gains 5 to 6 % in file size and 10 to 11 % in writes. At 1 MiB, where the quarantine dominates, ripeness's mixed file is 17 % larger, as its skewed one is 11 % larger.

Main's data pages freeze here as they do under skewed overwrites. Main's data pages lose fill for the whole run, from about 0.85 early on to 0.71 at its end, since a page whose hot content has died keeps its cool and cold content above the churn floor; ripeness holds its data pages at 0.77. The file's size moves in steps, when freed pages come to lie at its end, where truncation returns them, so the space its live pages take shows the difference better: 1.39 times the live size under ripeness, against 1.47 to 1.48 under main.

Neither policy separates the cold content from the rest. At the start, 91 % of the cold data shares a page with content that still changes. Under ripeness, 90 % still does at the end of the 8 MiB run and 88 % at the end of the 64 MiB one, as 89 % does without any cleaning; under main, the share falls slowly, to 85 and 84 %, most likely as the cool content of frozen pages dies away and leaves some of them holding cold content alone. Ripeness cleans the mixed pages that main leaves frozen, but packs their survivors together again: once a page's hot content has died, what survives on it is cool content, which still dies, next to cold content, which never will, and nothing tells the two apart. The draft's "cold with cold" assumes that what survives has stopped dying; here it may only be slow.

45.6 Without the logarithm

Ranking pages by the myopic $(1-x)/x$ in place of $h(x)$ writes 3 to 6 % more for the same file size under skewed overwrites, as the draft's ablation predicts, and 1 to 4 % more under uniform ones, where it predicts no difference. The ablated branch differs from the full one in nothing but the fill term of the index, and its controller aims at the same targets. Under uniform overwrites, too, pages do not all drain at one rate: leaves drain at a rate of their own, and each page's estimate strays from its true rate.

8 MiB, steady state	ripeness	myopic, same file size
skewed, $\tau = 0.7$	1.67, 2.32	2.41 (+4 %)
skewed, $\tau = 0.75$	1.56, 2.44	2.55 (+5 %)
skewed, $\tau = 0.8$	1.48, 2.63	2.78 (+6 %)
uniform, $\tau = 0.7$	1.75, 3.12	3.16 (+1 %)
uniform, $\tau = 0.75$	1.66, 3.44	3.52 (+2 %)
uniform, $\tau = 0.8$	1.60, 3.94	4.09 (+4 %)

At the default target, the 64 MiB files come out the same size either way, and the myopic rule writes 5 % more under skewed overwrites and 4.5 % more under uniform ones. Under appends it writes 1.6 % more, under churn 1.5 % more for a file 1 % smaller, and after the mass free it compacts exactly as the full rule does.

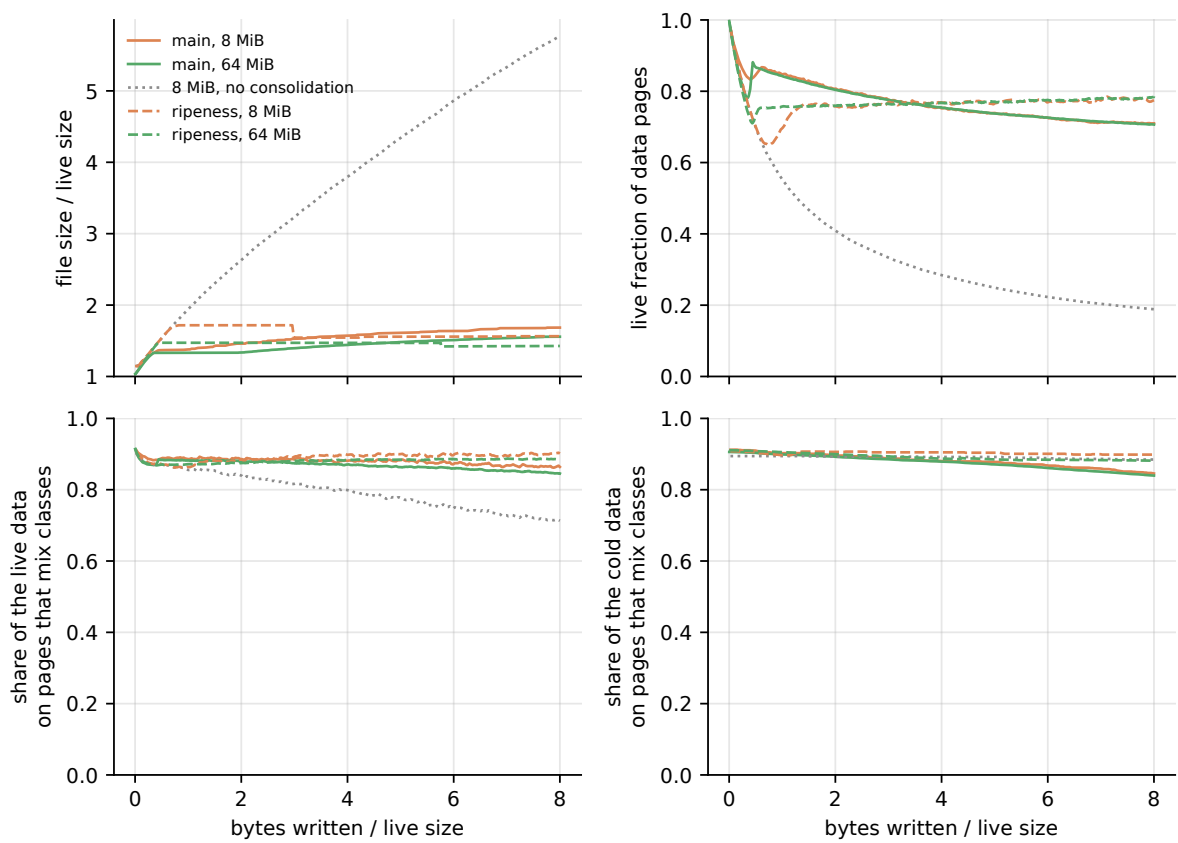


Figure 45.7: The mixed workload under main and ripeness: file size over live size, the live fraction of the data pages, the share of the live data on pages that mix classes, and the share of the cold data on such pages.

The extra writes are survivors, and they are few because survivors are a small part of what a flush writes. Under skew, the myopic rule moves 30 % more survivors per byte the application wrote: 0.44 bytes against 0.34 at 8 MiB, 0.59 against 0.45 at 64 MiB. The draft’s cost table for content that drains slowly, up to seven times the optimum, applies to that share alone, and the flush’s own content, which both rules write alike, makes up most of the rest.

The myopic rule also moves cleaning from leaves to data pages. Under skew, the leaves come out at 0.60 against 0.64 at 8 MiB, and 0.46 against 0.56 at 64 MiB, while the data pages gain a little. That fits the draft’s warning: the myopic rule ranks slowly draining pages higher, and they take writes that would free more elsewhere. Its controller settles the price three to five times lower, since $(1 - x)/x$ exceeds $h(x)$ at every fill.

45.7 What it costs

Keeping every page’s estimate in the consolidator state costs 1 to 7 % of the bytes written, and buys nothing within one session. The draft has the state record the estimate of every page the fold drains, which at 1000 operations per flush means an entry for about every page those operations touched, and a snapshot of every page below u_0 whenever the records outgrow the last one. Without the state, the 64 MiB runs write 7 % less under both uniform and skewed overwrites, for files 1 % smaller; append writes 3 % less, the 8 MiB runs 2 to 3 %, churn 1 %. The estimates matter only across a reopen, which these runs do not exercise; a denser encoding, or recording only pages whose estimate moved by a margin, would cut the cost.

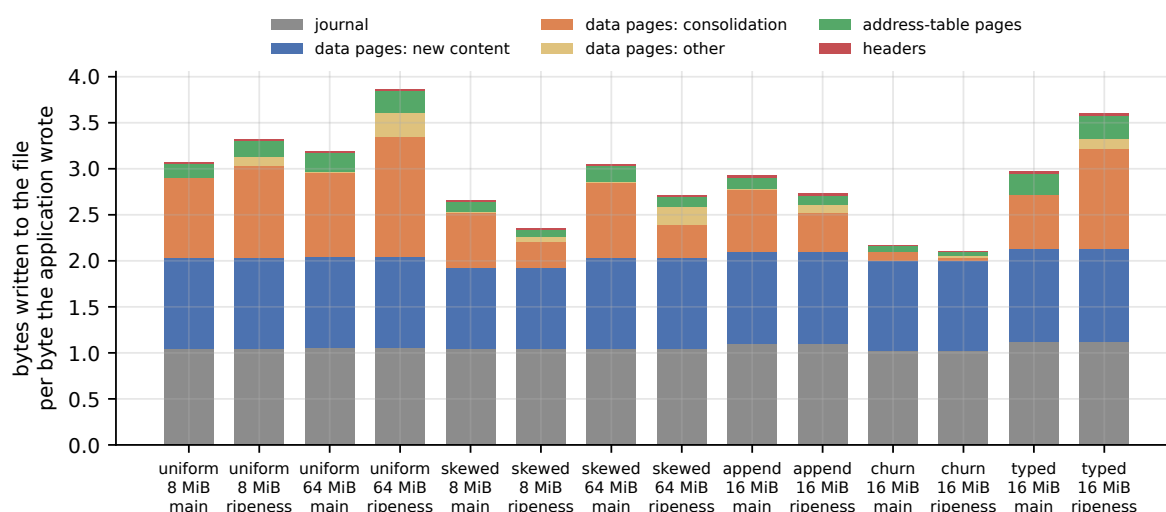


Figure 45.8: Where the bytes written to the file went, per byte the application wrote, over each whole run. On ripeness, “other” is mostly the consolidator state.

A flush takes time in proportion to the pages it writes, on either branch. These times are from the first version’s run, made back to back with main’s; the pages each writes per flush have changed by 3 % at most since. Per page written, ripeness’s flushes took between 4 % less and 16 % more time than main’s in the same session, within the spread identical runs showed before. Where ripeness writes fewer pages, as under skew, its flushes are faster: at the median over the second half of each run, 9.7 ms against 11.6 for the 8 MiB skewed file, 16.5 against 17.7 at 64 MiB. Where it writes more, as under uniform overwrites at its default target, they are slower: 34.9 ms against 27.0 at 64 MiB.

In memory, each page’s estimate takes 16 bytes in the page table, and each ranked page takes an entry in one of two ordered sets and one in a map from page to entry. The ranking takes $O(\log P)$ per page that changed and per victim, where main’s bucket queues take $O(1)$; the flush times do not show it.

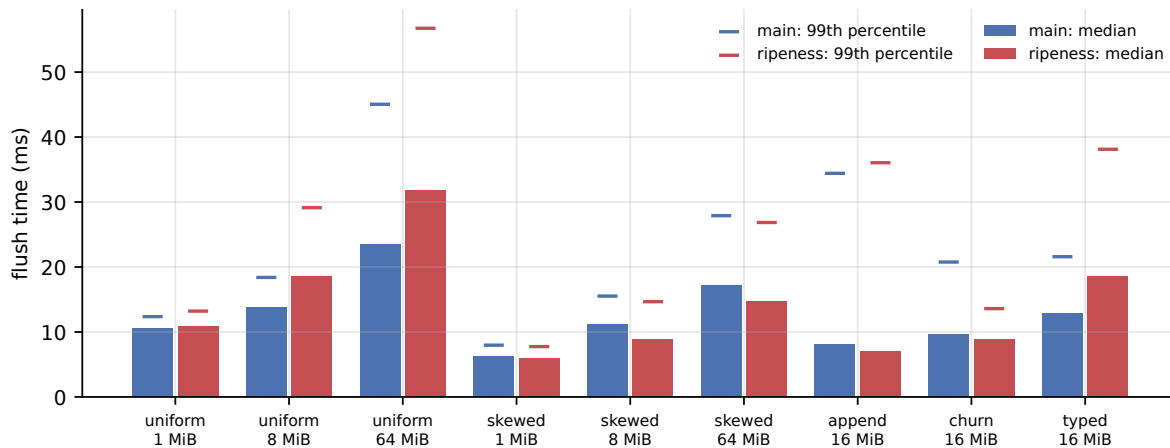


Figure 45.9: Flush times, including the fsync, in the first version's run: each run's median, and its 99th percentile.

45.8 What to change in the draft

- **Measure the fill the price acts on:** the data pages and leaves, not the whole file, whose fill lags a change of price by many flushes.
- **Hold the price where moving it can change nothing, in both directions.** The clarified draft holds it while the budget binds; it should also hold it while the fill is above target and nothing is cleaned, and keep it from sinking in calm phases, where every page is fuller than the target: a floor well above 10^{-6} , or holding the price whenever the budget took no ripe page, would spare the file the peaks after a mass free.
- **Record estimates more cheaply.** A drain record could leave out the estimate's epoch, which is the record's own, store the rate in a byte on a logarithmic scale, and skip pages whose estimate moved by less than a margin.
- **Keep the logarithm.** Without it, skewed overwrites cost 3 to 6 % more writes for the same file size, and nothing is simpler for it.
- **Look into churn,** the one workload where ripeness loses: replaced values free whole allocations every flush, and ripeness leaves more of the file free than main does.
- **Do not count on survivors being cold.** On the mixed workload, survivors that still die slowly share the pages opened for them with survivors that never will, and the cold content never ends up on pages of its own. Ordering survivors by their source page's estimate, or by their allocation's last_written, might separate them; this page has not tried it.
- **Expect noise under uniform overwrites.** Estimates from sporadic losses differ between pages that drain alike; a larger β or two rates per page, as the draft's open questions consider, trades that noise against how soon a frozen page is recognised.

45.9 Reproducing

The runs of the branch used one checkout of kladde-rs, with main in a worktree of its own:

```
git worktree add ../kladde-main main
cargo build --release -p kladde-bench --manifest-path ../kladde-main/Cargo.toml
git checkout ripeness
cargo build --release -p kladde-bench
../kladde-main/target/release/kladde-bench /tmp/bench-main
./target/release/kladde-bench /tmp/bench-ripeness
KLADDE_BENCH_NO_STATE=1 ./target/release/kladde-bench /tmp/bench-ripeness-no-state uniform skewed appe
KLADDE_BENCH_MYOPIC=1 ./target/release/kladde-bench /tmp/bench-ripeness-myopic uniform skewed append c
```

The mixed workload ran from the branch at 799c661, with and without the consolidator state, and on

main from the branch main-mixed at 7646408, which is main with the workload added:

```
git checkout 799c661
cargo build --release -p kladde-bench
./target/release/kladde-bench /tmp/bench-ripeness mixed
KLADDE_BENCH_NO_STATE=1 ./target/release/kladde-bench /tmp/bench-ripeness-no-state mixed
git checkout main-mixed
cargo build --release -p kladde-bench
./target/release/kladde-bench /tmp/bench-main mixed
```

In kladde-docs, keep the tables gzipped under content/evaluation/data/ripeness/, in main/, ripeness/, ripeness-no-state/, and ripeness-myopic/, and draw the figures from them, all but the flush times, which come from the tables of commit 08febdf:

```
D=content/evaluation/data/ripeness
tools/plot-evaluation.py --summary --out content/evaluation/figures/ripeness \
    --only space-amplification, live-fraction, write-breakdown, budget, shrink, description, defrag-share, ka
    main=$D/main ripeness=$D/ripeness
tools/plot-evaluation.py --only tradeoff --out content/evaluation/figures/ripeness \
    main=$D/main ripeness=$D/ripeness myopic=$D/ripeness-myopic
tools/plot-evaluation.py --only mixed --out content/evaluation/figures/ripeness main=$D/main ripeness=
```

Chapter 46

Ripeness with a static share

The static share makes no difference to cleaning on kladde-bench’s workloads, not even on one built for it, and costs twice as much to keep. Without the consolidator state, the static-share branch and the single-rate branch that [Cleaning by ripeness](#) evaluates write within 0.3 % of each other in every run of 8 MiB or more, and their live pages take the same space to within 0.2 %. With the state, the static-share branch writes 2 to 7 % more at the default target, and all of the difference is its state’s larger entries. Few pages earn a static share, even on the mixed workload, where content that never changes makes up 45 % of the live data and nine tenths of it shares pages with content that does: 1.9 % of the live bytes count as static there at 8 MiB, and almost none at 64 MiB. The fall that would show a static share, a page’s hot content dying, comes to a few statements of about 250 bytes, too little evidence for the test to let a static share through, and afterwards a page loses only the rare statements of its cool content, which one share draining slowly explains as well. The other workloads give the fit less still to find: their cold content still dies, only slowly, and skewed overwrites seldom put hot and cold content on one page. That matches what the draft’s own simulation found: the single-rate draft cleans as cheaply, and needs half the state.

46.1 What was measured

The [same benchmark](#) ran on the `ripeness2` branch of `kladde-rs`, which implements [the draft](#) as of `kladde-docs` commit `7c343b7`. The branch is at `074a0df`, on top of the single-rate branch at `362210a`; the two differ only in how a page’s estimate is kept and fitted, and in two counters of static shares, which the benchmark records. The benchmark is deterministic but for its times, so every number of either branch’s runs compares with the other’s: the single-rate branch’s and main’s tables are those of [Cleaning by ripeness](#). A second run of the static-share branch left the consolidator state out, as one of the single-rate branch did, to tell what the policy does from what its state costs. The mixed workload ran later on both branches, with and without the state, at `a856b06` here and `799c661` on the single-rate branch, which add it to `kladde-bench` and change nothing else. This page compares no times, since its runs shared their host with other work.

The branch takes the draft’s constants, $\beta = 0.1$ per flush, a starting estimate worth $n_0 = 10$ flushes of losses, and a test that a static share must pass by $c = 3$ statements’ worth of log-likelihood, and the single-rate branch’s for the rest. Its fit agrees with the tested estimator of `tools/simulate-ripeness.py` to within 0.1 % of the index, where the simulation fits rates on a fine grid, and reproduces the draft’s worked example. `implementation-notes.md` in `kladde-rs` lists where it departs from the draft: both fits are solved exactly, and the test’s unit, the mean statement size, is measured separately for data pages and leaves.

46.2 How often pages earn a static share

Rarely, even where content that never changes shares pages with content that does: at most 2.6 % of the live bytes count as static, on average over the second half of any run of 8 MiB or more.

second half of each run	pages with a static share	live bytes estimated static
uniform, 8 MiB	0.15 %	0.12 %
uniform, 64 MiB	0.16 %	0.14 %
skewed, 8 MiB	2.6 %	2.3 %
skewed, 64 MiB	0.57 %	0.66 %
mixed, 8 MiB	4.3 %	1.9 %
mixed, 64 MiB	0.06 %	0.05 %
append, 16 MiB	2.4 %	2.6 %
churn, 16 MiB	0.53 %	0.13 %
typed, 16 MiB	0.70 %	0.60 %

The mixed workload is built for a static share, and the fit finds almost none of it. Its allocations, four to a page, are hot, cool, or cold at random, and the cold ones, 45 % of the live data, never change. Nine tenths of the cold data share a page with content that still changes, 90 % at 8 MiB and 88 % at 64 MiB, on both branches alike. Yet at most 19 of the 8 MiB file's 2098 data pages hold a static share during the first flushes, while the hot content of the pages written at the start dies, and at most 2.4 % of the live bytes count as static at any time; at 64 MiB, never more than 0.2 %.

A page's fall is too small to earn a static share. A page written at the start holds four allocations of 1 KiB, usually one hot one at most, and that allocation dies within a few flushes: the whole fall, the evidence for a static share, comes to about four statements of 250 bytes, too little for the three statements' worth of log-likelihood the test asks for. Afterwards the page loses only its cool content, a statement every 40 flushes or so, which one share draining slowly explains as well as a draining share over a static one. The test counts in statements, and a statement in these files states 100 to 380 bytes of data, so a data page holds 10 to 30 of them. The draft's simulation found the same: with statements of 32 to 256 bytes, its tested fit costs as much as the single-rate draft, and with larger ones a little less, since the test that keeps it from inventing static shares also keeps it from finding them quickly. A leaf states some 300 statements of 10 to 14 bytes each, plenty of losses, but its content, statements that the fold supersedes, is no more static than the data they state.

The other workloads offer less to find. Their cold content is not static, only slow: under skewed overwrites, a tenth of the writes go to all allocations alike, so a cold byte of the 8 MiB file dies at about 0.003 per flush, against a price of space of 0.025, a ratio r/κ of about 0.12, at which the draft's threshold lies at 0.64. The 64 MiB file's cold content dies eight times slower, and the price settles eight times lower, at the same ratio. And skewed overwrites seldom put hot and cold content on one page: their hot allocations hold the lowest ids, their allocations of 16 KiB fill pages of their own, and a flush packs its writes in key order, so that of the 60 or so pages of new content a flush writes at 8 MiB, one mixes hot with cold.

46.3 Cleaning

Without the consolidator state, the two branches reach the same steady states.

steady state, no consolidator state	single rate: file size / live size, bytes written per byte	static share	change
uniform, 8 MiB	1.652, 3.342	1.650, 3.345	-0.1 %, +0.1 %
uniform, 64 MiB	1.467, 3.707	1.467, 3.703	0.0 %, -0.1 %
skewed, 8 MiB	1.548, 2.394	1.547, 2.397	-0.1 %, +0.1 %
skewed, 64 MiB	1.415, 2.648	1.417, 2.645	+0.1 %, -0.1 %
mixed, 8 MiB	1.660, 2.398	1.606, 2.403	-3.2 %, +0.2 %
mixed, 64 MiB	1.412, 2.638	1.412, 2.636	0.0 %, -0.1 %
append, 16 MiB	1.420, 2.864	1.424, 2.854	+0.3 %, -0.3 %
churn, 16 MiB	1.655, 2.088	1.655, 2.093	0.0 %, +0.2 %

The mixed 8 MiB files differ in size only because their free pages reached the file's end, where truncation returns them, at different times; in every run here, the live pages take the same space on both branches

to within 0.2 %, 1.38 times the live size in the mixed ones. The 1 MiB files, dominated by the quarantine and too short for the controller to settle, differ by up to 2 %.

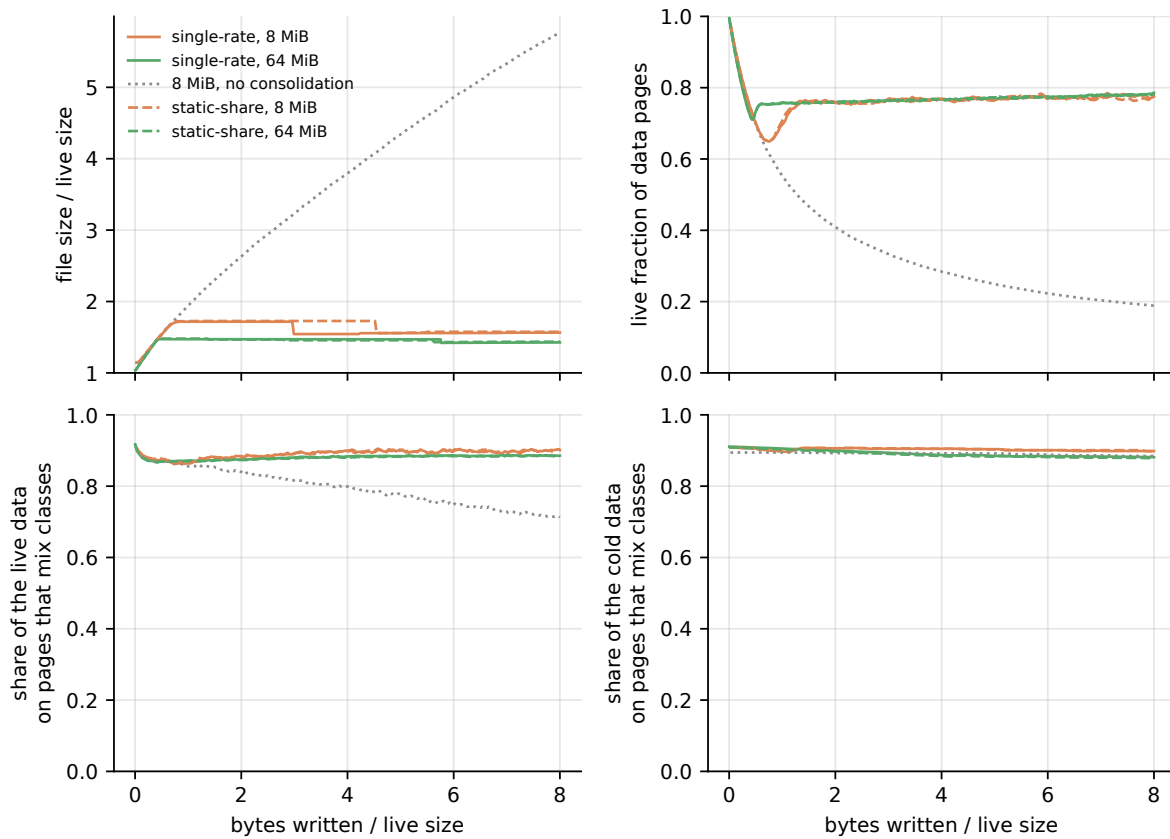


Figure 46.1: The mixed workload on both branches: file size over live size, the live fraction of the data pages, the share of the live data on pages that mix classes, and the share of the cold data on such pages.

With the state, the static-share branch writes 2 to 7 % more, and its trade-off curve lies 3 to 7 % to the right of the single-rate branch's.

steady state, with the state	single rate	static share	change
uniform, 8 MiB	1.66, 3.44	1.68, 3.55	+1 %, +3 %
uniform, 64 MiB	1.48, 3.99	1.49, 4.29	+1 %, +7 %
skewed, 8 MiB	1.56, 2.44	1.57, 2.51	+1 %, +3 %
skewed, 64 MiB	1.43, 2.84	1.45, 3.04	+1 %, +7 %
mixed, 8 MiB	1.56, 2.47	1.59, 2.53	+2 %, +3 %
mixed, 64 MiB	1.44, 2.85	1.44, 3.06	0 %, +7 %
append, 16 MiB	1.44, 2.96	1.44, 3.05	0 %, +3 %
churn, 16 MiB	1.68, 2.12	1.67, 2.15	-1 %, +2 %
typed, 16 MiB	1.46, 3.81	1.47, 4.02	+1 %, +6 %

On the mixed workload, against main, which [Cleaning by ripeness](#) compares there, the static-share branch keeps less of the single-rate branch's gain in writes, 6 % fewer at 8 MiB and 3 % at 64 MiB, against 9 % at both; its live pages take 5 to 6 % less space than main's, as the single-rate branch's do. At the targets from 0.65 to 0.85, the 8 MiB files write 3 to 6 % more under skewed overwrites and 5 to 7 % more under uniform ones for the same file size. Main's curve stays where [Cleaning by ripeness](#) puts it: under skew, both ripeness branches lie below it.

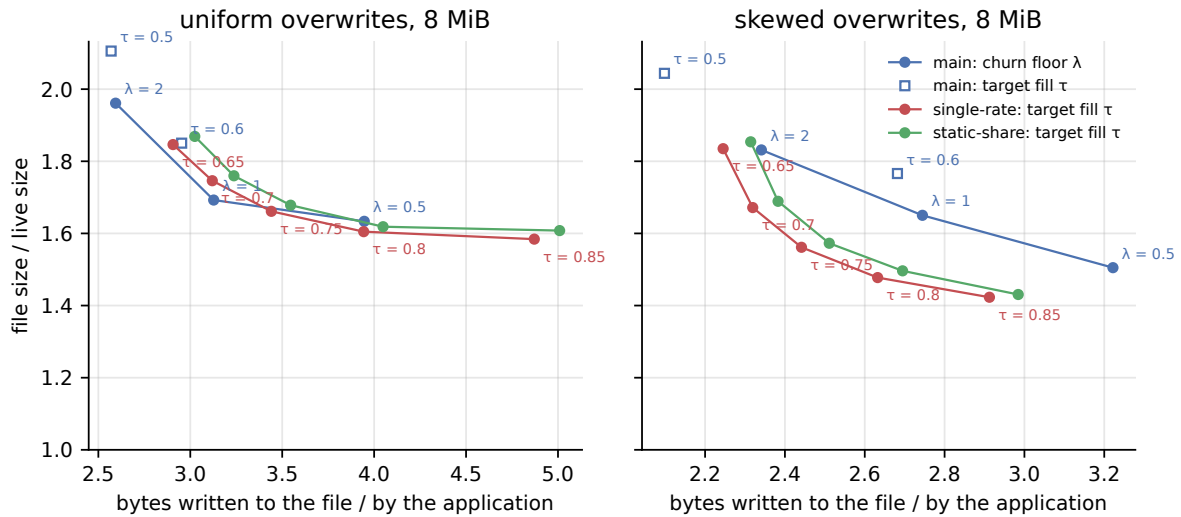


Figure 46.2: File size against bytes written in the steady state of the 8 MiB overwrite runs: main's churn floors and targets, and the two ripeness branches' targets, labelled where the single-rate branch's are.

46.4 What it costs

The static-share branch's consolidator state costs twice what the single-rate branch's does: 3 to 14 % of the bytes written, against 1 to 7 %. An entry holds the fit's two sums, its rate, and its share, four f32s where the single-rate branch's holds one rate, and the draft has the state record every page the fold drains. At 64 MiB, the state takes 14 % of the writes under uniform overwrites and 13 % under skewed ones, against 7 %; at 8 MiB, 6 and 5 %, against 3 and 2 %; under appends 6 % against 3 %, and under churn 3 % against 1 %. It also takes space: with it, the live pages hold up to 1.1 % more than without, where the single-rate branch's adds up to 0.7 %.

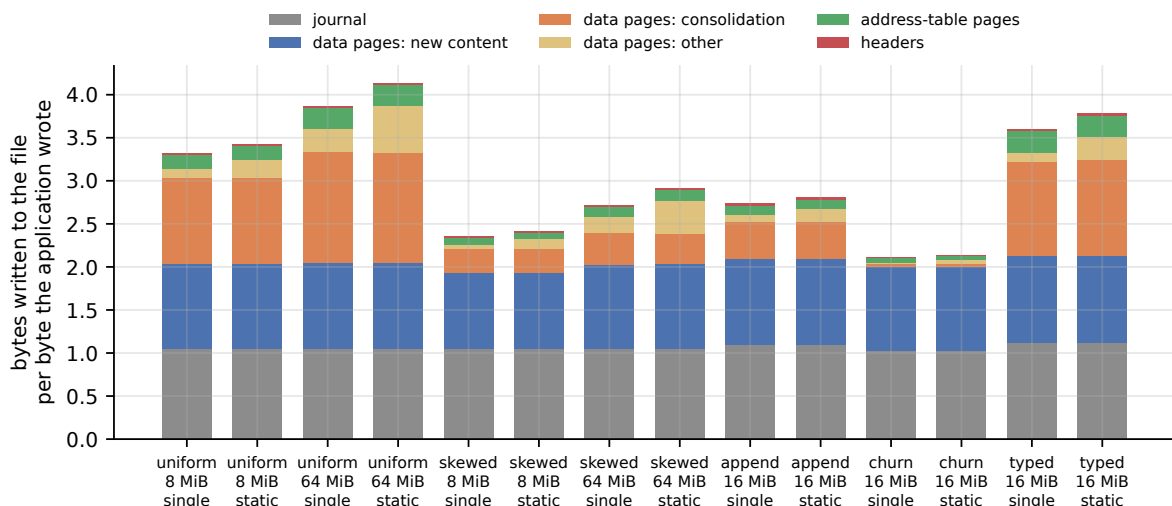


Figure 46.3: Where the bytes written to the file went, per byte the application wrote, over each whole run, on the single-rate branch and the static-share branch. "Other" is mostly the consolidator state.

In memory, a page's estimate takes 24 bytes, against 16. The ranking keeps its two ordered sets, and a page's entry also holds the key of its floor, which the single-rate branch derives from its estimate.

Each flush fits every page that lost content, with two bisections of 50 steps, each step a handful of exponentials and logarithms. The runs shared their host with other work, so this page does not say what that costs in time.

46.5 What to change in the draft

- **Keep one rate per page:** on kladde-bench's workloads, including one where 45 % of the data never changes and shares pages with data that does, the static share changes nothing, and doubles the state.
- **Expect a static share to pay only where a page's losses are plentiful.** The evidence for one is a page's fall, and a page of statements of 250 bytes shows a fall of a few statements at most, too little for the test; the draft's simulation found a static share paying only on small statements, and then only without the test.
- **If the static share stays, record less of it.** A page's rate and share follow from its two sums, its age, and its coverage, which an entry holds already, so they could be fitted again at open, for 8 bytes an entry less.
- **Say which statements the test counts in.** The draft speaks of the file's mean statement size, but a data page loses the bytes statements state, 100 to 380 per statement in these files, and a leaf their encoding, 10 to 14.

46.6 Reproducing

The static-share branch ran from one checkout of kladde-rs, at 074a0df, and the mixed workload later, at a856b06; at a856b06, these commands run all of it:

```
git checkout ripeness2
cargo build --release -p kladde-bench
./target/release/kladde-bench /tmp/bench-ripeness2
KLADDE_BENCH_NO_STATE=1 ./target/release/kladde-bench /tmp/bench-ripeness2-no-state uniform skewed mix
```

In kladde-docs, keep its tables gzipped under content/evaluation/data/ripeness2/, in ripeness2/ and ripeness2-no-state/, and draw the figures with those of [Cleaning by ripeness](#):

```
D=content/evaluation/data
tools/plot-evaluation.py --only tradeoff --out content/evaluation/figures/ripeness2 \
    main=$D/ripeness/main single-rate=$D/ripeness/ripeness static-share=$D/ripeness2/ripeness2
tools/plot-evaluation.py --only write-breakdown --out content/evaluation/figures/ripeness2 \
    single=$D/ripeness/ripeness static=$D/ripeness2/ripeness2
tools/plot-evaluation.py --only mixed --out content/evaluation/figures/ripeness2 \
    single-rate=$D/ripeness/ripeness static-share=$D/ripeness2/ripeness2
```

tools/simulate-ripeness.py example prints the draft's worked example, which the unit test the_fit_follows_the_drafts_example in kladde-rs checks the fit against.

Chapter 47

Bayesian ripeness

On kladde-bench's workloads, a posterior per chunk cleans no better than one rate per page, under either rule, and costs more to keep and to rank. The [Bayesian draft](#)'s cure model, started from the file's empirical prior, ran on a branch of kladde-rs, deciding by the expected gain, rule (a), and by the option to wait, rule (c'). Without the consolidator state, its live pages take the same space as the single-rate branch's to within 0.3 % in every run of 8 MiB or more. It writes as much, to within 0.6 %, under uniform overwrites, on the mixed workload, and under churn, but 0.5 to 1.1 % more under skewed overwrites and 4 to 5 % more under appends. Rules (a) and (c') write within 0.7 % of each other without the state, and within 1.2 % with it: (c') is (a) in effect, as the draft's simulation found. The consolidator state costs 2.5 to 12.6 % of the bytes written, nearly the static-share branch's 2.7 to 13.8 %, against the single-rate branch's 1.4 to 7.4 %. And ranking, a mixture over up to some 300 chunks per page, first took three quarters of the flushes' instructions, which came to four times the single-rate branch's; with the index found by regula falsi on logarithms, ranking takes a third of them, and the flushes 1.6 times the single-rate branch's instructions and half the static-share branch's, as callgrind counts them on the uniform workload at 1 MiB.

The posterior finds static content where the static-share fit found little, and cleaning cannot use it. On the mixed workload, where 45 % of the live data never changes, it expects 6 % of the live bytes to be static, against the fit's 2 % at most; yet nine tenths of the cold data share a page with data that changes, on every branch alike, as they did when the file was filled: cleaning moves all of a page's survivors, cold and cool alike, onto pages of their own with other pages' survivors, so which pages it cleans does not separate them. Where the content it calls static only lives long, logs until they rotate or cold data that still dies slowly, it cleans earlier and moves 5 to 18 % more bytes, which make up most of its extra writes.

The file's empirical prior is sure of what it has not seen. Where fresh pages lose content early, it comes out as the draft intends: sure under uniform overwrites, where they all drain alike, and weak under skewed ones, where they differ. But where they have lost nothing, as in the fill phase that starts the overwrite workloads and churn, or under appends, whose logs rotate long after a page's first flushes, it comes out at the floor rate with the weight of 1000 loss events, which a page's own losses take some hundred flushes to outweigh. The pages written in the fill phase then look static while they drain: in the first ten flushes of the skewed and mixed workloads at 8 MiB and of churn, the posterior cleaned 22 to 132 pages where the other branches cleaned three at most, and on the mixed workload, the file's peak, and with it its size for the rest of the run, came out 11 % larger.

47.1 What was measured

The [same benchmark](#) ran on the [bayesian-ripeness](#) branch of kladde-rs, which implements [the draft](#) as of kladde-docs commit [369ae0e](#), in four configurations: rules (a) and (c'), each with and without the consolidator state. The branch is at [9f99a1f](#), on top of the static-share branch at [a856b06](#); it replaces the fit and its test with the posterior, and the consolidator state's entries with the posterior's, and changes nothing else that the benchmark exercises. The benchmark is deterministic but for its times, so every number of these runs but the times compares with those of the single-rate branch and main on [Cleaning by ripeness](#), and of the static-share branch on [Ripeness with a static share](#), whose tables this page reuses.

The branch takes the draft's constants, $\beta = 0.1$ per flush, a class prior worth $v_{\pi} = 10$ chunks, and $R_{\text{MIN}} = 10^{-4}$, and the draft's recommendations: the cure model per chunk, the empirical prior for all content, and the floor as a cap. The empirical prior learns from what the pages a flush writes lose in their first ten flushes, the posterior's memory; before it has two pages' worth, from the pages still young, and before any, it is the weakest prior the branch allows, 0.05 loss events over an exposure of one. `implementation-notes.md` in `kladde-rs` lists where the branch departs from the draft: a chunk's size is what its statement states, a Ref that the cut restates is a new, untouched chunk, and an Inline statement's death counts as two loss events.

This page compares the space the live pages take, as well as the file's size. The file's size also counts free pages that have not reached its end, where truncation returns them; the live pages, data pages and leaves, are what cleaning decides. Both are averaged over the second half of each run, and writes are the bytes written during that half per byte the application wrote, as on the other pages.

This page compares no times. The runs' host was on battery power, so their times say nothing about the branches. Where the page speaks of what ranking costs, it counts instructions with `callgrind`, on the one run small enough for it: `kladde-bench's quick uniform`, the uniform workload at 1 MiB, run by the single-rate branch at 799c661, the static-share branch at a856b06, the posterior at 9f99a1f, and the posterior with faster ranking at 08b575d.

47.2 Cleaning

Without the consolidator state, the posterior reaches the single-rate branch's steady states in space, and in writes on three workloads of five. Each cell gives the live pages over the live size, and the bytes written per byte the application wrote.

no state	single rate	static share	posterior, (a)	posterior, (c')	(a) against single rate
uniform, 8 MiB	1.429, 3.342	1.429, 3.345	1.429, 3.344	1.428, 3.349	0.0 %, +0.1 %
uniform, 64 MiB	1.432, 3.707	1.432, 3.703	1.432, 3.723	1.432, 3.714	0.0 %, +0.4 %
skewed, 8 MiB	1.385, 2.394	1.386, 2.397	1.386, 2.419	1.386, 2.415	+0.1 %, +1.0 %
skewed, 64 MiB	1.387, 2.648	1.387, 2.645	1.387, 2.677	1.388, 2.663	0.0 %, +1.1 %
mixed, 8 MiB	1.382, 2.398	1.381, 2.403	1.382, 2.389	1.382, 2.395	0.0 %, -0.4 %
mixed, 64 MiB	1.382, 2.638	1.383, 2.636	1.383, 2.639	1.383, 2.622	0.0 %, 0.0 %
append, 16 MiB	1.304, 2.864	1.306, 2.854	1.299, 3.003	1.301, 2.989	-0.3 %, +4.9 %
churn, 16 MiB	1.353, 2.088	1.353, 2.093	1.353, 2.090	1.354, 2.084	0.0 %, +0.1 %

The files differ more than their live pages: the posterior's mixed 8 MiB file is 1.85 times the live size, against the single-rate branch's 1.66, and its churn file 1.71 against 1.66, with 25 % and 21 % of their pages free against 17 % and 18 %. [The first flushes](#) account for the mixed file; the churn files peak within 0.5 % of each other and differ in when they truncate, and the others differ by less than 2 %.

With the state, the posterior writes 1 to 6 % more than the single-rate branch, nearly as much as the static-share branch, and its trade-off curve lies with the static-share branch's.

with the state	single rate	static share	posterior, (a)	posterior, (c')	(a) against single rate
uniform, 8 MiB	1.435, 3.441	1.442, 3.546	1.440, 3.532	1.440, 3.534	+0.4 %, +2.7 %
uniform, 64 MiB	1.439, 3.991	1.446, 4.286	1.444, 4.206	1.444, 4.257	+0.4 %, +5.4 %
skewed, 8 MiB	1.391, 2.441	1.399, 2.511	1.397, 2.513	1.398, 2.508	+0.5 %, +2.9 %
skewed, 64 MiB	1.394, 2.837	1.401, 3.036	1.399, 3.009	1.399, 3.015	+0.4 %, +6.1 %
mixed, 8 MiB	1.390, 2.468	1.395, 2.534	1.394, 2.532	1.394, 2.505	+0.3 %, +2.6 %

with the state	single rate	static share	posterior, (a)	posterior, (c')	(a) against single rate
mixed, 64 MiB	1.391, 2.852	1.398, 3.062	1.396, 3.022	1.396, 3.020	+0.4 %, +5.9 %
append, 16 MiB	1.305, 2.955	1.311, 3.045	1.310, 3.116	1.309, 3.141	+0.4 %, +5.4 %
churn, 16 MiB	1.357, 2.118	1.361, 2.153	1.361, 2.145	1.361, 2.139	+0.3 %, +1.3 %
typed, 16 MiB	1.419, 3.808	1.427, 4.018	1.425, 3.835	1.425, 3.835	+0.4 %, +0.7 %

At the targets from 0.65 to 0.85, the posterior's 8 MiB files hold their live pages in the static-share branch's space to within 0.1 %, and in 0.3 to 0.6 % more than the single-rate branch's. They write 2.6 to 5.9 % more than the single-rate branch's under uniform overwrites and 2.3 to 4.0 % more under skewed ones, from 1.3 % less to 3.0 % more than the static-share branch's, the more the higher the target; rule (c') writes as rule (a) does to within 1.5 %. The figure plots the files' sizes, which also count free pages: under skewed overwrites, at the targets 0.65 and 0.7, the posterior's files are 3.8 to 5.3 % larger than the single-rate branch's for live pages 0.6 % larger.

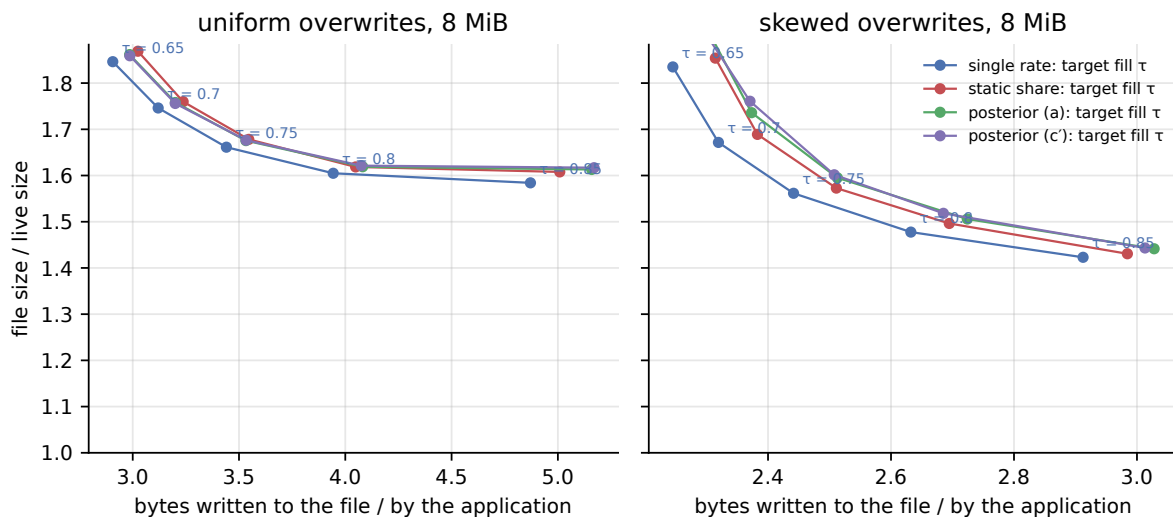


Figure 47.1: File size against bytes written in the steady state of the 8 MiB overwrite runs, with the state: the targets of the single-rate branch, the static-share branch, and the posterior under rules (a) and (c'), labelled where the single-rate branch's are.

Rules (a) and (c') clean alike. Without the state, they write within 0.7 % of each other in every run, and their live pages take the same space to within 0.1 %; with the state, within 1.2 %. The draft's simulation found the same: the predictive that (c') weighs later losses by trusts the posterior it is drawn from, so the option to wait is worth little where the expected gain is, and (c') cleans as (a) does.

47.3 What the posterior calls static

The posterior expects a static share on every workload, where the static-share fit found one on few pages, but mostly a small one. For the static-share branch, each cell gives the pages with a static share and the live bytes in those shares; for the posterior, the pages expected at least a tenth static and the live bytes expected static; all averaged over the second half of each run, with the state.

second half	static share	posterior, (a)	posterior, (c')
uniform, 8 MiB	0.15 %, 0.12 %	0.45 %, 2.4 %	0.44 %, 2.4 %
uniform, 64 MiB	0.16 %, 0.14 %	0.53 %, 4.2 %	0.52 %, 4.3 %
skewed, 8 MiB	2.6 %, 2.3 %	6.1 %, 5.8 %	5.8 %, 5.7 %
skewed, 64 MiB	0.57 %, 0.66 %	5.8 %, 6.1 %	5.6 %, 6.1 %

second half	static share	posterior, (a)	posterior, (c')
mixed, 8 MiB	4.3 %, 1.9 %	15 %, 6.6 %	13 %, 6.3 %
mixed, 64 MiB	0.05 %, 0.05 %	12 %, 6.0 %	12 %, 6.0 %
append, 16 MiB	2.4 %, 2.6 %	2.6 %, 4.9 %	2.7 %, 4.8 %
churn, 16 MiB	0.53 %, 0.13 %	2.2 %, 1.4 %	2.2 %, 1.3 %
typed, 16 MiB	0.70 %, 0.60 %	19 %, 7.0 %	19 %, 7.0 %

A page's expected static share is the mixture's weight on its untouched chunks being static; under uniform overwrites, where no content is static, it comes to 2 to 4 % of the live bytes, spread thinly: all but half a percent of the pages are expected less than a tenth static.

The mixed workload's cold data stays where it was, on pages that mix classes. Its allocations, four to a page, are hot, cool, or cold at random, and the cold ones, 45 % of the live data, never change. The posterior expects 6 % of the live bytes to be static, where 45 % are. A chunk counts as static for outliving what drains around it, and two things weaken that evidence here. The pages written in the fill phase start sure that they drain at the floor rate, as below, and at that rate an untouched chunk's survival says nothing about its class. And a page's hot content dies within a few flushes; once its losses stop, their weight fades at β per flush, and the class prior, which starts fresh content at all draining with the weight of ten chunks, takes over again. Whatever it finds, 88 to 90 % of the cold data share a page with data that still changes, on all three branches, with and without the state, much as 91 % did when the file was filled: a cold allocation's three neighbours are all cold with a chance of only 0.45³. Cleaning does not change that, since it moves all of a ripe page's survivors, and once a page's hot allocation has died, its survivors are its cold and its cool allocations alike. In the second half of the runs without the state, the survivors went almost all to pages opened for them alone, whose capacity they fill to within 1 %, with the survivors of other pages; the 2.5 to 9.3 % of the victims that rode in the room of pages the flush wrote anyway were small ones. Separating the cold data from the cool would take moving survivors by class, a chunk at a time rather than a page at a time.

Where the content the posterior calls static only lives long, cleaning it early costs writes. Under appends, logs are truncated to nothing when they reach 64 KiB, so a page loses nothing for hundreds of flushes and then all of a log's records at once. At the same price of space, about 10^{-4} on all branches, the posterior moves 0.65 bytes of survivors per byte the application writes, against 0.55 on the single-rate and static-share branches, 18 % more, for live pages 0.3 % smaller. Under skewed overwrites, whose cold data still dies, only slowly, it moves 5 to 7 % more: 0.369 and 0.475 bytes per byte at 8 and 64 MiB, against 0.345 and 0.451, for the same space. These are the runs without the state, rule (a); rule (c') moves as much to within 2.5 %.

47.4 The empirical prior

Where fresh pages lose content in their first flushes, the empirical prior comes out as the draft intends.

A replay of the uniform and skewed workloads at 4 MiB, which asked the store for its priors every 50 flushes, found the data pages' prior at 1000 loss events and a rate of 0.06 per flush under uniform overwrites, and the leaves' at 170 to 190 events and 0.04; under skewed overwrites, both at 0.3 to 0.6 events. Under uniform overwrites, fresh pages drain alike, the method of moments finds no spread beyond the Poisson noise, and the prior is as sure as the floor on its spread lets it be, a thousandth of its mean squared; under skewed ones, hot and cold pages differ, and the prior is weak.

Where fresh pages lose nothing, the empirical prior takes both of its floors, and is sure that pages hardly drain. With no losses, the mean rate is the floor R_{MIN} , and the spread the floor under it, so the prior weighs 1000 loss events at a rate of 10^{-4} , over an exposure of 10^7 events. The overwrite workloads and churn start with a fill phase in which nothing is lost; the replays found both priors, the data pages' and the leaves', at those floors when it ended, under uniform and skewed overwrites alike, so the pages written in it, most of the file, start from them. Discounted by β per flush, such a prior's exposure outweighs a page's own for about a hundred flushes, so the posterior's rate stays near the floor while the page drains, and the page looks static. At 8 MiB, the posterior cleaned 132 pages in the first ten flushes under skewed overwrites, where the single-rate and static-share branches cleaned 2 and 3, and 64 on the mixed workload, where they cleaned none and one; at 16 MiB under churn, 22 against none. On the mixed workload, the controller answered by lowering the price of space from $1.4 \cdot 10^{-3}$ to $2.6 \cdot 10^{-4}$, the

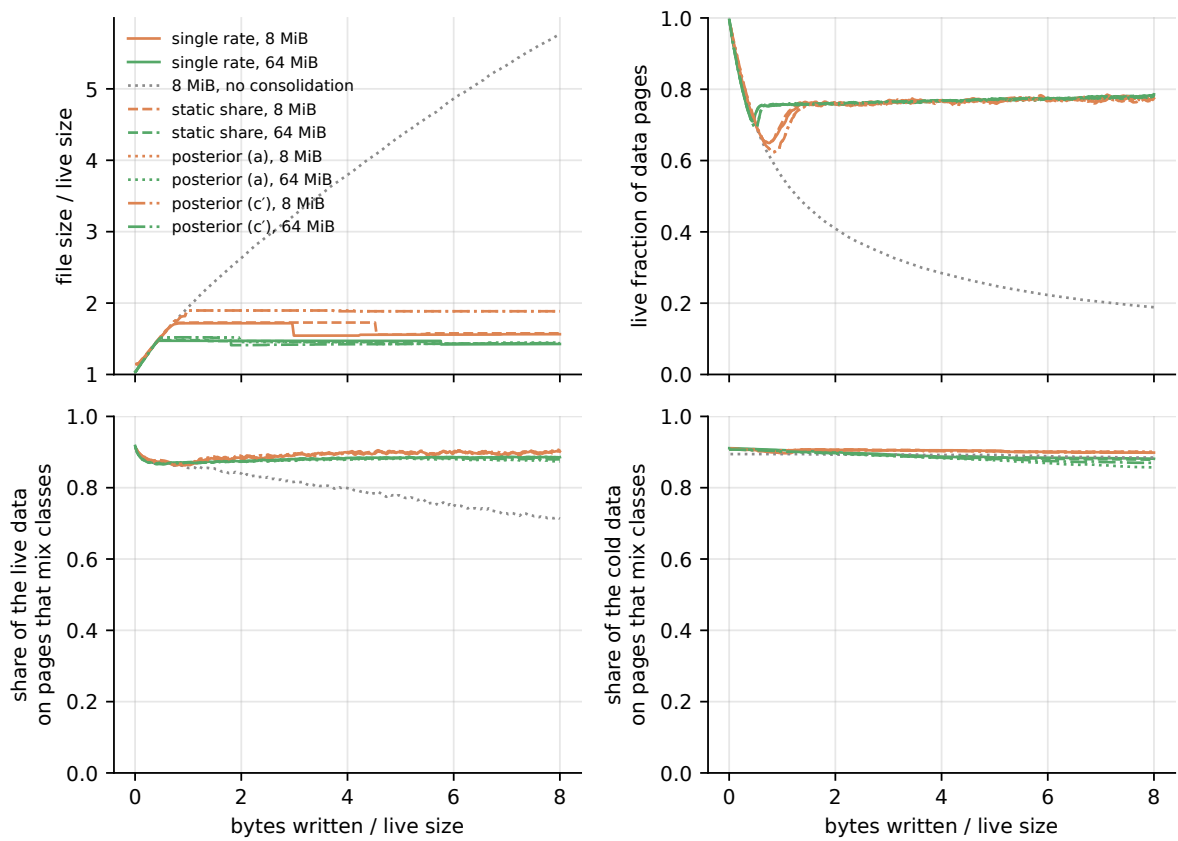


Figure 47.2: The mixed workload on the three branches, the posterior under both rules, with the state: file size over live size, the live fraction of the data pages, the share of the live data on pages that mix classes, and the share of the cold data on such pages.

file grew to 3791 pages where the single-rate branch's peaked at 3425, and the holes stayed: no truncation returned them in the rest of the run. Under appends, which have no fill phase but whose logs rotate hundreds of flushes after a page is written, the replay found the prior at the floors in its first flushes too, and afterwards, as rotations came in bursts, swinging between 0.01 and 1.1 events, with means from $2 \cdot 10^{-3}$ to 0.09.

47.5 What it costs

The consolidator state costs 2.5 to 12.6 % of the bytes written, 0.7 to 0.93 times the static-share branch's. An entry holds the posterior's discounted loss events, its exposure, and its count of touched chunks as f32s, and the class prior as a byte, 13 bytes against the static-share branch's 16, beside the page, its epoch, its coverage, and the estimate's epoch as varints, some 7 bytes more. The draft has the state record every page the fold drains, so what a flush records grows with the entry, and the costs' ratio is mostly near the entries', some 20 bytes against 23. Each cell gives the bytes written per byte the application wrote over the whole run, with the state and without it, and the state's share of the former.

whole run	single rate	static share	posterior, (a)	posterior, (c')
uniform, 8 MiB	3.317, 3.228, 2.7 %	3.422, 3.233, 5.5 %	3.380, 3.210, 5.0 %	3.380, 3.215, 4.9 %
uniform, 64 MiB	3.861, 3.605, 6.6 %	4.128, 3.600, 12.8 %	4.054, 3.620, 10.7 %	4.096, 3.610, 11.9 %
skewed, 8 MiB	2.354, 2.306, 2.0 %	2.418, 2.311, 4.4 %	2.419, 2.326, 3.9 %	2.413, 2.320, 3.8 %
skewed, 64 MiB	2.714, 2.532, 6.7 %	2.908, 2.530, 13.0 %	2.869, 2.550, 11.1 %	2.874, 2.542, 11.5 %
mixed, 8 MiB	2.387, 2.329, 2.5 %	2.454, 2.334, 4.9 %	2.455, 2.325, 5.3 %	2.437, 2.329, 4.4 %
mixed, 64 MiB	2.747, 2.544, 7.4 %	2.950, 2.542, 13.8 %	2.903, 2.543, 12.4 %	2.905, 2.538, 12.6 %
append, 16 MiB	2.736, 2.653, 3.0 %	2.810, 2.654, 5.6 %	2.904, 2.788, 4.0 %	2.912, 2.779, 4.6 %
churn, 16 MiB	2.108, 2.079, 1.4 %	2.141, 2.083, 2.7 %	2.134, 2.080, 2.5 %	2.128, 2.076, 2.5 %

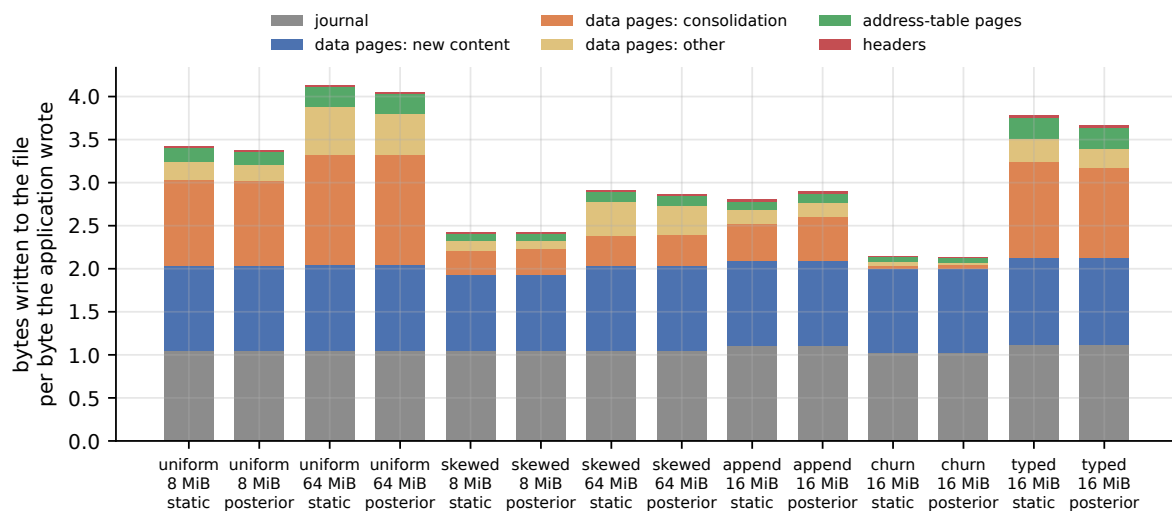


Figure 47.3: Where the bytes written to the file went, per byte the application wrote, over each whole run, on the static-share branch and the posterior under rule (a). “Other” is mostly the consolidator state.

Ranking a mixture takes work, and as the draft describes it, most of a flush's. A page's index is the root of a sum over the mixture's components, one for each number of its untouched chunks that may drain: some 20 on a data page, and on a leaf, whose chunks are its statements, up to some 300, of which some 250 carry weight. The first version of the branch found the root by bisection, 32 steps, each looking every component up in the table of Φ with two logarithms, and computed the weights with six $\ln$ Γ each. At 08b575d, regula falsi on the logarithms of what waiting saves and what cleaning gains finds the root in

about 10 evaluations, a shape's row of the table is found once per page, and the weights follow from one another by recurrence; the indices agree with bisection's to within 10^{-8} . Rerun with the state and rule (a) on the uniform, skewed, append, and churn workloads, 08b575d cleans as 9f99a1f does: identically, but for 0.08 % more writes under uniform overwrites at 64 MiB.

Each cell gives the instructions that callgrind counted in the flushes of the uniform workload at 1 MiB, in billions, and the share of them that ranking took, the table's construction aside, which takes 3.6 billion more, once per process, at the first ranking.

flushes, uniform, 1 MiB	single rate	static share	posterior, first	posterior, faster ranking
instructions, billions	2.16	6.46	8.55	3.37
ranking's share			74 %	33 %

The first version's flushes took four times the single-rate branch's instructions, and 1.3 times the static-share branch's, whose fit, two bisections of 50 steps for every page that loses content, takes 60 % of its run; with the faster ranking, the posterior's flushes take 1.6 times the single-rate branch's instructions and half the static-share branch's. At 1 MiB, the file has few leaves, the pages whose rankings cost the most, so the posterior's share of the work is likely larger in larger files.

In memory, a page's estimate takes 36 bytes, against the static-share branch's 24, and each statement 5 more. The estimate holds the posterior's three sums, its class prior, and the draining share the ranking last found as f32s, and its epoch; the page table adds the count and bytes of its untouched chunks. Each statement's slab record adds the size its statement states and whether it has lost bytes, where the draft asks for one bit and takes the size from the fragment at the first loss.

47.6 What to change in the draft

- **Keep one rate per page for cleaning.** On kladde-bench's workloads, the cure model cleans no better than the single-rate draft, writes more where content lives long, and costs nearly twice the single-rate draft's state and, with the faster ranking, 1.6 times its flushes' instructions in the one run measured. This page does not test the single-rate posterior, the draft's Gamma posterior without the static share, which would keep the single-rate draft's entry and cost a ranking of one component per page.
- **Weigh the empirical prior by the losses it has seen.** The floor on the spread, a thousandth of the mean squared, lets the prior claim 1000 loss events when it has seen none, as at the end of a file's fill phase, and those pages then look static for some hundred flushes; a prior of A_0 events should rest on at least that many, $A_0 \leq \sum k_i$, or on a fixed weak prior until the file's fresh pages have lost that much.
- **Watch pages for longer than their first flushes, or expect the prior to miss lifetimes.** Logs rotated every few hundred flushes lose nothing in a page's first ten; the prior then says nothing about how long content lives, only that it has not died yet.
- **Expect nothing from finding static content until survivors are moved by class.** Cleaning moves all of a page's survivors together, cold and cool alike, so no estimate of which pages are ripe changes that nine tenths of the cold data share pages with data that changes. The cure model knows of each chunk whether it has lost bytes, and could send a cleaned page's untouched chunks to other pages than its touched ones.
- **Say what finding the index costs, and how to find it.** The draft counts $O(n)$ work per loss for the weights, but ranking costs the weights plus a root-finding over them, some 250 components on a leaf; with bisection, ranking took three quarters of the flushes' instructions, and regula falsi on logarithms, which the index's slope makes nearly linear, cuts ranking's instructions more than fivefold.
- **Keep leaves out of the cure model.** A leaf's statements die whole, where the draft's simulation found the cure model gaining least over the static-share fit, and a leaf holds a few hundred of them, which the mixture pays for on every ranking.

47.7 Reproducing

The runs of this page ran from kladde-rs's branch `bayesian-ripeness` at `9f99a1f`:

```
git checkout 9f99a1f
cargo build --release -p kladde-bench
./target/release/kladde-bench /tmp/bench-gain
KLADDE_BENCH_OPTION=1 ./target/release/kladde-bench /tmp/bench-option
KLADDE_BENCH_NO_STATE=1 ./target/release/kladde-bench /tmp/bench-gain-no-state uniform skewed mixed ap
KLADDE_BENCH_OPTION=1 KLADDE_BENCH_NO_STATE=1 ./target/release/kladde-bench /tmp/bench-option-no-state
```

The rerun at `08b575d` ran `./target/release/kladde-bench /tmp/bench-08b575d uniform skewed ap-
pend churn`. The instruction counts come from the same build at `08b575d`, at `799c661` on `ripeness`, at `a856b06` on `ripeness2`, and at `9f99a1f`, each run under `callgrind`, whose file descriptor check needs a finite limit:

```
ulimit -n 4096
valgrind --tool=callgrind --callgrind-out-file=cg.out ./target/release/kladde-bench /tmp/bench-cg quick
callgrind_annotate --inclusive=yes cg.out | grep -E 'Inner>::flush_now|refresh_ranking|PhiTable>>::ini
```

In `kladde-docs`, keep the tables gzipped under `content/evaluation/data/bayesian-ripeness/`, in `gain/`, `gain-no-state/`, `option/`, `option-no-state/`, and `gain-08b575d/`, and draw the figures with those of the other ripeness pages:

```
D=content/evaluation/data
O=content/evaluation/figures/bayesian-ripeness
tools/plot-evaluation.py --only tradeoff,mixed --out $O "single rate=$D/ripeness/ripeness" \
    "static share=$D/ripeness2/ripeness2" "posterior (a)=$D/bayesian-ripeness/gain" \
    "posterior (c')=$D/bayesian-ripeness/option"
tools/plot-evaluation.py --only write-breakdown --out $O "static=$D/ripeness2/ripeness2" \
    "posterior=$D/bayesian-ripeness/gain"
```

The empirical priors quoted above come from a replay of the workloads that asks the store for them with `Store::describe_priors`, which is not part of the benchmark.

Chapter 48

Superseded

Designs that were worked out, sometimes built and measured, and then replaced.

They are kept for the reasoning rather than for the result. Each of them was a reasonable answer to the constraints as they stood, and understanding *which* constraint changed is usually more instructive than the replacement design on its own — and it is the cheapest defence against re-proposing one of them.

Nothing here describes current behaviour. Where a document says “kladde does X”, read “kladde did X”.

48.1 What was replaced, and by what

superseded design	replaced by	what changed
The relocatable heap — allocations as contiguous ranges in a flat address space, an <code>id</code> → address table, and sizedness as a placement hint	Pages and the address table	the move to copy-on-write pages: nothing is adjacent to anything any more, so contiguity, placement-by-address, and sizedness all stop meaning anything
Incremental compaction — a potential function over byte addresses, frontier slides, and an augmented B+ tree to find the best move in $O(\log n)$	Consolidation	same cause: there is no address ordering left to descend, so victim selection becomes the log-structured live-fraction rule instead
The in-place flush — conflict detection, write-ahead copies, read hoisting for restartability	The copy-on-write protocol	a flush no longer overwrites anything a recoverable state depends on, so there is nothing to protect against and nothing to restart from
Whole-allocation table entries — one statement per allocation, chunk lists, extents, spilling, and a separate index of table pages	Statements	keying by byte offset instead of by allocation made per-range statements viable, which removed the need for every mechanism in the left column

48.2 A note on measurement

The compaction work in particular was implemented, benchmarked across several orders of magnitude of heap size, and tuned. Those numbers describe a system that no longer exists, but the *methodology*

notes in it — especially about build-to-build timing drift swamping the effects being measured — apply to anything measured later, and are the reason that document keeps its numbers rather than being summarised away.

Chapter 49

The relocatable heap

Superseded. Replaced by [pages and the address table](#).

A partition of a flat address space over stable ids, which could close its own gaps one bounded step at a time. Implemented, measured, and tuned; the description below is of a system that no longer exists.

49.1 What it was

The snapshot was a **heap**: a partition of $[0, \text{end})$ into allocations — contiguous byte ranges, each tagged with the stable id naming it — and **gaps**, the free space between them. Gaps were not represented on disk at all: the set of gaps is derivable from the set of live allocations, which is derivable from the per-allocation id tags, so nothing needed to record them. Opening a file rebuilt the $\text{id} \rightarrow \text{address}$ table by **scanning** live allocations.

An allocation therefore had an **address** in addition to its id and size — its current offset in the file, which could change over the file’s lifetime.

49.2 The trait

```
pub trait RelocatableHeap {
    type Id: AllocationId;
    type Address: Word;
    type Size: Word + Into<Self::Address>;

    fn alloc(&mut self, id: Self::Id, size: Self::Size) -> Result<Self::Address, HeapError>;
    fn free(&mut self, id: Self::Id) -> Result<(), HeapError>;
    fn resize(&mut self, id: Self::Id, new_size: Self::Size)
        -> Result<Relocation<Self::Address, Self::Size>, HeapError>;

    fn lookup(&self, id: Self::Id) -> Option<(Self::Address, Self::Size)>;
    fn len(&self) -> Self::Address;
    fn live_bytes(&self) -> Self::Address;
    fn iter(&self) -> impl Iterator<Item = (Self::Id, Self::Address, Self::Size)> + '_;

    fn propose_compaction_step(&self, budget: Self::Address) -> Option<Step<Self::Address>>;
    fn commit_compaction_step(&mut self, step: Step<Self::Address>);
}
```

49.2.1 The heap owned the id table

The naive division of labour would put free-space management in an “allocator” and the $\text{id} \rightarrow \text{address}$ table in the layer above. That is wrong, and the reason was **compaction**: choosing a good move requires

simultaneous, index-grade access to *both* the geometry — where are the gaps, what is adjacent to what — and the id mapping. Split them and the compactor has to interrogate its caller in a loop, or keep a shadow copy that can drift.

The component that decides where things go should own that table rather than shadow it.

A bad id was therefore the *heap's* error, which it would not be for a free-space-only allocator.

This argument survives the supersession, in changed form: the [fragment map](#) is still the single structure that both resolution and consolidation read, for the same reason.

49.2.2 Ids came from outside

The heap never minted ids; the caller did. Id minting is coupled to *persistence* — an id has to be a stable identity from the moment anything might serialize it — whereas placement is coupled to geometry.

49.2.3 A step was a byte range, not an allocation

```
pub struct Step<A> { pub from: A, pub to: A, pub len: A }
```

Deliberately id-free, so that one step could describe a whole **run** of neighbouring allocations sliding together — the natural unit for I/O, since the fixed per-operation cost is paid once per run rather than once per allocation. `to < from` always, so a source/destination overlap is a downward move and can be done with a forward copy.

49.2.4 budget was a ranking input, not a cap

`propose_compaction_step` could return a step costing **more** than the budget when no worthwhile cheaper step existed. Otherwise a heap whose only useful move is one large slide would claim quiescence and never compact at all.

This idea survives in the [consolidation budget](#), for the same reason.

49.3 Sizedness

Every id recorded whether the allocation it named was **fixed-size** or **resizable**, carried in the low bit of the raw pointer value:

```
raw = (counter << 1) | fixed_bit
```

The counter pool was **shared** between the two sizednesses, so the counter alone was unique and ids stayed dense at roughly twice the counter — which mattered for any table indexing by id. A high bit would have been the alternative and is much worse: it doubles the apparent id range and is fatal to density.

It was **placement-relevant**, which is why it had to be visible to the heap: fixed-size allocations are minted in bulk at a handful of distinct sizes, so they form dense size classes worth indexing, while resizable ones scatter across many sizes and would have to move again on the next growth.

Two consequences followed:

- a sizedness conversion could not re-tag in place; it had to mint a **new** id, copy the content, and free the old one — which is why the journal carried a `Convert(old_id, new_id, size)` record;
- the Rust API needed two owned handle types, `UniquePointerResizable` and `UniquePointerFixedSize`, differing only in which operations they permitted, as a typestate.

Why it is gone. The distinction existed so that neighbours of a fixed-size allocation could rely on it not moving. Under the page-oriented design nothing is adjacent to anything — every reshape is a statement edit — so it buys nothing. The `Convert` record, the second handle type, and the low pointer bit all went with it.

49.4 Errors and composition

```
pub enum HeapError {
    OutOfMemory,    // no free range large enough, and the space cannot extend
    UnknownId,      // freed, or never allocated
    DuplicateId,    // alloc with an id that is already live
}
```

OutOfMemory arose only when the address width was exhausted, not when the heap was merely full, because extending the address space at end was always available and always correct.

A concrete backend was a `Composed(Storage, RelocatableHeap, id pool)`: the heap owned geometry and the table; the backend owned id minting, storage I/O, and the journal. There was no second in-memory table anywhere.

A marker trait `IncrementallyCompactableHeap` gated the *user-facing* compaction controls — the methods themselves were defaulted on the base trait so a shared flush path could call them without a bound, with a non-compacting heap simply proposing nothing. Compare `ExactSizeIterator`, which marks `size_hint` as meaning something.

49.5 Placement

Placement was compaction with the copy pre-paid.

An evacuation moves an existing allocation into the lowest gap that fits it, paying size bytes of copying. A placement puts a *new* allocation into the lowest gap that fits it, and the bytes were going to be written regardless. So placement was free compaction, using the same index and the same query: `lowest_gap_fitting(min_len) -> Option<(Address, Size)>`, answered in $O(\log n)$.

Best-fit by **address**, not by size: filling from the bottom keeps the compact prefix growing, whereas filling the tightest gap wherever it happens to be scatters live bytes above the frontier and leaves work for later.

Deferred placement. The journaled backend deferred placement to flush time rather than assigning an address when the allocation was minted, because a flush knows the *whole* batch of pending allocations and can sort them **first-fit-decreasing** — serving the large ones while the large gaps are still intact. Sorting also made the layout independent of hash-map iteration order, which would otherwise make the layout, and therefore how much work compaction later had to do, unreproducible from run to run.

Both of these survive the supersession, in the [flush's placement phase](#): first-fit-decreasing over the whole batch, and the same caveat that it becomes best-effort over the ready set once dependencies block.

49.6 resize could ask for two moves

```
pub enum Relocation<A, S> {
    None,
    Single { old: A, new: A },
    Double { first: (A, A), then: (A, A), then_len: S },
}
```

Double existed for the *lift*, described in [Incremental compaction](#): a shrinking allocation is moved up and a replacement is moved down into the space it vacated. The replacement's destination *is* the mover's own old address, so the two copies had to happen in the order given, and `then_len` was carried because the second move is of a *different* allocation, one the caller never asked about and whose size it therefore could not look up.

Chapter 50

Incremental compaction

Superseded. Replaced by [consolidation](#).

How the [relocatable heap](#) closed its own gaps, one bounded step at a time, choosing each step in logarithmic time. This was the most developed part of the old implementation: built, benchmarked across several orders of magnitude, and tuned.

It is gone because there is no address ordering left to descend: content lives in pages, nothing is adjacent to anything, and “moving a byte downward” is not a meaningful operation. What replaced it is the log-structured live-fraction rule, which is a much simpler policy over a much simpler geometry.

Three things in it are worth keeping, and are called out below: the **no-stop-the-world requirement**, the **adversarial-workload stance**, and the **measurement methodology**.

50.1 The constraints

- **No stop-the-world.** Every step bounded and independently valid.
- **Finding the next step must be $O(\log n)$,** not just executing it. A scan over allocations to pick the best move would make each step linear, and a full compaction quadratic.
- **Steps must compose** into an actual compaction — repeated application must converge, without cycling or stalling.

The no-stop-the-world requirement was a hard one rather than a nicety: kladde is meant to sit under interactive applications, and a multi-second pause while a large heap is rearranged is not acceptable. It also forced a genuinely useful property — because every step is independently valid, the heap is always consistent, so a crash mid-compaction costs at most one step.

This requirement survives, and the page design satisfies it more easily: a flush’s consolidation work is bounded by a budget, and a crashed flush costs nothing at all.

50.2 The potential

The key move was to find a single number that every useful step decreases.

$$\Phi = \sum_{\text{live bytes } b} \text{address}(b)$$

The sum of every live byte’s address. Equivalently, per allocation, $\sum_a \text{size}(a) \cdot (\text{address}(a) + (\text{size}(a) - 1)/2)$.

With L total live bytes, $\Phi \geq L(L - 1)/2$, with equality for **every** gapless layout regardless of the order allocations happen to sit in. The excess counts exactly the (free byte, live byte) pairs where the free byte sits *below* the live byte — the heap’s inversions. So **minimal $\Phi \Leftrightarrow$ compact**, and the excess doubled as a fragmentation-debt metric.

A tempting simpler variant — summing allocation *start* addresses rather than every byte’s — is subtly wrong: its minimum depends on allocation order, insisting that small allocations come first, and the bias this induces in the greedy policy is actively harmful.

Relocating an allocation of size s downward by distance d copies s bytes and makes $\Delta\Phi = s \cdot d$ progress, so **the gain per byte copied is d** — the travel distance. That single lens priced every candidate move, which is what made them comparable.

Φ was the *progress* measure and end the *payoff*, and the payoff lags: end only shrinks when a step clears the topmost live byte. So the real objective added a term:

$$\Phi' = \Phi + K \cdot \text{end}$$

where K is a tuning knob in **addresses per byte of file size**. $K = 0$ recovers pure Φ -descent.

50.3 The candidate shapes

Four kinds of move, best-scoring one taken.

Frontier slide — take the run of allocations immediately above the lowest gap and slide it down into that gap. The workhorse, and the choice of *lowest* gap rather than *widest* gap is what made compaction linear rather than quadratic: sliding bottom-up telescopes, because every slide extends the already-final prefix, so each byte is copied once.

The measured difference was dramatic — on the endgame benchmark, switching from widest-gap to frontier slides took total copying from 157 MB to 573 KB, and steps from 81,280 to 2,135. Bytes copied per live byte went from scaling with heap size to a flat 1.03 at every scale.

End slide — a slide of the topmost run, chosen because it retires end rather than because it travels far. Offered only when the untruncated run fits the budget, since a truncated end slide moves bytes without retiring the length that justified it.

End evacuation — move the topmost allocation into the lowest gap that fits it.

Classic evacuation — move any allocation into the lowest gap that fits, chosen for travel distance. A classic evacuation never needed an end credit: whenever it would earn one it coincides with the end evacuation, which is scored anyway — verified byte-identical across 66,907 recorded steps.

Why it terminated. Every step strictly decreases Φ' , a non-negative integer bounded below, so the process cannot cycle or run forever. It also could not stall: if any gap exists below any live byte, a frontier slide with positive gain exists, so quiescence is exactly compactness.

50.4 The evacuation index

The data structure that answered “what is the best move?” in $O(\log n)$ — an augmented B+ tree holding allocations **and** gaps in one ordered structure, keyed by a triple:

```
( (size << 2) | (is_gap << 1) | is_fixed,  score,  address )
```

The crucial property is that **the key order is the validity relation**: for a gap G and an allocation A ,

$$\text{key}(G) > \text{key}(A) \iff G.\text{width} \geq A.\text{size}$$

So “does this gap fit this allocation?” is a key comparison, “the set of gaps that fit A ” is a suffix of the tree, and a maximum-over-pairs query becomes a range query. The bit layout made the ordering within one size *resizable allocation*, *fixed allocation*, *gap*, so a gap sorts above every allocation it can hold, including one of exactly its own size.

Each node carried a subtree summary with `min_gap_pos`, `max_gap_pos`, and per-class best-mover and best-pairing fields. `min_gap_pos` at the root *was* the compaction frontier.

50.4.1 The lesson worth keeping

Adding a class cost **seven** scalar slots rather than the three one might expect — the node grew from 72 to 112 bytes — and the reason generalises.

The class weight **cannot enter the merge**: the tree’s aggregation function is a pure fold over subtree summaries and has no access to a runtime parameter. So the merge had to keep each class’s best *unweighted*, and the weighting was applied afterwards, at the root, by a separate scoring step. Roughly half the aggregate’s growth existed purely to defer the weighting.

A **multiplicative** weight that takes finitely many values can be expressed by *partitioning* the aggregate — one slot per value — rather than by linearising it into the merge.

That works precisely because the class count is small and fixed, and it is the kind of thing worth remembering the next time a tunable has to live inside an augmented tree.

A related saturation subtlety: saturate *before* scaling, or a weight turns a wrapped difference into a large positive.

The implementation used the `sweep-bptree` crate, which appears unmaintained. The augmentation API it provided — `Argument`, `from_leaf`, `from_inner`, `descend_visit` — is small, and the expectation was to vendor or replace it eventually; nothing in the design depended on that crate beyond the shape of its augmentation hook.

Maintenance cost was the dominant CPU cost of a step: every move re-keys the moved allocations and adjusts the neighbouring gaps, roughly sixteen index operations per allocation moved, of which two are irreducible. That is why steps were expressed as **runs** rather than single allocations.

50.5 Pathologies

50.5.1 The stance

Kladde is meant to be used by applications outside the authors’ control, so the workload is treated as **adversarial** in the oblivious sense: not that an attacker is choosing it, but that no assumption may be made about it beyond what the API allows.

This rules out a category of reasoning that would otherwise be tempting. “That does not happen in our benchmarks” is not an argument, because our benchmarks are not the workload. The design accepts a small average-case cost in exchange for a bounded worst case.

This stance survives and applies to every policy decision in the current design.

50.5.2 The stress test

A resizable allocation sits at a **low address** and is repeatedly **shrunk** by a small amount.

Each shrink opens a small gap just above it, at a very low address, so the frontier is always at that gap, and a frontier slide dutifully slides *everything above it* down. The next shrink opens the gap again, and the whole heap slides again.

Measured: 255,360 bytes copied for 16 shrinks over 15,960 live bytes — and, crucially, **independent of the shrink size**, because the cost is driven by how much lives above the gap, not by how big the gap is.

The greedy policy was not wrong here; each individual slide really was the steepest descent available. The problem is that the *sequence* of locally optimal moves is globally terrible, because the churning allocation keeps re-creating the same gap.

50.5.3 Remedy 1: weighting by class

Give fixed-size and resizable allocations different weights γ in the potential, on the grounds that a fixed-size allocation, once placed well, stays placed. Expressible in the index precisely because γ takes finitely many values, which is what cost the four extra slots per class.

What it does not fix: in the stress test the heap has only *one* resizable allocation, so γ scales every competing mover uniformly and has no effect at all. It addressed a different failure mode and was implemented first because it is the cheaper half.

50.5.4 Remedy 2: the lift

The mechanism that actually addressed the stress test.

Diagnosis: the problem is not any single move; it is that a *specific allocation* keeps re-opening a gap at a low address. So detect that allocation and move it out of the way — *up*, against the direction compaction normally pushes.

Mechanism: a sparse `HashMap<Id, u8>` counts how many times each allocation has shrunk; resetting the counter is removal, so the map holds only currently-churning allocations and stays small. When a count crosses a threshold the allocation is **lifted** to a gap that fits it, or to the end, and a **replacement** allocation is moved down into the space it vacated.

The trigger compares the work the churn is causing against the cost of getting rid of it:

$$\text{count} \times L \geq T \cdot (s + r_len) \quad \text{and} \quad \text{count} > 1$$

where L is the distance the lift would travel, s the churning allocation's size, r_len the replacement's, and T a tuning knob. The left side is the copying the churn has already caused; the right side is what fixing it costs.

Choosing the replacement is largest-fit rather than best-gain, because the goal is to fill the space rather than to make progress — a partly filled hole would just become the next frontier. The search used a predicate that is *sound but incomplete*: `max_alloc_addr[class] > floor` proves a candidate might exist and never claims one exists when none does. An incomplete predicate was acceptable because failing to find a replacement merely means the lift falls back to a plain slide.

Measured: the lift reduced total copying about fourfold — from 1,995 bytes per byte reclaimed to 505 — and left the churning allocation topmost in every enabled configuration.

Both mechanisms defaulted to off ($\gamma = 1$, $T = 0$), and that path was bit-identical to the unweighted policy.

50.6 Measurement methodology

Worth keeping, because it applies to anything measured later.

Three workloads — growing, shrinking, and quiescing heaps — recorded per-step across several orders of magnitude of heap size. Headline results: frontier slides make total copying linear in live bytes at all scales; the endgame went from 7.51 s to 63 ms; adding the end term with a plausible K improved file-length payoff without measurably hurting Φ descent.

Build-to-build timing drift on the development host exceeded 30 %, while within-burst variance was ± 3 %. Several early conclusions about the cost of small code changes turned out to be measuring the drift. Compare alternating builds, not sequential ones.

50.7 What was still open when it was retired

- **A generation or age counter**, weighting allocations by how long they had sat still, to generalise the shrink counter from “has shrunk repeatedly” to “is churning at all”. It needed compaction to be able to *swap* runs rather than only slide them.
- **The reshape conservatism**: several places treated “this allocation was resized” as “it may have relocated”, because placement is not known until it runs. Safe but pessimistic — and the same conservatism reappears in the current design's [hoist condition](#).
- **Other pathologies not yet characterized**. The stress test was found by reasoning about the policy, not by fuzzing. A systematic adversarial search over operation sequences, maximizing bytes copied per byte reclaimed, would be a good way to find the next one, and did not exist.

That last item is the one most worth carrying forward: the current consolidation policy has had no equivalent adversarial search either.

Chapter 51

The in-place flush

Superseded. Replaced by [the copy-on-write protocol](#).

The flush design that wrote updated allocations over their old locations, and the machinery that was needed to make that safe.

51.1 What it was, and what it needed

A flush applied the folded journal by writing each allocation's new content to its address — in place where the size allowed, relocating where it did not. Because executing a flush **destroyed bytes that re-executing it would need**, three mechanisms were required, none of which exists any more:

Conflict detection. A flush that relocates allocation A into space that allocation B still occupies, while some other action still needs to read B, has a read/write conflict. Finding those conflicts meant lowering the schedule from ids to addresses and checking overlaps.

Write-ahead copies. Where a conflict could not be scheduled away, the overwritten source had to be copied somewhere safe first, so that a re-run could still find it.

Restartability analysis. A flush interrupted partway through had left the file in a state that was neither the old one nor the new one, so recovery had to determine how far it had got and what remained safe to redo.

On top of that, page-granularity **collateral damage** to untouched neighbours — “passenger corruption”, where a torn write damages bytes the flush never meant to touch — could at best be *detected*, never prevented, because those bytes were inside pages the flush was legitimately writing.

51.2 Why it went

Under [the reuse rule](#) a flush writes only to pages that no recoverable state references. That single change removes all four problems at once:

- there are no read/write conflicts, because nothing a flush writes is something anything still reads;
- there are no write-ahead copies, because nothing is destroyed;
- there is no restartability analysis, because a crashed flush is simply re-run against an untouched previous state;
- there are no untouched neighbours in the write path at all, so passenger corruption stops being a category rather than being detected.

The fold survived unchanged — it is [still there](#) — and everything downstream of it in the old flush design is simply gone.

This was the single largest argument for the redesign, and it is worth stating as a general lesson rather than a local one: **the machinery removed was bespoke, and the machinery that replaced it is well-trodden**. Never-overwrite plus cleaning is Rosenblum's log-structured file system; the alternating header

commit is LMDB's meta pages; copy-on-write with checksums is ZFS, WAFL and btrfs. Their failure modes are understood and bounded; the conflict/hoisting apparatus's were not.

51.3 What kladde does differently from those systems

Worth recording, because it is what made the adoption cheap.

Kladde **reads in bulk and looks nothing up on disk**, and every difference that follows cuts toward simplicity: no on-disk search tree, a flat rewritten root instead of path-copied interior nodes, whole-world validation available for free at open, and a single `fsync` per flush where LMDB needs two — because kladde can afford to let the header lag one write behind and lean on journal replay, which a random-access database cannot.

51.4 What was given up

Stated in full at the time of adoption, and still accurate:

1. **Cleaning debt.** Every dirty piece is written once at flush and, on average, a fraction of a time again when consolidation later relocates its page-mates or itself. The worst case is uniform random small writes across a large file — the workload that hurts every log-structured system.
2. **Space overhead is a policy outcome, not a constant.** An early conjecture that it would be a constant was wrong: garbage accrues in proportion to write traffic and drains in proportion to consolidation effort.
3. **Loss of contiguity.** An allocation's bytes must be gathered at open and scattered at flush, and its description grows as it fragments.
4. **A minimum flush of several pages** even for a tiny transaction, where the in-place design might touch fewer. The crossover favours copy-on-write as soon as writes scatter at all, but a workload of frequent single-byte flushes pays more.
5. **Full trust in `fsync`,** leaned on harder: a single barrier, with the header issued unsynced.
6. **Both header pages corrupting simultaneously** is unrecoverable without a scan. They are single-sector writes at opposite ends of a two-page span, never written in the same flush, so this requires two independent failures; replicating headers at two more fixed locations would cost two extra page writes per flush and is probably not worth it.

The one workload that regresses — high-frequency tiny flushes — is measurable, and batching flushes attacks it directly.

Chapter 52

Whole-allocation table entries

Superseded. Replaced by [per-range statements](#).

The first address-table design under copy-on-write pages: one statement per allocation, carrying its size and the locations of all its chunks. It was worked out in full — including the escalation mechanisms below — before per-range statements made all of it unnecessary.

52.1 What it was

Two statement kinds:

- `entry(id) = (size, chunk list)` — the allocation’s size plus the lengths and locations of all its chunks, extent-compressed;
- `tombstone(id)` — the id is unallocated.

A statement in a page with a higher epoch shadowed any statement about the same id in a page with a lower epoch.

Chunks were the unit of content: an allocation was a sequence of chunks, each of arbitrary size up to `MAX_PAGE_CONTENT`, each contained entirely in one page, with logical offsets implied by the lengths before them rather than stored. A chunk could reference a reserved **null page** to state that its range was uninitialized.

That chunk model is the direct ancestor of today’s Ref and Zero statements, and two of its rules [survive unchanged](#): the writer prefers maximal pieces, and whoever rewrites bytes may re-cut them. The free-re-cutting argument — that consolidation should pack pages the way stock is cut rather than the way bins are packed — was developed here.

52.2 Why per-chunk statements were rejected first

An earlier version tried a per-allocation shape statement plus independent per-chunk statements, so that updating one chunk of a large allocation would restate only that chunk’s location.

That founders on **key stability** once chunking is flexible: with consolidation free to split and merge chunks, and splices free to insert them, a chunk’s *index* within its allocation is not a stable name, and a statement keyed by `(id, chunk index)` can be silently re-aimed by an unrelated edit earlier in the same allocation.

Whole-entry statements had no sub-keys to destabilize, and bought three simplifications outright: a resize is just a new entry, re-allocating a freed id is just a new entry, and the reader never assembles an allocation from statements of mixed epochs.

The resolution, which came later: key statements by **byte offset** rather than by chunk index. A byte offset *is* stable under overwrites and resizes — the operations that dominate — which is what makes per-range statements viable where per-chunk statements were not. The price is that a middle splice shifts the offsets of everything behind it, which is [recorded as a known cost](#) rather than a blocker.

52.3 The mechanisms it needed, and no longer does

52.3.1 Extents

Chunk locations were encoded as extents wherever chunks happened to be contiguous: a run of full-size chunks in consecutive pages was one (start_page, page_count) pair, and any other chunk a (length, page, offset) triple.

The reasoning behind this is worth keeping. File systems have given both answers historically: FFS and ext2 used fixed-depth radix trees of block pointers, while every modern system — ext4, XFS, NTFS, btrfs — moved to **extents**, precisely because real files are mostly contiguous, escalating to a per-file extent tree only when fragmentation forces it. Kladde took the extent position, with a cheaper escalation than a tree, because it never searches the structure on disk.

Per-range statements make this moot: each statement carries its own address, and a run of contiguous content is simply one statement with a large size.

52.3.2 Spilling

If an entry's encoding outgrew a threshold — say a quarter page — the entry **spilled**: its statement in the shared table page shrank to (size, continuation page numbers), and the chunk list moved to dedicated pages owned by that entry alone and rewritten together with it.

The obvious question is why this was needed at all — why an oversized chunk list could not simply split across ordinary table pages with no extra machinery. Under per-chunk statements it could; under whole-entry statements it could not, because **the statement is the unit of shadowing**, so it must be replaceable as one atom, and fragments of one entry spread over shared pages would need stable sub-keys to be shadowed individually — precisely the key-stability problem that ruled per-chunk statements out.

Spilling kept the statement *logically* whole while letting its bytes exceed a page. Its cost was stated rather than hidden: a huge, badly fragmented allocation rewrites its spill pages on every touch.

Per-range statements remove the need entirely: a large allocation's statements split across pages naturally, because no single statement ever needs to span one.

52.3.3 A separate index

The **index** was a small rooted structure that made scanning unnecessary: a list of (page_number, epoch) for every live AddressTable page, held in pages of its own kind, whose page numbers were in the header, and rewritten in full every flush.

Sizing, to justify “small”: at ~8 bytes per entry, a million allocations occupy $\approx 8\text{ MB} \approx 2000$ table pages; at ≈ 6 bytes per index entry that is $\approx 12\text{ KB} \approx 3$ index pages per flush. If the table grew to where the full index rewrite hurt, the index would get a second level and only the changed leaves would be rewritten — a two-level copy-on-write tree. That was the honest boundary of the “no trees” stance: trees are unnecessary for *lookup* here, but a bounded incremental commit of a large directory is legitimately what they are for.

Because the index was rewritten atomically with the commit, membership in it was authoritative: an address-table page not listed was dead, whatever its bytes said. Resolution never depended on the *absence* of an unreferenced page, only on the listed set and their epochs — which is what kept the scheme free of the reclamation-ordering subtleties a scan-based design has to solve.

What replaced it is the [child-reference tree](#) inside the address-table pages themselves, rooted at the header page. The Index page kind is gone; the escalation story survives as [the shape of the tree](#), with the same “escalate then and not before” reasoning and the same observation that it is a directory rather than a search structure. Since epochs live in the pages' own framing, the (page, epoch) pairing the index carried is no longer needed either.

52.3.4 Tombstone lifetime

A tombstone(id) had to exist as long as some **indexed** address-table page carried an entry(id) at a lower epoch *and* no indexed page carried one at a higher epoch — in that second case the id had been

re-allocated and the newer entry already shadowed both.

Because the index committed atomically, the lifetime rule was purely logical, with none of the durable-invalidation lag a scan-based design needs: a tombstone became droppable in exactly the commit that removed or consolidated away the last lower-epoch page carrying an entry for the id.

The modern rule is [the same idea expressed in pins](#), and the underlying question — “is any older mention still physically present?” — is the same one, now answered by the `mentions` counter.

52.4 What else survived

Beyond the chunking rules, the [coverage accounting](#) was designed here and carried over essentially unchanged: coverage is in-memory only, maintained $O(1)$ per shadowed statement, rebuilt for free at open during the bulk read, and never persisted.

So was the split of consolidation into data-page victim selection by live fraction and address-table consolidation by **id range** rather than page identity — including the observation that rewriting a window of statements sorted and adjacent re-establishes delta-encoding density, so that density is restored by cleaning rather than maintained by an invariant.

Chapter 53

Drafts

Half-baked ideas in progress.

Documents here are **not** held to the separation the rest of the hub observes: they may mix normative rules, language-agnostic algorithms, and Rust-specific details freely, because working out which is which is usually the last step rather than the first.

Nothing here is adopted. When a draft settles, it is split along the usual seams and moves into [Specification](#), [Implementation](#) and [kladde-rs](#) — or, if it is rejected, into [Superseded](#) with the reasoning intact.

53.1 Current drafts

draft	status
Open issues	an audit: where the other documents contradict each other, and what blocks a first prototype
The Move operation	the record is specified; how a flush states it is sketched, with the open questions listed but not answered
Segments	a direction, not yet a design
Cleaning by ripeness	a proposal: clean each page once waiting no longer pays, judged by how fast it still drains
Bayesian ripeness	a proposal on top of it: keep a posterior over each page's drain, for either variant of the draft, and decide from the posterior
Sorting survivors by temperature	another idea for reducing file size, orthogonal to how ripeness is estimated: try to avoid creating pages that mix cold chunks with chunks that are still draining.
Variable-size values	a proposal: separate how many bytes a value's encoding takes from how much room its place reserves; packed places unpad enums, write integers and pointers as varints, and admit small strings and vectors that keep short content inline

[Open issues](#) is the odd one out: it is not an idea in progress but a standing list, and it is kept here because it is maintained by audit and goes stale the moment the documents it indexes are edited.

Chapter 54

Open issues

Two lists, both maintained by audit rather than by memory: where the documents outside [Superseded](#) disagree with each other, and what has to be settled before a first Rust prototype can be built.

The prototype in question is one that can **create a file, open it, check the schema, hand out a `Kladde<T>`, accept mutations and record them in the journal, and flush**. [Segments](#) are deferred, and so is anything that only affects how well it performs.

54.1 Contradictions

Ordered most to least severe, where severity means *how likely this is to make someone write the wrong code*.

54.1.1 1. The journal is two different things — resolved

Was: [Transactions and batches](#) treated the journal as a **fixed-capacity allocation**, grown by being freed and re-allocated while empty, while [Durability](#) and [File format](#) described a **chain of journal pages** drawn from the reusable-page pool.

Resolved in favour of the chain. [Journal](#) now specifies the page layout, and [Transactions and batches](#) treats the segment's page budget as a flush trigger rather than a capacity: a transaction is appended whole, taking as many pages as it needs, and the flush that follows folds it.

54.1.2 2. Journal pages cannot satisfy the page framing — resolved

Was: page framing required every non-header page, `Journal` kind included, to carry a CRC over its content, while journal appends were exempted from whole-page writes — and a page appended to incrementally cannot keep a valid CRC unless every append rewrites it.

Resolved by scoping the framing to “every page except the pages that make up the current journal” and dropping `Journal` from the kind values. Journal pages now carry no framing at all, and their integrity rests solely on the chained, epoch-salted transaction CRCs — which is what [How much of the framing is load-bearing](#) already implied and now says.

Kept here, struck through, for one more audit; the shape of a journal page is now [specified](#) as well.

54.1.3 3. The durability promise is stated unqualified where users read it

[Durability](#) is careful: under a power cut, transactions issued after the last completed flush survive only up to a valid journal prefix, “minus whatever the operating system never wrote out”, because journal appends are not fsynced.

The two places an application author actually reads make the unqualified promise:

- the [hub page](#) — “an append-only on-disk journal that is durable by the time the mutating call returns”;

- the [tutorial](#) — “Nothing that has been acknowledged. If push returned, the element is in the file.”

Both are true for an application crash and false for a power cut. This is severe not because it is subtle but because it is a promise, and because the tutorial is where someone decides whether kladde is safe enough for their use.

54.1.4 4. Framing is per-record in one place and per-transaction in another

[Journal](#) and [Conformance](#) frame **transactions**, with a chained epoch-salted CRC. The [tutorial](#) frames **records**: “Every record carries a length prefix and a checksum.”

A format-level disagreement, and the two imply different recovery code. The tutorial’s version also pre-dates the chaining, so it describes recovery as truncating a torn record rather than as keeping the longest valid prefix.

54.1.5 5. mentions is claimed to have one reader and has at least three

[Liveness](#) states flatly: “This is the only reader of mentions”, meaning the release of a tombstone anchor’s pin at mentions == 1. [In-memory state](#) repeats it as “one thing only”.

But [Id recycling](#) reads it twice — to prefer ids whose tombstone is already dead, and as the tiebreak that argues for taking high-mentions ids first — and [Liveness](#)’s own ranking section reads it again, to score “dead statements whose removal would let a tombstone retire”.

The claim matters because it is the argument for how cheap mentions is. With three readers the field is harder to remove than the text suggests, and anyone auditing whether it can be dropped will reach the wrong conclusion.

54.1.6 6. Reclamation is optional and also required

[Allocations](#) says whether an implementation reclaims garbage is unconstrained, and that one which never consolidates is conforming.

[Guarantees](#) says “reclamation is incremental by construction” as though it were a property the spec asserts.

What remains is the wording of the guarantees row: the resolution is presumably that *incrementality* is required of whatever reclamation an implementation does, while *doing any* is not.

54.1.7 7. Conversion is gone, and the flush still reasons about it

Sizedness conversion was removed along with sizedness, taking the Convert record with it ([the superseded heap](#)).

[The flush](#) still uses it as the motivating example in three load-bearing places: the three-cycle that forces frees-first to be a priority rather than an edge, the [content-blind fast path](#)’s flag condition, and the [implementation order](#)’s step 5.

The arguments themselves survive — `Alloc(B); Copy(A-B); Free(A)` has the same shape as the conversion it describes — so this is stale vocabulary attached to live reasoning rather than a collapsed argument. It is listed here because a reader who checks whether conversion exists will conclude the section is obsolete, which it is not.

54.1.8 8. “Five layers”, six rows

[The layers](#) says a kladde file is described by five layers and then tabulates six. Trivial, and only listed because the table is the first thing a new implementer reads.

54.1.9 9. Which “read” the complexity bound governs — narrowed, not closed

[Guarantees](#) requires “read any byte of any allocation” in $O(\log F)$, and now also that “a non-mutating operation once the file is loaded” have no overhead over a non-persisted type. The second row is the application read, so the ambiguity the first row used to carry is mostly gone.

What remains: the first row still does not say it means the *storage-level* read, and it counts only the fragment-map lookup. If data pages are [not resident after loading](#), reaching the bytes is a pread rather than a dereference, so the honest bound is $O(\log F)$ lookups *plus* whatever fetching the bytes costs. Worth saying, since the whole point of the row is to tell an implementer what is achievable.

54.1.10 10. Grouping is “free” but constrained

[What is fixed and what is free](#) lists batching — “how an implementation groups application operations into transactions” — as explicitly not fixed. [Transactions](#) then requires that a record occurring outside any transaction be recorded as a transaction of its own.

That is a real constraint on grouping: an implementation may not leave a record ungrouped. The intent is clearly “how you *batch* is free, but everything lands in some transaction”; the wording does not say so.

54.2 Blockers for a first prototype

Ordered most to least difficult. Difficulty here means *design work plus implementation risk*, not lines of code — several large items are only large.

Items marked **deferrable** are not blockers; they are listed so that the line between them and the blockers is explicit.

54.2.1 1. The journal’s on-file representation — resolved

Was: unspecified beyond “the header names a journal segment’s start page”.

Resolved by [Journal](#): a segment is a singly linked chain of pages, each reserving an 8-byte trailer for the next page’s number and a checksum; transactions and records may span pages; and every checksum is a value of one CRC-32C chain salted with the segment’s epoch.

54.2.2 2. The byte encoding of journal records — resolved

Was: the record set had semantics but no bytes.

Resolved by [Records](#) and [Framing](#).

54.2.3 3. Bootstrapping — partly described

Still open in [Header pages](#): what the first header’s address-table payload contains, how the root and schema-table allocations come into existence before an address table describes them, and whether creation is a degenerate flush or a distinct path.

54.2.4 4. File extension is not covered by the durability argument

[The reuse rule](#) permits “extend the file”, and [I1](#) says a header is issued only after an fsync covering everything it references.

But extending a file changes **metadata**, and on common filesystems the new length is not durable merely because the data blocks were fsynced — it can require fsyncing the directory, or an explicit allocation step. A power cut can therefore leave a valid header referencing a page beyond the file’s recovered length, which breaks [I1](#) in exactly the way the design says cannot happen.

Small to fix, easy to miss, and it invalidates the central invariant if missed — which is why it ranks above items with far more code.

54.2.5 5. The header's byte layout

[Header pages](#) lists the fields and says “exact widths, ordering, and the reservation of space for future roots are TBD”. Blocking for both creation and open, and it is the one structure whose layout can never change, so the reservation decision has to be made now rather than discovered later.

54.2.6 6. The pointer encoding inside allocations

[Pointer encoding](#) is TBD: width, null representation, alignment.

This blocks the very first derived struct with a container field, because storing a `PersistableVec` means writing a pointer into the parent's bytes. The reference implementation's choice — 32-bit, zero reserved for null — is probably just right, but it needs to be *decided*, since it is a format-level fact that readers depend on.

54.2.7 7. The in-memory address table

The largest body of new code, and the only large item that is purely implementation: [Address-table operations](#) already gives pseudocode for load, resolve, read, the fragment-map primitives, apply-a-statement, resize, free, page rewrite and eviction.

Risk is concentrated in two places — the [partition invariant](#) of the fragment map, where a bug returns a neighbour's bytes rather than failing, and the [anchor replacement](#) obligation on page rewrites, where a bug silently resurrects truncated data. Both deserve assertions from the first commit rather than tests added later.

54.2.8 8. The fold, plus its differential oracle

[The flush](#) is fully designed. A prototype needs only the naive in-order replayer and the piece-table fold that must agree with it — but [the oracle comes first](#), and building it in the wrong order is the documented way to get this wrong.

The [scheduler](#) is explicitly *not* needed: hoisting collapses the graph, and the recommendation is to skip it until a workload proves otherwise.

54.2.9 9. Recording every mutation

The current `JournaledWriteBackend` [does not](#), which is the one defect that is not a performance matter. Design work is zero — [the log is the sole authority](#) — but every container and the derive macro have to be revisited to confirm each mutation actually emits its records, and nothing mechanically checks this yet.

54.2.10 10. Freeing

[Freeing](#) is designed and not built, so today a dropped or overwritten value orphans its allocation.

A prototype **can** ship leaking — it is harmless against a file that is being exercised rather than kept — but it interacts with the prototype's other goals: a leak makes the file grow, which exercises consolidation, which the prototype is otherwise entitled to skip. Worth deciding deliberately rather than by omission.

54.2.11 11. Choosing values for things that only need a value

Each of these blocks the prototype only in the sense that a number must be typed:

- **page size** — take 4 KiB and defer the 16 KiB measurement;
- **the `InLine` threshold** — the [documented starting policy](#) of ~64 bytes;
- **the journal segment's page budget and the batch size** — see [flush triggers](#);
- **consolidation constants** — see below.

54.2.12 Deferrable, and why

- **Data-page consolidation.** The [reverse-index gap](#) is unresolved, but a prototype may consolidate address-table pages only, which are self-describing, or nothing at all. Garbage accumulates; nothing breaks.
- **Schema evolution.** The prototype checks the schema, which means comparing the root fingerprint and failing closed. [Resolution](#) is a separate and much larger feature.
- **A flush that re-points moved bytes.** The [Move record](#) is specified, and a prototype can fold it like a Copy followed by zeroing the source range, giving up only the rewrite it would save; [re-pointing the bytes instead](#) can wait.
- **Non-owning references,** application versioning, and concurrency. All marked TBD and none reachable from the prototype's feature list.
- **Unwind behaviour.** [Poisoning on panic](#) is aspirational; the interim contract — reopen the file — is adequate for a prototype.

Chapter 55

The Move operation

Status: the record is [specified](#); how a flush states it is a sketch.

`Move(src_id, src_offset, len, dst_id, dst_offset)` relocates a range from one allocation to another, or within one, **without copying the bytes on disk**, and leaves the vacated source range reading as zero.

Recorded because it changes what the storage layer must support, and because it is the reason the address table can [decline copy-on-write clones](#).

55.1 It needs no new address-table statement

The flush emits a `Ref(dst_id, dst_offset, len, address)` pointing at the bytes where they already lie, and denies the source's claim in the same epoch; the old `Ref(src_id, src_offset, len, address)` is shadowed by that denial, loses its fragments, and dies.

So the address-table format is untouched, and Move is a [journal record](#) and an API operation rather than an address-table addition.

55.2 It keeps the byte-referenced-once invariant

Which is the point. That invariant forbids two *live* Refs over one byte, and a move **transfers rather than shares**: the new claim and the old claim's denial land in the same epoch, so the two are never live together.

Data-page coverage therefore stays a plain counter — the source fragment's destruction subtracts the length, the destination fragment's creation adds it back.

One ordering caution: **add before subtracting**, or net the two, so coverage never transiently reads 0 for a page whose last live extent is being moved, which a flush that frees pages part-way through could otherwise hand to the free list.

55.3 What it buys

Rope- and B-tree-like structures shuffle ranges between neighbouring nodes constantly, and without Move every one of those is a byte copy. With it, a split or merge that donates a suffix costs one Ref and one Shrink.

This is what lets the address table decline copy-on-write clones: cloning would then be needed only for genuine simultaneous sharing — snapshots and deduplication — and not for moving data between neighbours.

55.4 Settled by the record's semantics

- **Which moves to allow: any.** The record moves any range, and the cost difference between a tail move and a head or interior move is the caller's: closing the gap is a separate Splice, and it is that splice, not the move, that shifts every later offset in the source.
- **What becomes of the source range: it reads as zero, and the source keeps its size.** So the flush states it as `Zero(src_id, src_offset, len)`, or, when a `Resize` shrinking the source to `src_offset` follows, the two fold into one `Shrink(src_id, src_offset)` — the cheapest case, and the one a suffix donation produces.
- **Overlapping self-moves zero only source \ destination.** That is also what the address table needs: the new `Ref` and the denial cannot both name the overlap in one epoch, per [no conflicts within each epoch](#).

55.5 Open questions

- **Folding.** A move changes content in *two* ids at once, and may grow the destination. The rules for annihilating a move against a later free, resize, or overwrite of either id are not written.
- **Cleaner heuristics.** [Consolidation](#)'s cost-benefit victim selection assumes a data page's residents are the content that one flush wrote together. After moves, residents belong to ids that never wrote them, so the age signal gets noisier. That assumption is documented as degrading gracefully, so this is a note rather than an objection.

Chapter 56

Segments

Status: a direction, not yet a design.

A **segment** would be an independently loadable part of a kladde file: its own id space, its own address table, loadable without reading the rest of the file.

Nothing about this is worked out. It is recorded because several settled decisions were made *so that* it stays possible, and those decisions would otherwise look arbitrary.

56.1 Why it is reachable at all

The copy-on-write redesign made the file **rooted**: every live page is reachable from a committed header, rather than discovered by scanning.

That is the precondition. Under the [superseded heap](#), opening a file meant scanning every live allocation to rebuild the id-to-address table, so partial loading was not expressible: you could not know what was live without reading everything. With a root, “load only this subtree” becomes a question about which pages to follow.

56.2 What has already been decided with segments in mind

- **Allocation ids are 32 bit**, and the [bounds](#) note that each segment will likely get its own id space, so a file may eventually hold more than 2^{32} allocations without widening the id.
- **The header reserves space for future roots**, or should — the exact field list is explicitly [left open](#) partly for this.
- **Whether address-table child references should carry each page’s id range** is an open question: it costs a few bytes per reference now and buys partial loading later. This is the one decision that would be expensive to retrofit, since it changes bytes that files already contain.

56.3 What is not decided

Essentially everything else:

- How a segment is named, and whether one segment may point into another.
- Whether the ownership tree may cross a segment boundary, and what that means for [freeing](#).
- Whether segments share a journal or each carry their own, and what a transaction spanning two segments means for atomicity.
- Whether a segment can be loaded read-only while another is being written.
- How the [schema table](#) is shared or duplicated.

The last two are where this stops being a storage question and becomes a concurrency one, which is [out of scope](#) for now.

Chapter 57

Cleaning by ripeness

Status: a proposal, not adopted. It would replace data-page scoring, the churn floor, and part of free filling in [Consolidation](#). [Cleaning by ripeness](#) evaluates an earlier version, which estimated one rate per page.

Clean a page once waiting no longer pays, not once it has become sparse. A page whose content is still dying should wait, since every byte that dies before the page is cleaned is a byte the cleaning need not copy. A page whose content has stopped dying gains nothing by waiting, however full it is, and its garbage stays until something moves it. This draft turns that into three things: a threshold per page, derived from a price for space; an estimate of how much of each page still drains and how fast, chosen so that pages can be ranked by the threshold without rescanning them as time passes; and a placement rule that makes fewer frozen pages in the first place.

57.1 The problem

57.1.1 A mixed page drains, then freezes

A page written with hot and cold content together loses its hot share within a few flushes and then stops losing anything; it freezes at its cold share, which can be any fill. Nothing in a kladde file is overwritten in place: an update writes new bytes to a fresh page and leaves the old ones dead where they were. So a page is written nearly full and afterwards only loses live bytes. Workloads are skewed — a small hot set is rewritten over and over, a cold majority seldom or never — and a page holds whatever one flush wrote together. Packed in key order, the allocations the application keeps rewriting sit next to ones it has just created and will not touch again.

Under a naive policy that cleans only below a fixed fill threshold, the pages of a file would therefore fall into two populations. Pages whose content keeps dying cycle: written full, drained to the threshold, cleaned, written again. Frozen pages stay wherever they stopped, and nothing about skew says where that is; all that is known about their fill is that it lies somewhere between the cleaning threshold and full. Such a policy keeps every frozen page above that fill for good, and its garbage with it.

57.1.2 Why the current design never cleans a frozen page

Two rules keep a frozen page from ever being chosen, and a third makes new frozen pages. [Victim selection](#) keeps data pages in bucket queues ordered by fill alone, so the age term of its score can only be evaluated on a sample: the next victim is the best-scoring of K pages drawn from the sparsest non-empty bucket.

1. The pages that keep draining keep that bucket stocked, so a frozen page in a fuller bucket is never drawn, and age only ever decides between pages of about the same fill.
2. [The churn floor](#) rejects every page more than $1/(1 + \lambda)$ live — about half, for λ near 1 — whatever its age.
3. [Free filling](#) puts survivors, which are cold by selection, into the room of the flush's own pages, next to fresh content; once the fresh content dies, the page freezes at the survivors' share.

57.1.3 What LFS did about it

LFS cleaned cold segments at a high fill and packed their survivors together, and it needed both.

- Cleaning sorts content by temperature. A survivor of a cleaned segment is cold almost by definition, having outlived everything around it, and a segment written from survivors alone stays full: a frozen segment is rewritten once, and its content never drains again.
- Its cost-benefit score, $(1 - u) \cdot \text{age} / (1 + u)$, values a cold segment's free space more, because that space stays free. In LFS's simulations of a hot-and-cold workload, the cleaner took cold segments at about 75 % utilisation and let hot ones drain to about 15 %.

This draft keeps both ideas, but derives the threshold from a cost model rather than tuning a score, and estimates temperature in a way that makes ranking by it cheap.

57.2 The proposal

57.2.1 When waiting no longer pays

Clean a page of fill u whose content dies at rate r once $h(u/u_0) \geq r/\kappa$, with $h(x) = (1 - x)/x - \ln(1/x)$ and u_0 the fill at which survivors are packed. Here κ is the price of one page of space held for one epoch, measured in page writes: at $\kappa = 0.01$, a page of garbage kept for a hundred flushes costs as much as writing a page. Such a page is **ripe**. The rule sees a page's fill only relative to u_0 , so write $x = u/u_0$ for it.

The model behind it makes three assumptions:

- A page's live content dies at a rate r , a fraction r of it per epoch, however long it has lived, so survivors keep dying at r after they are moved.
- A policy pays κ per epoch for each page of space that live content does not fill, the unfilled room of a page as well as its garbage, and 1 for each page it writes, and is judged by its cost per epoch in the long run.
- It cleans a page once its fill has fallen to a threshold $u^* = x^*u_0$, and packs the survivors, with content that dies at the same rate, into pages of fill u_0 .

Follow one unit of content, measured in pages, from the moment it is packed. It takes up $1/u_0$ of a page, and its page reaches u^* after $T = \ln(1/x^*)/r$ epochs. Until then, whatever of that $1/u_0$ is not live costs κ per epoch, which comes to $\kappa (T/u_0 - (1 - x^*)/r)$ in expectation. With probability x^* the unit survives to T , is copied, which costs $1/u_0$, and starts over. So its expected cost over its whole life, V , satisfies

$$V = \kappa \left(\frac{T}{u_0} - \frac{1 - x^*}{r} \right) + x^* \left(\frac{1}{u_0} + V \right), \quad \text{with } T = \frac{\ln(1/x^*)}{r},$$

$$V = \frac{\kappa (\ln(1/x^*)/u_0 - (1 - x^*)/r) + x^*/u_0}{1 - x^*},$$

and setting $dV/dx^* = 0$ gives $h(x^*) = r/\kappa$: u_0 drops out, except as the unit in which fill is measured. The same threshold balances the two sides of waiting one more epoch at relative fill x : it costs $\kappa (1 - x)$, the page less the x of a page its survivors would take, and saves $r x (1 + u_0 V)$, since each byte that dies meanwhile would have cost a copy now and V afterwards.

r/κ	x^* , cleaning at	the myopic rule $(1 - x)/x \geq r/\kappa$ would clean at
0	any fill, up to the cap below	any fill
0.001	0.96	0.999
0.01	0.87	0.99
0.1	0.66	0.91
1	0.32	0.5
10	0.07	0.09

Content that has stopped dying is cleaned at almost any fill, and content that dies fast only once it is nearly gone, which is LFS's behaviour, derived rather than tuned.

The churn floor is the special case in which every page drains at the same rate, without the logarithm. $(1 - x)/x$ is the space a cleaning frees per byte it writes, which is exactly what the churn floor compares with λ . The $-\ln(1/x)$ term accounts for survivors that go on dying after they are moved; the myopic rule, which leaves it out, would clean every page too early, and pages that drain slowly far too early. However, the more important difference between the current “churn floor” design and the proposed ripeness design is that the current design compares its score $(1 - x)/x$ against the same threshold λ for every page, whereas the proposed ripeness design uses a different threshold r/κ for each page, where the rate r is [estimated per page](#). Under uniform random updates every page does drain at one rate, so there ripeness reduces to a single fill threshold, as the churn floor is today. [The ablation](#) works out what the logarithm contributes.

No page at or above u_0 is ripe, whatever its rate. Its survivors would take a whole page or more, so cleaning it would free nothing: $1 - x$, the space a cleaning gains, is not positive. The formula has to say so explicitly, since h , which falls to 0 at $x = 1$, rises again above it. The draft takes $u_0 = 1 - \theta$, the fill that packing promises for every page but a flush’s last. Pages come out between $1 - \theta$ and full, so their expected fill is nearer $1 - \theta/2$; taking that instead would clean a little earlier, but would also make pages ripe whose survivors might fill a page of their own. The cap also keeps full pages, among them nearly every page a flush has just written, out of the ranking altogether.

57.2.2 A page holds a draining share and a static one

Model a page’s live content as a share a that drains at rate r and a share s that does not drain at all, and clean it once $h(z) \geq r/\kappa$, with $z = a/(1 - s)$. Fills are relative to u_0 , as above, so the page’s fill is $x = a + s$. z is the fill of the draining share on the page with the static share taken out: the static share is copied once whenever the page is cleaned, and nothing about that changes by waiting, so it has no say in when. With $s = 0$, this is the rule for content that dies at one rate. With $a = 0$, the page would be ripe at any fill below u_0 , which the floor $R_{\min}$, [below](#), limits.

It is the mixture that the problem describes, as far as a page’s losses can tell it. A page of hot and cold content is a draining share over a static one once its cold content has stopped dying, and until then the cold content is part of the draining share or the static one, whichever describes the page’s losses better. A page that starts with two shares that both drain, one fast and one slowly, becomes a draining share over a static one as the fast share dies: the slow share is then the draining one, and whatever has stopped dying the static one. [The fit](#) follows that change as far as the page’s losses show it.

The rule follows from the same balance as before. Waiting one more epoch costs $\kappa(1 - x)$ and saves what the bytes that die meanwhile would have cost: ra of them, each of which would have been copied now and cost V afterwards. At its own threshold x^* , content of rate r balances the two, $\kappa(1 - x^*) = rx^*(1 + u_0V)$, so $1 + u_0V = (\kappa/r)g(x^*)$ with $g(x) = (1 - x)/x$, and waiting saves $\kappa ag(x^*)$. The page is ripe once that no longer covers the cost:

$$1 - x \geq ag(x^*) \iff \frac{a}{a + 1 - x} \leq x^* \iff h\left(\frac{a}{1 - s}\right) \geq \frac{r}{\kappa},$$

since g and h both fall on $(0, 1)$ and $a + 1 - x = 1 - s$. Once ripe, a page stays ripe, since a and x only fall as it drains; for a stopping problem of that kind, stopping at the first epoch at which one more epoch of waiting does not pay is optimal.

A second draining share would need more than a page’s losses determine, and a numerical solve for every index; the static share needs neither. With shares a and b draining at r_1 and r_2 , the same argument makes a page ripe once $1 - x \geq ag(x^*(r_1/\kappa)) + bg(x^*(r_2/\kappa))$, which has no closed form in κ , so every change to a page’s estimate would need a root-finding to key it. Two running sums of a page’s losses determine a draining share and its rate, as [the fit](#) shows, while two draining shares need a third sum and a fit in three unknowns, and separating two decay rates from one noisy decay curve is notoriously ill-conditioned. The static share is the limit $r_2 \rightarrow 0$: $x^*(0) = 1$ and $g(1) = 0$, so the second term vanishes, and with it both problems.

57.2.3 Estimating how fast a page still drains

Each page fits a draining share and its rate to its own recent losses, forgetting older losses at a rate β , and keeps a static share only once its losses show one.

The fit is the maximum-likelihood fit of a falling loss rate to the page's discounted loss history. Under the model, a page's losses fall exponentially: what it lost k epochs ago has expectation ℓe^{rk} , where $\ell = r a$ is what the draining share loses per epoch now. Weighting the epoch k epochs ago by $e^{-\beta k}$, and counting losses as if they were Poisson, the fit maximises

$$\mathcal{L}(\ell, r) = S_0 \ln \ell + r S_1 - \ell E(r), \quad \text{with} \quad S_0 = \sum_k e^{-\beta k} L_k, \quad S_1 = \sum_k e^{-\beta k} k L_k, \quad E(r) = \sum_k e^{(r-\beta)k},$$

where L_k is what the page lost k epochs ago, the first two sums run over its losses, and the third over the epochs it has been watched. The maximum lies where

$$\ell = \frac{S_0}{E(r)} \quad \text{and} \quad \frac{S_1}{S_0} = \frac{E'(r)}{E(r)} :$$

the mean lag of the page's losses, weighted by their size and discounted, equals the mean lag the model predicts at rate r . The right side grows with r , and E and E' are geometric sums with closed forms, so a few Newton steps solve it, once per loss. The draining share is then what the current losses imply at that rate, $a = \ell/r$, and $s = x - a$.

- **If the losses lie at longer lags than an even spread would put them**, they have fallen: the draining share dies at r and is the smaller for it.
- **If they do not**, $a \geq x$, and nothing on the page is known to be static: $s = 0$, and the page drains at ℓ/x , as content that dies at one rate would.

Two running sums are the whole state of the fit, and both age in closed form. When d epochs pass, every loss is d epochs older: $S_1 \leftarrow e^{-\beta d} (S_1 + d S_0)$ and $S_0 \leftarrow e^{-\beta d} S_0$. E counts the epochs since the page was written, which its epoch gives, so it needs no state.

The fit models the lag of its own window, which the estimator it replaces did not. That estimator compared a discounted average of the losses, standing in for ℓ , with D , the bytes lost since the page was written: under the model, $\ell t/D = \psi(rt)$ with $\psi(w) = w/(e^w - 1)$. But an average over the last $1/\beta$ epochs of losses that fall at r exceeds the current loss rate by a factor of about $\beta/(\beta - r)$, twice at $r = \beta/2$, so it took pages for less drained than they were. And D never forgets, so a share that had died early went on weighing on the fit. The fit instead predicts what its own window should hold at each rate, and forgets as fast as the average did.

A static share must be earned: the fit keeps one only if it explains the losses better than one share draining alone, by c statements' worth of log-likelihood. One share draining alone is the same fit with $a = x$, so $\ell = r x$, which leaves one unknown. $\mathcal{L}$ counts bytes, but a statement dies whole, so the gain is divided by the file's mean statement size to count it in the units the losses really come in. Without the test, the fit reads a chance gap between the rare losses of a page that drains slowly as a fall in its loss rate, finds a static share that is not there, and has the page cleaned early: [in simulation](#), that made it cost more than the estimator it replaces, and the test turned that into a quarter less. The draft takes $c = 3$.

Between losses, the split sums and the rate decays, which is what keeps the ranking cheap. Drain::lose only needs to be called in epochs where the page loses a nonzero amount of bytes. The epochs in between are epochs watched without a loss, which the next call accounts for when it ages the sums and extends E . Until then, the ranking lets the fitted rate decay by $e^{-\beta}$ per epoch, on every page alike, as a discounted average of the losses would, and keeps the split. A refit would follow the missing losses more closely, but it would change every page's key every epoch.

```
/// Per page, beside kind, epoch and coverage: 18 bytes.
struct Drain {
    s0: f32, // natural losses, in bytes, each discounted by e^(-β·lag), as of epoch `at`
    s1: f32, // the same, each also weighted by its lag
    rate: f32, // the draining share's rate at epoch `at`
    at: u32, // the epoch of the page's last natural loss, from the session's base
    fast: u16, // the live bytes of the draining share; the rest of the coverage is static
}
```

```

impl Drain {
    /// A flush's fold superseded `bytes` of the page's live bytes, `live` of which are left;
    /// the page was written `age` epochs ago.
    fn lose(&mut self, bytes: u32, live: u32, age: u32, now: u32) {
        let d = (now - self.at) as f32;
        let decay = (-BETA * d).exp();
        self.s1 = decay * (self.s1 + d * self.s0);
        self.s0 = decay * self.s0 + bytes as f32;
        self.at = now;
        let live = live as f32;
        // The draining share's rate, and what it loses per epoch now.
        let (rate, loss) = fit(self.s0, self.s1, age);
        // One share draining alone: its rate, and how much worse it explains the losses.
        let (one, worse) = fit_one_share(self.s0, self.s1, age, live);
        let fast = loss / rate;
        (self.rate, self.fast) = if fast < live && worse >= STATIC_EVIDENCE * mean_statement_size() {
            (rate, fast as u16)
        } else {
            (one, live as u16)
        };
    }

    /// The draining share's rate, a fraction of it per epoch.
    fn rate(&self, now: u32) -> f32 {
        self.rate * (-BETA * (now - self.at) as f32).exp()
    }
}

```

A page whose losses hold steady fits as one share, at the rate of its losses. A page with no draining share left, $\text{fast} == 0$, is ranked by the floor alone. Below, $\hat{r}$ stands for a page's `rate()`, and $R_{\min}$ for `R_MIN`.

A page's starting estimate enters the fit as pseudo-observations, with a weight of its own: n_0 epochs.

A page starts from an estimate of its draining share a_0 and its rate r_0 , as "A new page starts from what it holds" below describes. It enters the sums as the losses that estimate predicts for the page's first n_0 epochs, observed once more on top of what the page will really lose then: those epochs add to S_0 , S_1 , and E , and for a new page they lie ahead, at negative lags, which the sums take like any other. So the fit reproduces the start until the page's own losses say otherwise, and the start then weighs as much as n_0 epochs of them, fading as the page ages, with the page's own early losses. The estimator this replaces also started from an estimate, but with a weight fixed at about $1/\beta$ epochs, where Adam's bias correction would give it none; here the weight is a constant of its own. And it fitted its split from the page's own losses alone, so the first loss replaced an inherited split with one fitted to that loss, which could show no static share; the pseudo-observations carry an inherited split through, if the test lets it. In simulation, the weight matters little: from 3 to 30 epochs, a heavier start helps where it is right and hurts where it is wrong, about equally. The draft takes $n_0 = 10$.

Only natural losses count: bytes that the application's writes, frees, and shrinks supersede, which reach a page through the fold. Bytes that consolidation moves out of a page say nothing about how fast the rest will die, so evacuation, the page rewrite, the rotating window, and [the cursor](#) leave the estimate alone. Where they move out part of a page, which share they came from is unknown, so they shrink S_0 , S_1 , and fast in proportion, which leaves the rate where it was.

The fit follows a page whose content changes character as far as its losses show it, and no further.

Take a page that starts with a share of 0.3 dying at 0.5 per epoch, a share of 0.3 dying at 0.02, and a share of 0.4 that does not die at all, with $u_0 = 1$. It starts from the draft's starting estimate, a draining share of 0.6 at 0.26 over a static share of 0.4, and loses exactly what the shares predict, with $\beta = 0.1$, $n_0 = 10$, and statements of 144 bytes for the test. Its true index is that of the three shares it actually holds:

epoch	fill	true index	the estimator replaced	the fit	the tested fit
10	0.65	24	1	40	40

epoch	fill	true index	the estimator replaced	the fit	the tested fit
20	0.60	44	3	311	6
50	0.51	137	72	285	41

- **While the fast share dies**, its losses show the fall, and the test lets the fit keep a static share: 0.62 of the page, which takes most of the slow share for static, and the page for somewhat riper than it is.
- **Once the fast share is gone**, the slow share loses a statement every eight epochs or so, about one within the fit's memory, which cannot tell a static share from a slow drain. Weighed against those few, the fading losses of the fast share look like a steep fall, and the untested fit takes the page for far riper than it is. The tested fit falls back to one share, and takes it for less ripe.
- **The estimator replaced** took the page for far less ripe throughout: its loss average lagged the falling losses, and D kept the fast share's.

The floor $R_{\min}$ is the assumption that no content lasts forever: no page is riper than it would be if all of its live content drained at $R_{\min}$. Without it, a page whose draining share had died, or whose estimate had decayed to nothing, would be ripe at any fill below u_0 . With it, the index of a page is

$$I = \min\left(\frac{h(z)}{\hat{r}}, \frac{h(x)}{R_{\min}}\right),$$

and a frozen page is ripe up to the relative fill x_{cold} for which $h(x_{\text{cold}}) = R_{\min}/\kappa$ — at $R_{\min} = 10^{-4}$ and $\kappa = 0.01$, that is 0.87. For a page without a static share, the minimum is a floor on the rate, $\max(\hat{r}, R_{\min})$. For one with a static share, it is exact at both ends, and in between it takes the page for riper than a static share that drains at $R_{\min}$ would make it, since that share's slow drain would add to what waiting saves.

A new page starts from what it holds. Fresh content joins its draining share, at a running estimate of what pages lose in the epoch after they are written. Moved content brings its source page's split: what was static there is static here, and the rest joins the draining share at the source's rate. The draining share's rate r_0 is the byte-weighted mean of its parts' rates, and the start enters the fit as [pseudo-observations](#).

Matching the mixture's losses instead would call too much of it static. One share over a static one can match both what a mixture loses now and how fast that falls, with

$$r_0 = \frac{\sum_i r_i^2 a_i}{\sum_i r_i a_i}, \quad a_0 = \frac{(\sum_i r_i a_i)^2}{\sum_i r_i^2 a_i},$$

which is the byte-weighted mean when the parts drain alike, and calls a part static when it drains far slower than the others. But what makes content static for ripeness is draining slowly against κ , not against other content: at $\kappa = 0.01$, a part that drains at 0.02 has a threshold of 0.23, however much faster its neighbour drains. [In simulation](#), the page of the example above cost 9.0 % more than cleaning at its true index from that start, against 7.6 % from the byte-weighted one, and a page mixed from parts at 0.2 and 0.002 cost 2.4 % against 2.5 %, so the draft keeps the byte-weighted mean.

Open restores each page's estimate from the [consolidator state](#), and seeds a page the state cannot vouch for from its fill and its age. The seed assumes that the page has drained at one rate since it was written, from the fill its framing records, `content_size`, to its current coverage, at $\hat{r} = \ln(\text{content_size}/\text{coverage})/\text{age}$. It enters the sums as the page's whole past, watched and seen to lose what that rate predicts, and as the pseudo-observations of a start with all of `content_size` draining. So all of its content is draining, fast is the coverage, rate is $\hat{r}$, and a_t is the current epoch. The age counts from the epoch the session's first flush will have, so that no page is younger than one epoch. The file records nothing that could reveal a static share, so the seed has none, and averages over the page's whole life: a page that drained early and then froze looks at first like one still draining slowly. The session's own losses correct it, and must earn a static share against the seeded past, which counts as watched.

The seed counts the bytes consolidation moved out as losses, which errs on the safe side. The file records neither how a page lost its bytes nor when, so this cannot be avoided without the state. Moved-out bytes make the page look as if it had drained faster than it did, which raises its rate, lowers its index, and so delays its cleaning: the error can hold garbage a little longer, but never makes a page ripe that the

true rate would not. It is also rare. Evacuation, the page rewrite, and the rotating window move whole pages, which are then free rather than seeded; only the cursor and description defragmentation move part of a page, and the cursor empties its page within a flush or two.

57.2.4 Checking the fit in simulation

In simulation, the tested fit costs a quarter to a half less than the estimator it replaces, unless pages hold only a few statements, and about as much as the single-rate draft, which has no static share at all. `tools/simulate-ripeness.py` simulates pages of statements, each statement belonging to a share with a rate of its own, and cleans each page once its estimated index reaches $1/\kappa$, at $\kappa = 0.01$. It charges each page what the model charges, κ per epoch for each page of room that live content does not fill, plus the copy of its survivors, plus what they cost afterwards under the optimal policy, and compares that with cleaning each page once its true index reaches $1/\kappa$, the exact index of the shares it actually holds. Eight scenarios cover one share at rates from 0.01 to 0.2, with starting estimates that are right and wrong, a share over a static one, the page of the example above, a page mixed from a fast and an almost static source, and a page seeded at open. Their summed excess cost, at $\beta = 0.1$, $n_0 = 10$, and $c = 3$:

statements of	the estimator replaced	the fit	the tested fit	the single-rate draft
16 to 64 bytes	64 %	38 %	28 %	29 %
32 to 256 bytes	62 %	67 %	46 %	46 %
256 to 1024 bytes	106 %	109 %	112 %	121 %

- **The test is what makes the fit pay.** Untested, the fit wins wherever a page holds a static share or started from a wrong estimate, but on pages that drain slowly as one share it invents static shares, and on statements of typical size it loses more than it wins.
- **Pages of a few statements are beyond any estimator:** with four to sixteen statements a page, every estimator costs about a sixth more than cleaning at the true index in most scenarios, and the estimator replaced a little less than the fit, tested or not.
- **The static share does not pay in these scenarios.** The single-rate draft, which this one replaces and which survives on the branch ripeness of these docs, costs as little as the tested fit on statements of up to 256 bytes, and a little more on larger ones. What a static share could be worth shows in the untested fit on small statements, where losses are plentiful: it costs a sixth as much as the tested fit, both on the page with a static share and on the seeded page. But the test that keeps it from inventing static shares also keeps it from finding them quickly, and the gain goes.
- **The tested fit is robust to its constants.** From $\beta = 0.05$ to 0.2, its summed excess cost stays between 46 % and 62 %, where the estimator replaced ranges from 61 % to 112 % and the single-rate draft from 46 % to 74 %. c from 1 to 3 and n_0 from 3 to 30 move it by three points at most.

57.2.5 Ranking pages by ripeness

Rank every page below u_0 by its ripeness index $I = \min(h(z)/\hat{r}, h(x)/R_{\min})$, which is the $1/\kappa$ at which the page becomes ripe. Because every rate estimate decays by the same factor $e^{-\beta}$ per epoch while the splits stay put, every index below its floor grows by the same factor, so the order changes only when a page's own content does, and an ordered map keeps it.

Two ways would rank by a time-dependent score exactly, and the second is chosen. The current design samples only because its bucket queues order pages by fill, and the age term changes every page's score every epoch.

- LFS's score is a line in time for each page, with a slope $(1 - u)/(1 + u)$ of its own, so pages overtake one another as time passes. Maintaining the maximum of a changing set of lines is what a kinetic tournament does: $O(\log P)$ per change of a page's fill, plus a repair every time one page overtakes the one it was compared with. That second cost would be paid per epoch, whether or not any page changed.
- A score whose dependence on time is common to all pages keeps its order until a page changes. The ripeness index has that property as long as a page's index stays below its floor:

$$\ln I(\text{now}) = K + \beta \cdot \text{now}, \quad \text{with} \quad K = \ln h(z) - \ln \text{rate} - \beta \cdot \text{at}$$

K changes only when the page loses content, is written, or has content moved out of it, and then once per flush however many of its fragments changed.

```

/// A log-index, totally ordered.
type Key = OrderedF32;

struct Ripeness {
    draining: BTreeSet<(Key, PageNumber)>, // K; the index grows as  $e^{(\beta \cdot \text{now})}$ 
    settled: BTreeSet<(Key, PageNumber)>, //  $\ln h(x) - \ln R_{\text{MIN}}$ ; the index is constant
}

impl Ripeness {
    /// The page with the highest index, if it is ripe at price  $e^{(-\text{neg\_ln\_kappa})}$ .
    fn next_ripe(&mut self, now: u32, neg_ln_kappa: f32) -> Option<PageNumber> {
        // A draining page whose index has passed its floor is overstated by its key,
        // so it can surface here but never hide further down: it is settled lazily.
        while let Some(&(k, p)) = self.draining.last() {
            if k + BETA * now <= key_settled(p) { break; }
            self.draining.remove(&(k, p));
            self.settled.insert((key_settled(p), p));
        }
        let draining = self.draining.last().map(|&(k, p)| (k + BETA * now, p)); //  $\ln I(\text{now})$ 
        let (ln_index, p) = draining.max(self.settled.last().copied())?;
        (ln_index >= neg_ln_kappa).then_some(p)
    }
}

```

The controller keeps the price as $-\ln \kappa$, which is what the keys compare with, and a step in it changes κ by a factor, which suits a price that ranges over orders of magnitude.

This gives up the bucket queues' $O(1)$ for $O(\log P)$ per changed page and per victim. **An indexed binary max-heap would do as well as the B-trees, with less memory.** Each page's entry in the page table would hold its position in its heap, 4 bytes, so that a changed key can be sifted up or down from where it is; every swap updates the two pages' positions, which is why a stock heap will not do. The trees need no position, but hold the key and page number again in every node. Both are $O(\log P)$ per changed key, and the heap is $O(1)$ for the maximum, which `next_ripe` reads on every call. If that ever shows in a profile, note that the current value of $\ln I$ lies in a window of bounded width — z and x are at least one byte in a page, and a rate that matters is at least R_{min} and at most the whole draining share per epoch — so a circular array of buckets over quantised keys, turned like a timer wheel as $\beta \cdot \text{now}$ advances, would do the same in $O(1)$.

Both page kinds are ranked alike. A table page drains as the fold supersedes its statements, and a page of space and a page write cost the same whichever kind they are, so indexes compare across kinds and the budget loop takes the highest of either.

At open, the index resembles LFS's score for nearly full pages, and exceeds it by far for nearly empty ones. Take a page written at u_0 and seeded with $\hat{r} = \ln(1/x)/\text{age}$ and no static share, so that $z = x$. Its index is $I = h(x) \cdot \text{age} / \ln(1/x)$, proportional to its age, as LFS's score $(1-x) \cdot \text{age} / (1+x)$ is; the two differ in how the factor depends on the fill.

- Nearly full, $x \rightarrow 1$: $h(x) \approx (1-x)^2/2$ and $\ln(1/x) \approx 1-x$, so $I \approx (1-x) \cdot \text{age}/2$, which is LFS's score to first order. Both fall to 0.
- Nearly empty, $x \rightarrow 0$: $h(x) \approx 1/x$ grows faster than $\ln(1/x)$, so $I \approx \text{age}/(x \ln(1/x))$ grows without bound, while LFS's factor rises only to 1, leaving its score at age. A nearly empty page costs almost nothing to clean, so ripeness takes it at any age, where LFS would still rank it by age.

57.2.6 The budget, and the price

A flush cleans ripe pages, highest index first, until its page budget is spent: ripeness takes the churn floor's place. The budget stays what it is, a cap on the work one flush does, and so does the check that an offer fills its page.

κ , **not the budget, is what moves the file toward its target fill τ** . Raising κ raises every page's threshold, so a controller that measures the live fraction after each commit and moves κ up while it lies below τ , and down while above, converges on τ wherever the budget allows — the job the current design gives the budget. λ could not do that job, since it would clean frozen and draining pages at the same fill; κ can, because the thresholds it sets depend on how fast each page drains.

The policy does not change when the budget binds: it always takes the highest index first, which matters only then. While the budget does not bind, a flush cleans every ripe page, and the order is immaterial. When it binds, the flush can clean only some of them, and the order decides which. A binding budget means that page writes are scarcer than κ assumes, which is the same as a lower price for space, $\kappa' < \kappa$; the pages that would be ripe at κ' are those with $I \geq 1/\kappa'$, which are exactly the highest indexes. So the one rule is right in both cases, and no mode switch is needed. If the budget binds flush after flush, ripe pages queue up and the controller cannot reach τ : the target and the cap conflict, and the cap wins, as it should. The controller must then not wind up: raising κ would only lengthen the queue, so it holds κ while the budget binds.

57.2.7 Where survivors go

Whether to free a page now and where its survivors should go are separate decisions, and neither needs pairs. Whether freeing a page now pays depends only on that page, and is its ripeness. What its survivors cost afterwards depends only on what they share a page with. A counterfactual defined by pairs — wait until two pages fit into one — would not even exist for pages that never will, such as two frozen at 60 %.

57.2.7.1 What a mixed page costs

Once its fast share a has died, a mixed page is just a page of fill b whose content drains at r_B , so mixing is cheap when b lies below that content's own threshold $u^*(r_B)$, and expensive when it lies above. For the page of the mixture above:

- $b \leq u^*(r_B)$. The page is ripe as soon as the fit has seen its losses fall, which takes a few multiples of $1/\beta$ epochs and enough losses to earn the static share, during which it holds its garbage a . Cleaning it then copies the slow share once more. When that share was moved in from a victim, whose own cleaning would have copied it anyway, mixing defers a copy rather than adding one, and frees the victim early. The smaller b , the shorter the wait, since $h(b)$ grows like $1/b$.
- $b > u^*(r_B)$. The page holds its garbage a until its slow share has drained down to $u^*(r_B)$. For content that has stopped dying, which is what survivors of ripe pages are, that is never.

So the room of a page is best filled with content that dies at about the rate of what the page already holds. Cold content in a hot page costs a wait that grows with its share, and above u^* it costs the page's garbage for good.

So each source of survivors has one place it may go, decided by its temperature and its size, not by a partner. The case distinction gives the rule: survivors may join a hotter page only as a share b small enough to stay below their own threshold $u^*(r_B)$; otherwise they need a page of their own temperature. The next two sections apply it to the two kinds of page that take survivors:

source	its share b in the host	where it goes	case
survivors of ripe pages, except the small ones below	any	pages the budget loop opens, each filled with survivors of many ripe pages	cold with cold: no hot content to wait for
a ripe page whose survivors take at most θ of a page	at most θ	the room of a flush's own page	$b \leq u^*(r_B)$, as long as θ lies below the threshold of content that has stopped dying
the survivors of the previous flush	whatever fits	the room of a flush's own page	little mixing: about the host's temperature

None of these needs a counterfactual about pairs: each source is judged by its own rate and size against a threshold, and the room is simply offered in that order.

57.2.7.2 Budgeted pages take survivors of ripe pages

A page opened for consolidation should hold survivors of ripe pages only, cold with cold, so that it stays full. This is LFS's segregation. The current budget loop already comes close: a page it opens holds survivors of victims and nothing else. The proposal changes two things about it. First, the victims are the ripe pages, taken in index order, rather than pages that pass the churn floor. Second, the loop's first page no longer takes the rest of a victim cut to fill the flush's last page; the cursor below fills that room instead, so no victim is split between a hot page and a cold one.

57.2.7.3 The flush's own pages take survivors from the previous flush

This is the placement half of the proposal, and apart from its first step it would help any cleaning policy. Ripeness decides when to clean; this decides what goes into the room of the pages a flush writes anyway, so that fewer frozen pages are made in the first place. Only step 1 uses the index.

The room in every data page a flush writes anyway goes first to small ripe victims, then to the survivors of the previous flush's least-filled page.

1. Whole ripe victims that fit the room and whose survivors take at most θ of a page, highest index first. Each frees a page for almost no room, and a host that freezes at under θ is ripe soon after its own content has died.
2. The survivors of the **cursor page**, as many of its statements' as fit, a whole statement's at a time, so that the cursor never cuts a statement. The cursor page is the least-filled data page of the latest earlier epoch that wrote data pages, chosen at open among the governing header's epoch's pages and again whenever the current one is empty.

A statement too long for the room cannot stop the cursor for good. The cursor takes whichever of the page's statements fit, not only the next in order, so a long one is skipped until a flush leaves room enough for it. If no flush does, the page grows old, and the rule below that gives up a page after W epochs moves the cursor on and leaves the rest of the page to ripeness. A statement can be at most a page long, so it takes a flush whose last page leaves that much room; how often a page is abandoned this way is a question for measurement.

The previous flush's content is the closest in age to the flush's own that exists outside the flush itself. A page *closes short* when it is written with more than θ of it empty, which only a flush's last page can, when the flush runs out of content and free filling finds nothing to fill the room with. Unless the previous flush's last page closed short, its least-filled page is the one the current flush's fold drained most, which is the hottest of what the previous flush wrote; a page that closed short is least filled because of how it was written, not because anything died. So the room of the flush's own pages fills with content of about their own temperature, and the shortfall of the flush's last page, about half a page on average, comes out of a page that the next flush or two empty and free, with no page written for it. With about half a page of room per flush and at most a page to empty, the cursor stays about two epochs behind the flush.

Four details keep it from going wrong:

- **It still mixes, only less.** The previous flush's survivors are exactly what the current flush did not rewrite, so they are colder than fresh content. No other source matches fresh content better, and ripeness cleans whatever freezes.
- **It moves content without writing a page, but not for free.** Every survivor it moves is stated again in the flush's epoch, and its old statement becomes garbage in a table page. The volume is bounded by the room of the flush's own pages, about half a page per flush.
- **It does not count as a loss** in the cursor page's estimate, and the host's estimate starts as the byte-weighted mix of fresh and moved content.
- **It gives up a page that has grown old.** Flushes that end nearly full leave little room, and a cursor page drained slowly across many of them would hand out ever older content; once it is more than W epochs old, the cursor moves on to the least-filled page of the latest earlier epoch, and the rest of the old page is left to ripeness.

In [compaction mode](#), the tail page still takes the room ahead of both, since returning the tail is worth a mixed page.

57.3 What would change in Consolidation

- [Scoring a data page](#) would give way to the ripeness index, for table pages as well, and sampling would go. The bucket queues would give way to the two ordered sets, and victim selection would cost $O(\log P)$ rather than $O(1)$.
- [The churn floor](#) would give way to ripeness, and λ to κ ; the controller would move κ toward τ , and the budget would remain a cap.
- [A free sink changes the arithmetic](#) — “free filling takes any victim that fits” — would no longer hold: free filling would take small ripe victims and the cursor’s survivors.
- [Free filling’s order](#) would shrink to those two sources, with the cursor page taking the place of the victim too big to take whole.
- [The page table](#) would gain Drain, 18 bytes per page, and the [consolidator state](#) would carry it, with κ , from one session to the next.
- [The constants still to be chosen](#) would lose λ and gain κ ’s controller, β , n_0 , c , $R_{\min}$, and W .

57.4 Ablation: without the logarithm

Dropping $-\ln(1/x)$ would make the split unnecessary, and keep the per-page threshold and with it the cleaning of frozen pages; but it would clean pages that drain slowly too early, with extra writes that grow the slower they drain, and could not tell a page on a static share from one that drains slowly throughout. The ablated rule is the myopic one: a page is ripe once $g(z) = (1-z)/z \geq r/\kappa$, and its index is $g(z)/\hat{r}$.

The split would drop out, and with it the fit. With $z = a/(1-s)$ and $\hat{r} = \ell/a$, the index is $g(z)/\hat{r} = (1-x)/\ell$: the page’s garbage over its loss rate in bytes, whatever the split. So Drain would shrink to a discounted average of the losses and its epoch, 8 bytes, and neither the fit nor its test would be needed. The threshold would also have a closed form, $x^* = 1/(1 + r/\kappa)$, though nothing needs it. The keys would still be logarithms, and the ordered sets, the floor, and the controller would stay as they are.

It behaves as the full rule does in every limit but one.

- **All pages drain at one rate.** Both rules reduce to a single fill threshold, as the churn floor does, and the controller moves κ until that threshold gives τ . All three then clean the same pages: the logarithm changes only which κ gets there.
- **Nearly empty pages, $x \rightarrow 0$:** $h(x) \approx g(x) \approx 1/x$, so the two agree.
- **Content that dies fast, $r/\kappa \gg 1$:** both clean once the page is nearly empty, at 0.07 and 0.09 for $r/\kappa = 10$.
- **Content that drains slowly, $r/\kappa \ll 1$,** is the exception: near $x = 1$, $h(x) \approx (1-x)^2/2$ but $g(x) \approx 1-x$, so the thresholds are about $1 - \sqrt{2r/\kappa}$ and $1 - r/\kappa$, exactly 0.87 and 0.99 for $r/\kappa = 0.01$. The myopic rule ignores that a byte which dies now would otherwise have been copied again and again, so it copies such a page as soon as a percent of it is garbage.

It could not tell a page on a static share from one that drains slowly throughout. Take two pages at $x = 0.8$ that lose the same bytes per epoch: one a draining share of 0.1 at rate r over a static share of 0.7, the other all draining, at $r/8$. Their ablated indexes are equal, $2/r$. The full index ranks the first at $h(1/3)/r = 0.90/r$ and the second at $8h(0.8)/r = 0.22/r$, four times lower: waiting pays on the second, whose survivors would go on dying, and hardly at all on the first, whose survivors would not.

Frozen pages are cleaned either way, since the floor makes every page’s threshold its own. With $R_{\min}$ in place of r , the ablated rule makes a frozen page ripe below 0.99 rather than 0.87, at $R_{\min}/\kappa = 0.01$. That page is copied once, its survivors never die, and so the early cleaning costs one copy, not a cycle of them. It does rank frozen pages much higher: at $x = 0.95$, $g/R_{\min} = 526$ against $h/R_{\min} = 13$, so under a binding budget they would take writes that would free more elsewhere.

Expect the same results on uniform workloads, and more page writes for the same space on skewed ones, most of them spent on warm content. Over its whole life, content that drains at r/κ costs V at the myopic threshold against V at the optimal one (taking $u_0 = 1$):

r/κ	10	1	0.1	0.01
cost of the myopic rule, relative to the optimum	1.0	1.2	2.5	7

Retuning κ cannot undo this, since the error lies in how the thresholds of pages of different rates relate, not in their level, and there is no error when all pages drain at one rate. The warm content that pays is whatever lives for longer than $1/\kappa$ epochs but does die, 100 epochs at $\kappa = 0.01$. Running the evaluation with h replaced by g would measure how much of a real workload that is.

57.5 Open questions

- **The clock.** Rates per epoch count flushes rather than work: a run of tiny flushes makes every page look colder than it is. A clock that counts the bytes the application writes would measure drain per unit of work, but pages record only their epoch, so the seed at open would still have to use epochs.
- **The flush's own content.** Packed in key order, it mixes allocations the application keeps rewriting with ones it has just created, which is the largest source of mixed pages this draft leaves in place. `last_written` could split a flush's chunks by whether their allocation was also written recently, at the price of key order; whether that pays is a question for measurement.
- **Description defragmentation's rewrites** are cold by selection and today share pages with the flush's fresh content. Packing them with the survivors of ripe pages instead would keep them apart, at the price of a second page per flush that may close short.
- **Whether the static share pays.** In [simulation](#), the single-rate draft cleans as cheaply as this one, and a static share pays only where losses are plentiful enough to find it without a test. Running both on kladde-bench's workloads would show whether real pages hold enough statements for it.
- **Too few losses to tell.** A share that drains slowly loses too few statements within the fit's memory of about $1/\beta$ epochs to be told from a static one, as in [the example](#), so the tested fit takes it for draining and its page for less ripe. A longer memory would see more of its losses, but also more of whatever died before them.
- **A static share that turns out not to be.** Losses on a page that had looked static lie at shorter lags than the fit expects, so it takes the page for less static, or for one share, which is less ripe than before, until those losses have aged. Whether that delay costs anything on real workloads is a question for measurement.
- β trades how soon a frozen page is recognised against noise: with a large β , a page that loses content in rare bursts looks frozen between them.

Chapter 58

Bayesian ripeness

Status: a proposal, not adopted. It would replace how both variants of [Cleaning by ripeness](#) estimate a page's drain and decide on it: the single-rate draft on branch ripeness, and the draft with a static share on branch ripeness². It assumes both drafts, and uses their notation: fills relative to u_0 , x for a page's fill, a and s for its draining and static shares, $z = a/(1 - s)$, h, g, κ, β , and $R_{\min}$.

58.1 Executive summary

Keep a posterior over each page's drain rather than a point estimate, and decide from it: with a static share, that is what makes the static share pay. A page's estimate rests on little: a data page in kladde-bench's files holds 10 to 30 chunks, and the fall that would show a static share comes to a few of them ([Ripeness with a static share](#)). The drafts cope with that by devices of their own, a forgetting rate, a weight for the starting estimate, a test, and a floor; here each becomes a parameter of one prior.

- **Evidence is counted in bytes, and weighed in loss events.** A page loses the bytes that writes supersede, on a data page often part of a Ref's payload at a time; divided by the dispersion of the events' sizes, $\mathbb{E}[s^2]/\mathbb{E}[s]$, bytes lost and bytes exposed count as the independent observations they are.
- **With one rate per page, the posterior is a Gamma distribution, exact even for a rate that drifts, and its mean is exactly the single-rate draft's estimate.** What it adds is how certain the estimate is, and a start learned from the file.
- **With a static share, the posterior is a mixture over how many of the page's untouched chunks drain.** A chunk that has lost bytes is known to drain, and a prior on the draining fraction takes the place of the static-share draft's test.
- **A page's start is learned from the file, not tuned.** Fresh content starts from the file's empirical prior, the mean and spread of the rates at which fresh pages have drained, which comes out weak and beats every fixed start in simulation. Starting moved content from its sources' posteriors, matched in their moments, does not pay until they are corrected for moved content being colder than its source, so moved content starts from the empirical prior too.
- **How the decision reads the posterior matters little.** Deciding by the expected gain (a) or by the probability of a gain (b) differs from the posterior mean by about a point in simulation, except that (b) costs a few points more on pages of a few large chunks. Rule (c), which guards against every loss a page could still suffer, is too cautious; (c'), which weighs each by its predictive probability, is the posterior mean in effect, since the predictive trusts the very posterior whose dips make cleaning early.
- **In simulation, the cure model's posterior, started from the empirical prior and decided by (a), costs 10.9 % more than cleaning at the true index where the static-share draft's fit costs 24.3 %, with chunks losing their bytes in parts; with chunks dying whole, 37.9 % against 45.0 %.** It sees a static share almost exactly once chunks lose their bytes in parts, since a chunk's first loss shows that it drains. The single-rate posterior costs a point or three less than its draft.

These are simulations, costed by the drafts' own model; nothing has run on kladde-bench.

58.2 The decision under a posterior

58.2.1 The gain of cleaning now

Cleaning a page now rather than one epoch later gains $\gamma = \kappa [(1 - x) - a \varphi(r/\kappa)]$ per epoch, where φ solves $\varphi - \ln(1 + \varphi) = y$ for $y = r/\kappa$. Waiting one more epoch costs $\kappa(1 - x)$, the room the page's survivors would not fill, and saves what the bytes that die meanwhile would have cost: ra of them, each of which would have been copied now and cost V afterwards. At its own threshold x^* , content of rate r balances the two, so $r(1 + u_0 V) = \kappa g(x^*)$, and since $h = g - \ln(1 + g)$, $g(x^*)$ is $\varphi(r/\kappa)$. γ/κ is measured in pages: the room a cleaning frees, less the draining share weighted by what waiting would still save on it. The page is ripe once $\gamma \geq 0$, which is the drafts' rule: $a \varphi(r/\kappa) \leq 1 - x$ exactly when $h(z) \geq r/\kappa$, and with one rate per page, $a = x$ and $z = x$.

φ rises from 0, like $\sqrt{2y}$ for small y and like $y + \ln(1 + y)$ for large y , and it is concave.

$y = r/\kappa$	0.0001	0.01	1	10	100
$\varphi(y)$	0.014	0.15	2.1	12.6	104.7

58.2.2 Three ways to decide

With a posterior over the page's parameters θ , which are r with one rate per page and (r, a) with a static share, γ becomes a random variable, and there are three ways to turn it into a decision.

- **(a) The expected gain:** clean once $\mathbb{E}[\gamma] \geq 0$. This is what a decision-maker who is neutral to risk does; over a mixture posterior, it is the weighted sum of the components' expected gains.
- **(b) The probability of a gain:** clean once $P(\gamma \geq 0) \geq 1/2$, that is, once cleaning now is more likely than not to be right. The step function in $P(\gamma \geq 0) = \mathbb{E}[\mathbf{1}\{\gamma \geq 0\}]$ counts how often cleaning is right, not by how much.
- **(c) Ripe whatever it loses next:** clean once the page would be ripe by the posterior mean, or by (a), even after any number of further losses. [Waiting teaches](#) derives it: the other two clean at the first epoch the posterior calls a page ripe, which is early, because a page's next loss can make it unripe again. [In simulation](#), it is too cautious to pay. Its refinement (c') weighs each further loss by its predictive probability, and cleans once cleaning now gains at least what the option to wait is worth.

All of them keep the drafts' ranking by an index, since each is monotone in κ . For every θ , γ/κ rises with κ , since φ rises with r/κ ; so $\mathbb{E}[\gamma]/\kappa$ and $P(\gamma \geq 0)$ rise with κ too, and each criterion holds for every κ above one price κ^* . A page's index is $I = 1/\kappa^*$, the $1/\kappa$ at which it becomes ripe, as in the drafts, so the budget loop, the controller, and the floor stay as they are: a page is ripe once $I \geq 1/\kappa$, and the loop takes the highest index first.

Between losses, the ranking ages each posterior's scale, as the drafts age a rate. A posterior changes every epoch, losses or not, but the ranking can only afford a change when a page changes. So each page's index is computed exactly at each of its losses, and between them its rate is taken to fall by $e^{-\beta}$ per epoch in distribution: the posterior's scale shrinks and its shape stays. Every criterion below then grows the index by the same factor e^β per epoch, on every page alike, which is what the drafts' ordered sets need. Where that differs from the exact update, it takes the posterior for more certain than it is between losses, which [the section on learning](#) shows to be the safe side.

58.3 One rate per page

58.3.1 Losses counted in bytes, weighed in loss events

The observations are the bytes a page loses, weighed by how many independent events they came in: a write supersedes a range, and the bytes of one range are one observation, not many. The model is the single-rate draft's: a page's live bytes die at a rate r , a fraction r of them per epoch, whatever their age. But they do not die one at a time: a write supersedes a range, which on a data page is often part of a Ref's payload, the rest of which lives on, and on a leaf is a whole statement. So a page's losses are a *compound* Poisson process: events arriving at a rate proportional to r and to the live bytes, each removing

some s bytes. Over an *exposure* E , the live bytes times the epochs they were watched, the bytes lost L have mean rE and a variance larger than that by the factor $\sigma = \mathbb{E}[s^2]/\mathbb{E}[s]$, the dispersion of the events' sizes. Divided by σ , L and E behave as a Poisson count and its exposure, and the likelihood of r is

$$p(L | r) \propto r^{L/\sigma} e^{-rE/\sigma},$$

exact where the events are of one size, and L/σ counts them, and a match of the first two moments otherwise. Over an epoch in which a page starts with x live and loses L , its exposure is $x - L/2$, counting what died as live for half the epoch; x , L , and σ may be in pages or bytes, as long as all three are in the same unit.

- **Counting every byte as an observation** would make the posterior σ times too sure: 250 times, for statements of 250 bytes that die whole.
- **Counting statements** would miss that a data page's statements lose their payloads in parts and live on, and would count a statement's many partial losses as none.

The fold knows each event's size, since it supersedes one range at a time, so it can keep $\mathbb{E}[s]$ and $\mathbb{E}[s^2]$ as running averages over the file, one pair for data pages and one for leaves. On a leaf, whose statements die whole, an event is a statement's death, and σ is the mean size of the statements that die, weighted by their size. The dispersion sets only how sure the posterior is: the posterior mean below is a ratio of bytes lost to bytes exposed, the same in any unit.

58.3.2 The exact posterior is a Gamma distribution

A Gamma prior on the rate gives a Gamma posterior, whose two parameters count the page's loss events and its exposure, in events. With a prior $r \sim \text{Gamma}(A_0, B_0)$, of density proportional to $r^{A_0-1} e^{-B_0 r}$, the posterior after losing L over an exposure E is

$$r | \text{losses} \sim \text{Gamma}(A_0 + L/\sigma, B_0 + E/\sigma),$$

with mean A/B and a relative spread of $1/\sqrt{A}$, where A and B are the posterior's two parameters. A is the number of loss events the estimate rests on, the prior's included: the prior acts as A_0 events over an exposure of B_0 , so a starting estimate r_0 worth ν epochs of the page's x_0 would be $B_0 = \nu x_0/\sigma$ and $A_0 = r_0 B_0$, though [below](#) the weight is learned rather than set.

58.3.3 A rate that drifts keeps the posterior exact

A prior under which the rate drifts by random factors, of mean one, turns the update into a discounted one, and keeps it exact: $A \leftarrow \delta A + L/\sigma$ and $B \leftarrow \delta B + (x - L/2)/\sigma$ **each epoch, with $\delta = e^{-\beta}$.** The drafts forget old losses because a page's rate changes as its content does. The Bayesian way to say so is a prior over the rate's path, and one prior makes the forgetting exact: between epochs, $r_t = r_{t-1} \eta_t / \delta$, with η_t drawn from a $\text{Beta}(\delta A, (1 - \delta)A)$ distribution independent of r_{t-1} , where A is the posterior's shape before the step: the gamma-beta model of Smith and Miller, *A non-Gaussian state space model and application to prediction of records* (J. R. Stat. Soc. B, 1986), known in forecasting as the Poisson-gamma steady model with a discount factor. A $\text{Gamma}(A, B)$ variable times an independent $\text{Beta}(\delta A, (1 - \delta)A)$ one is $\text{Gamma}(\delta A, B)$, so the rate's prior for the next epoch is $\text{Gamma}(\delta A, \delta B)$: the same mean, and a variance larger by $1/\delta$. The epoch's losses then update it as above. The drafts' forgetting rate β becomes the prior's drift, its third parameter beside A_0 and B_0 .

58.3.4 The single-rate draft is this posterior's mean

Once a page's exposure has settled, the posterior mean follows exactly the single-rate draft's update. With x live, B settles at $x/(\sigma(1 - \delta))$, and then $(\delta A + L/\sigma)/B = \delta A/B + (1 - \delta)L/x$, which is the draft's lose. The posterior adds two things.

- **Its mean is right while the exposure is still growing.** Until then, the mean is the ratio of the discounted losses to the discounted exposure, as Adam's bias correction would make it, but with the prior's weight, A_0 events, [learned from evidence](#); the draft's start weighs about $1/\beta$ epochs, fixed.

- **Its shape counts the loss events the estimate rests on.** At the settled exposure, $A \approx rx/(\sigma\beta)$. A page at fill 0.5 draining at 0.01 per epoch, with $\beta = 0.1$, rests on 3.2 events if it loses 64 bytes at a time, and its rate is uncertain by 56 %; if it loses whole statements of 250 bytes, it rests on 0.8 of one.

58.3.5 What a page keeps

A page keeps A , B , and the epoch it last lost bytes, all updated in closed form: one number more than the single-rate draft. Over d epochs without a loss, with x live, $A \leftarrow \delta^d A$ and $B \leftarrow \delta^d B + x(1 - \delta^d)/(\sigma(1 - \delta))$.

```
/// Per page, beside kind, epoch and coverage: 12 bytes.
struct Drain {
    a: f32, // the posterior's shape: loss events, discounted, with the prior's
    b: f32, // its rate: exposure in events, discounted, with the prior's
    at: u32, // the epoch of the page's last natural loss, from the session's base
}

impl Drain {
    /// A flush's fold superseded `lost` of the page's `live` bytes; `sigma` is
    /// the dispersion of loss events on pages of its kind.
    fn lose(&mut self, lost: u32, live: u32, sigma: f32, now: u32) {
        let (lost, live) = (lost as f32 / sigma, live as f32 / sigma);
        // The epochs since the last loss, without one:
        let (delta, d) = ((-BETA).exp(), (now - self.at - 1) as f32);
        let decay = delta.powf(d);
        self.a *= decay;
        self.b = self.b * decay + live * (1.0 - decay) / (1.0 - delta);
        // This epoch:
        self.a = delta * self.a + lost;
        self.b = delta * self.b + live - 0.5 * lost;
        self.at = now;
    }
}
```

The dispersion drifts as the application's writes change, and a page's sums mix the values it had when they were added, which matters little since it moves slowly.

58.3.6 Where a page's prior comes from

A page's prior is learned rather than tuned: fresh content starts from the file's empirical prior, and moved content could start from its sources' posteriors, matched in their first two moments, though in simulation the empirical prior serves it better. The drafts start a page at a rate with a fixed weight; here the weight, too, comes from evidence: how alike the file's fresh pages have turned out, and how much its sources knew. Either way, the prior is a Gamma distribution, which is fixed by its mean μ and variance v : $A_0 = \mu^2/v$ and $B_0 = \mu/v$.

Fresh content starts from the file's empirical prior: the mean and the spread of the rates at which fresh pages have drained in their first epochs. The drafts start fresh content at "a running estimate of what pages lose in the epoch after they are written"; this estimates the spread as well. Over the pages the file has written, let page i lose k_i events over an exposure of E_i in its first H epochs, with H the posterior's memory, $1/(1 - \delta)$. Were all of them to drain at one rate, k_i would scatter about μE_i as Poisson counts do; the scatter beyond that is the spread of their rates, and the method of moments for a Gamma mixture of Poisson rates gives

$$\mu = \frac{\sum_i k_i}{\sum_i E_i}, \quad v = \frac{Q - (n-1)\mu}{\sum_i E_i - \sum_i E_i^2 / \sum_i E_i}, \quad Q = \sum_i \frac{(k_i - \mu E_i)^2}{E_i},$$

over the last n such pages, floored at a small fraction of μ^2 . The prior's weight, $A_0 = \mu^2/v$ events, is then how alike the file's fresh pages are: a file whose fresh pages all drain alike starts each one sure of its rate,

and one whose fresh pages differ starts each one unsure, so that its own losses soon decide. The fold keeps the five sums $\sum k_i$, $\sum E_i$, $\sum E_i^2$, $\sum k_i^2/E_i$, and n over the file, discounted like the rest, and adds a page to them once it is H epochs old.

Moved content starts from its sources' posteriors: the Gamma distribution with the mean and variance of the mixture they make. A page whose content comes from sources with byte shares w_i and posteriors of mean $m_i = A_i/B_i$ and variance $v_i = A_i/B_i^2$ starts with

$$\mu = \sum_i w_i m_i, \quad v = \sum_i w_i (v_i + m_i^2) - \mu^2,$$

the moments of the rate of a byte drawn from the mix, and fresh content in the mix counts as one more source, with the empirical prior. The variance includes the spread between the sources as well as each one's own: a page mixed from sources that drain alike starts as sure as they were, and one mixed from sources that drain differently starts unsure, as it should, since the single rate it will be estimated to drain at is not yet known. With a static share, the rate's prior comes from the draining content of the sources alone, and the draining fraction's prior from their draining shares, as before.

But a source sure of the wrong rate hands its certainty on, and moved content's source usually is: the survivors of a cleaned page drain slower than the page did. [In simulation](#), matching the sources' posteriors costs more than starting from the empirical prior, which gains even on pages whose start is right; so until the sources' rates are corrected for what their content's survival says about it, moved content, too, starts from the empirical prior.

A page the consolidator state cannot vouch for starts from the seed, as a past it was watched through. The seed at open has the drafts' rate, and weighs what the page's past would have, had the page been watched through it losing bytes at that rate: B is the discounted exposure of the bytes it would have held, over σ , and $A = \hat{r} B$.

58.3.7 The floor

The floor becomes a rate that no content falls below. Adding $R_{\min}$ times each epoch's exposure to its loss events asserts a background rate $R_{\min}$ for all content, and keeps the posterior mean at or above it; it changes every draining page's rate by $R_{\min}$, which is negligible, and it caps a frozen page's index as the drafts' floor does.

58.3.8 Deciding on one rate

With $r = G/B$ and $G \sim \text{Gamma}(A, 1)$, every criterion reduces to the drafts' index with an effective rate that depends on the posterior's shape A .

- **The posterior mean**, for comparison: $I_0 = h(x) B/A$.
- **(a)** $\mathbb{E}[\gamma]/\kappa = (1-x) - x \Phi_A(B\kappa)$, with $\Phi_A(m) = \mathbb{E}[\varphi(G/m)]$. The page is ripe once $\Phi_A(B\kappa) \leq g(x)$, so $I_a = B/m^*$ with $\Phi_A(m^*) = g(x)$. Φ has no closed form, but it is a smooth function of two variables, so a table of m^* over A and $g(x)$, computed once, gives the index in two lookups.
- **(b)** $P(\gamma \geq 0) = P(G \leq B\kappa h(x))$, the regularized incomplete gamma function $P(A, B\kappa h(x))$, which is $1/2$ where $B\kappa h(x)$ is the median of G . So $I_b = h(x) B/\text{med}(A)$: the drafts' index at the posterior's median rate, with $\text{med}(A) \approx A - 1/3$ for $A \geq 1$.

Both make an uncertain page riper than its posterior mean does, and the less the page has lost, the riper. φ is concave, so $\Phi_A(m) \leq \varphi(A/m)$ and $I_a \geq I_0$; and the median of a Gamma distribution lies below its mean, so $I_b \geq I_0$. The factor I/I_0 , for a page at fill 0.5:

A , the loss events the posterior rests on	0.5	1	2	5	20
(a), expected gain	1.29	1.15	1.08	1.03	1.01
(b), probability of a gain	2.20	1.44	1.19	1.07	1.02

For (a), the factor is larger at higher fills, 1.23 at $A = 1$ and fill 0.8, and smaller at lower ones; for (b), it does not depend on the fill.

58.3.9 Waiting teaches, and the drafts' rule does not see it

The drafts clean a page at the first epoch at which one more epoch of waiting does not pay; that is optimal only if a ripe page stays ripe, and under an estimated rate it does not: the page's next loss can make it unripe again. The static-share draft says so: "once ripe, a page stays ripe, since a and x only fall as it drains; for a stopping problem of that kind, stopping at the first epoch at which one more epoch of waiting does not pay is optimal." With the rate known, that holds. With the rate estimated, a loss does two things: it lowers the fill, which ripens the page, and it raises the estimated rate, which unripens it.

At the drafts' constants, the second wins at every fill above 1/11. Take the posterior mean at a settled exposure, $B \approx x/(\sigma\beta)$, and a loss event of σ . It raises A by one, and so the estimated rate by about $\sigma\beta/x$; and it lowers the fill by σ , which raises $h(x)$ by $\sigma g(x)/x$, since $h'(x) = -(1-x)/x^2$. A page that has just become ripe, $h(x) = \hat{r}/\kappa$, is still ripe after the loss only if $\sigma g(x)/x \geq \sigma\beta/(x\kappa)$, that is, if $g(x) \geq \beta/\kappa$, whatever the size of the event. At $\beta = 0.1$ and $\kappa = 0.01$, that asks for $g(x) \geq 10$, a fill of at most 1/11; at the price of 0.025 that the single-rate branch's controller settled on under skewed overwrites, a fill of at most 0.2. Above those fills, a page that has just become ripe is unripe again after its next loss, until the estimate has fallen again.

So every rule that cleans at the first ripe epoch cleans early, and the more so the fewer losses its estimate rests on. It cleans a page at the first dip of its estimate, not at the epoch at which the page's content has drained enough. That holds for the drafts' point estimates as much as for (a) and (b), which only add to it by making uncertain pages riper still, and it is part of what the static-share draft's simulation saw on pages that drain slowly.

(c) Clean a page only once no loss it could still suffer would make it unripe again. The index for (c) is the least of the indexes the page would have after j further loss events, for every j :

$$I_c = \min_{j \geq 0} I(A + j, x - j\sigma),$$

with I any of the indexes above. Losses at once are the case to guard against: losses spread over time are no worse, since the estimate falls between them, and epochs without a loss only ripen a page further. So once a page's (c) index has reached $1/\kappa$, no observation it could still make would reverse the decision, which restores the drafts' argument, at the price of cleaning later than the best rule would when the losses that would unripen the page are unlikely. The minimum lies a few losses out: for a page at fill 0.5 that loses a thirtieth of a page at a time, with a posterior resting on $A = 2$, it lies at $j = 3$, 24 % below the page's present index by the posterior mean. Rule (c) needs no parameter of its own, and costs a handful of evaluations of the underlying index per loss. But it guards against losses that a page is unlikely to suffer as firmly as against likely ones, and [in simulation](#) that makes it wait too long on every page but those that drain slowly as one share.

(c') Clean a page once cleaning now gains at least what the option to wait is worth. This weighs each further loss by its probability, which (c) did not. Over the posterior's memory, $H = 1/(1 - \delta)$ epochs, a page's loss events have a negative-binomial predictive distribution, the Poisson count averaged over the Gamma posterior:

$$J \sim \text{NegBin}\left(A, \frac{B}{B + aH/\sigma}\right),$$

with $a = x$ for one rate per page. After J events, the posterior is discounted by δ^H , with the events and the H epochs' exposure added, and the page has lost $J\sigma$. If that leaves it unripe, cleaning now forgoes the gain $-\gamma_J$ per epoch that waiting would have kept, so the option to wait is worth $\mathbb{E}_J[\max(0, -\gamma_J)]$ per epoch, and (c') cleans once

$$\gamma \geq \mathbb{E}_J[\max(0, -\gamma_J)],$$

both at the posterior mean. Both sides are monotone in κ , so the index is again the $1/\kappa$ at which they are equal, found by bisection over a sum of some tens of terms. This is the *knowledge gradient*, the one-step look-ahead of Bayesian optimal learning, with a step as long as the posterior's memory.

(c') is the posterior mean in all but name: the option to wait is worth anything only where the posterior rests on less than one loss event. At fills of 0.3 to 0.8, with loss events of 180 bytes, its index equals the posterior mean's for a posterior resting on 2 events or more, with a horizon of 1, 3, or H epochs alike, and falls 4 to 10 % below it for one resting on half an event. The predictive comes from the same posterior whose dip makes the drafts' rule clean early, so it expects no more losses than that posterior does, and over any horizon, the losses it does expect ripen the page further, by draining it, while the estimate relaxes. What (c) guarded against, the dip itself, is not in the predictive at all: it shows that the drift the prior assumes is faster than a page's rate really changes, and the remedy for that lies in the prior's drift, not in the decision rule.

58.4 A draining share over a static one

58.4.1 The cure model, per chunk

Each chunk a page is written with drains with some probability π and is static otherwise; a draining chunk loses its bytes at the rate r , in loss events, and a static one never loses any. A chunk is what the page holds of one statement: on a data page, a Ref's payload, and on a leaf, the statement itself. The class is the chunk's rather than each byte's because temperature belongs to allocations, so a chunk's bytes share one fate. In survival analysis, this is a *mixture cure model*: a population of which a fraction never experiences the event.

A chunk that has lost bytes is known to drain, and on a data page it usually lives on. A write that supersedes part of a Ref's payload shows that the chunk drains, and leaves the rest of it live; on a leaf, a statement's first loss is its death. So the draining share has a known part and an unknown one: the live bytes of the *touched* chunks, x_t , drain for certain, and of the n *untouched* chunks, some number j drain, so that $a = x_t + j\bar{c}$, with $\bar{c}$ the untouched chunks' mean size.

58.4.2 The exact posterior is a mixture over the untouched chunks that drain

With a Gamma prior on r and a Beta prior on π , the posterior is a mixture of $n + 1$ Gamma-Beta products, one for each number j of untouched chunks that drain. Take a page written T epochs ago with N chunks, of which K have been touched, and $n = N - K$ not. The touched chunks drain, and their losses, L in all over an exposure E_t of their bytes since the page's write, bear on the rate as in the single-rate posterior. Each untouched chunk either drains and has lost nothing in T epochs of it, which for c bytes has probability $e^{-rcT/\sigma}$, or is static:

$$\mathcal{L}(r, \pi) \propto \pi^K r^{L/\sigma} e^{-rE_t/\sigma} (1 - \pi + \pi e^{-r\bar{c}T/\sigma})^n = \pi^K r^{L/\sigma} e^{-rE_t/\sigma} \sum_{j=0}^n \binom{n}{j} \pi^j e^{-rj\bar{c}T/\sigma} (1 - \pi)^{n-j},$$

taking the untouched chunks at their mean size $\bar{c}$, which the binomial expansion needs; exactly, the sum runs over the subsets of untouched chunks that drain, each with its own size. Its term j is the case that exactly j of the untouched chunks drain, and each term is a Gamma density in r times a Beta density in π , so with the priors $r \sim \text{Gamma}(A_0, B_0)$ and $\pi \sim \text{Beta}(p_0, q_0)$, the posterior is the mixture

$$p(r, \pi \mid \text{losses}) = \sum_{j=0}^n w_j \text{Gamma}(r; A, B_j) \text{Beta}(\pi; p_0 + K + j, q_0 + n - j), \quad w_j \propto \binom{n}{j} \frac{B(p_0 + K + j, q_0 + n - j)}{B_j^A},$$

with $A = A_0 + L/\sigma$, $B_j = B_0 + (E_t + j\bar{c}T)/\sigma$, and B the Beta function. The weight w_j is the posterior probability that exactly j of the untouched chunks drain, and given j , the rate has the single-rate posterior of a page whose j draining untouched chunks have added their exposure to that of the touched ones.

- **The Beta factor** prefers splits like the page's past: $K + j$ draining chunks out of N , against the prior's p_0 and q_0 .
- **The Gamma factor** B_j^{-A} charges each draining untouched chunk for having lost nothing: the more of them drain, the lower the rate must be to explain their survival, and the less the losses that did happen fit.

- $j = n$ is the single-rate posterior: every chunk drains, and B_n is the page's whole exposure.

58.4.3 The prior replaces the test

The static-share draft makes a static share earn c statements' worth of log-likelihood; here the prior on π does that, with a weight of its own. A page starts from the draining fraction π_0 of what it holds, 1 for fresh content, which the draft starts with no static share; the prior is $\text{Beta}(\nu_\pi \pi_0 + \frac{1}{2}, \nu_\pi (1 - \pi_0) + \frac{1}{2})$, worth ν_π chunks, with half a chunk of each kind so that neither is ruled out. For fresh content, the weights of splits with static chunks start small, and grow only as the untouched chunks outlive what draining content would. Unlike the test, the prior does not decide between one share and two: the posterior keeps both, weighted, and the decision sees the mixture.

58.4.4 A drifting rate

Discounting all of the evidence, the class count with the rest, keeps the mixture's form; it is an approximation, since the drift has no exact update here. The single-rate posterior's discounting carries over to everything that bears on the rate: L , the touched chunks' exposure E_t , and T , which becomes the discounted age $T_\delta = (1 - \delta^T)/(1 - \delta)$, the exposure of a byte live throughout; the prior's A_0 and B_0 are discounted with them. With $j = n$, the discounted B_n is again the single-rate draft's settled exposure, so the two models agree where every chunk drains. A touched chunk's live bytes stay in the draining share for good, and the rate says how fast they drain now: if the chunk has since gone cold, the rate falls.

The count of touched chunks is discounted too, although a chunk's class does not drift, so that it weighs against as much of the untouched chunks' survival as the posterior remembers. The untouched chunks' exposure $j \bar{c} T_\delta$ is discounted, so a chunk that has outlived its page's draining content by many epochs counts for only the last $1/\beta$ or so of them. Kept whole, the count of touched chunks would go on weighing, undiminished, against that shortened survival: a page whose draining chunks were touched long ago would go on counting them as evidence that its untouched chunks drain too, when their long survival says they do not. And "draining" lumps together content that drains at different rates: on the page of [the static-share draft's example](#), the fast share's early losses would count, for good, as evidence that the untouched chunks drain, although they drain at a twenty-fifth of that rate if at all. Discounted, the class evidence fades with the rate evidence it belongs to, which [in simulation](#) costs less.

58.4.5 What a page keeps, and what a loss costs

A page keeps its discounted loss events, the touched chunks' discounted exposure, the discounted count of touched chunks, the epoch of its last loss, the count and bytes of its untouched chunks, and the prior's two class counts: 22 bytes; and each statement needs one bit, set at its first loss. The prior on the rate enters the discounted sums as pseudo-observations, as in [the single-rate posterior](#); the prior on π is the two class counts, p_0 and q_0 , a byte each in halves of a chunk, which a page keeps from its write. T_δ follows from the page's epoch.

The fold learns a chunk's first loss from a bit in its statement's record. [The statement slab's record](#) counts a statement's pins in 32 bits, which leaves room for one more. When the fold supersedes part of a Ref's payload whose bit is clear, it sets the bit, moves the chunk's bytes, the extent of its fragment before the split, from the page's untouched counts to its touched ones, and adds their exposure since the page's write, $c T_\delta$, to the touched exposure. At open, a Ref whose live fragments no longer cover its whole payload is touched; on a leaf, a statement's first loss is its death, and no bit is needed.

Each loss recomputes the $n + 1$ weights, $O(n)$ work in the untouched chunks, some 30 terms on a data page at most and some 300 on a leaf; the heaviest few carry almost all the weight, and the rest can be dropped.

58.4.6 Deciding on a mixture

Every criterion is a sum over the mixture, with one Gamma posterior per split. With $a_j = x_t + j \bar{c}$ and $z_j = a_j / (1 - x + a_j)$:

- (a) $\mathbb{E}[\gamma]/\kappa = (1 - x) - \sum_j w_j a_j \Phi_A(B_j \kappa)$. All components share the shape A , so one table of Φ_A serves every j .

- (b) $P(\gamma \geq 0) = \sum_j w_j P(A, B_j \kappa h(z_j))$, where a term with nothing draining, $a_j = 0$, counts as ripe at any price.
- (c') as with one rate, at the posterior means of the draining share a and its rate, with the predictive of the losses that a will suffer.

The index is the $1/\kappa$ at which the criterion holds with equality, found by bisection, each step $O(n)$. Aging the rate's scale between losses multiplies every B_j by the same factor, which leaves the weights w_j as they are and scales the index, so the ranking ages uniformly here too.

58.4.7 How it relates to the draft's fit

The draft's fit reads the page's losses alone; the cure model also reads which chunks they came from, and which they did not. The fit asks how fast a page's losses fall, and infers the draining share from the losses' current rate; the untouched content enters only as a cap on it. The cure model knows the touched chunks drain without waiting for them to die, and asks, of every untouched chunk, how likely it is to drain, given that it has lost nothing in the epochs the posterior remembers, and given how the page's other chunks behaved. So a page whose losses keep coming from a few chunks has evidence of a static share in all of its other chunks, even when its losses are few, which is the case [the evaluation](#) found the fit unable to see.

58.5 Checking in simulation

In simulation, the cure model's posterior costs half of what the static-share draft's fit does where chunks lose their bytes in parts, as a data page's Refs do, and 3 to 30 % less where they die whole; the single-rate posterior costs a little less than its draft; the file's empirical prior beats every fixed start; and how (a), (b), and (c') read a posterior matters less than the posterior itself. `tools/simulate-ripeness.py compare --set bayes` runs the eight scenarios of [the static-share draft's simulation](#), with its measure: the excess cost of cleaning each page by an estimate, over cleaning it once its true index reaches $1/\kappa$. Its pages hold chunks, which either die whole, as statements on a leaf do, or lose their bytes in pieces, each piece dying at its chunk's rate, as a Ref's payload loses the parts that writes supersede; the dispersion σ is that of the pieces, as the fold would measure it, and the tested fit counts its test in loss events too. It runs 400 pages a scenario rather than 800, since the mixture is slower, with $\beta = 0.1$, $\nu_\pi = 10$ chunks on the draining fraction, (c) for one rate per page only, over the posterior mean, and the drafts' floor as a cap on the index rather than as a background rate. The first two tables start each page as the drafts do, from its scenario's rate, at a weight of $\nu = 3$ epochs; [the third](#) compares that with the learned priors. The summed excess cost of the eight scenarios:

chunks of	losses	the single-rate draft	posterior mean	(a)	(b)	(c)	(c')
16 to 64 bytes	whole	27.4 %	27.0 %	26.3 %	26.0 %	46.8 %	27.5 %
32 to 256 bytes	whole	46.3 %	44.3 %	44.1 %	44.8 %	44.6 %	44.3 %
32 to 256 bytes	pieces of 32 bytes	27.2 %	26.7 %	26.0 %	25.3 %	58.1 %	27.3 %
256 to 1024 bytes	whole	112.8 %	111.8 %	113.7 %	117.3 %	111.6 %	not run
256 to 1024 bytes	pieces of 64 bytes	30.6 %	29.8 %	29.3 %	29.2 %	40.4 %	29.9 %

chunks of	losses	the static-share draft's tested fit	cure model, posterior means	(a)	(b)	(c')
16 to 64 bytes	whole	27.4 %	21.5 %	19.6 %	19.3 %	21.6 %

chunks of	losses	the static-share draft's tested fit	cure model, posterior means	(a)	(b)	(c')
32 to 256 bytes	whole	45.0 %	40.4 %	40.1 %	40.0 %	40.3 %
32 to 256 bytes	pieces of 32 bytes	24.3 %	12.7 %	11.8 %	11.5 %	12.8 %
256 to 1024 bytes	whole	109.3 %	106.0 %	106.2 %	109.6 %	not run
256 to 1024 bytes	pieces of 64 bytes	30.3 %	15.7 %	14.5 %	15.1 %	15.6 %

The pages are the same for every estimator, but a difference of a point or so in a sum is within what a different draw of pages moves.

- **Where chunks lose their bytes in parts, the cure model sees a static share almost exactly.** With chunks of 32 to 256 bytes losing 32 at a time, it costs 0.1 % on the page with a static share, against the tested fit's 6.4 % and the single-rate draft's 9.4 %, and 1.3 to 2.1 % on the page of the draft's example, against 7.1 %. A chunk's first loss shows that it drains, long before it dies, so the chunks that never lose anything stand out as static within a few epochs.
- **Where chunks die whole, the cure model still gains, 40 % against the tested fit's 45 %:** some 12 points on the pages with a static share, from the untouched chunks that outlive the draining content, less some 5 on the pages that drain as one share, where the fit's test keeps out static shares that are not there.
- **With one rate per page, the posterior costs about what the draft does,** a point or two less, since its mean is the draft's estimate; what it adds, the certainty, changes little in the decisions.
- **Partial losses make every estimate better,** since they are more events: the single-rate draft's excess cost falls from 46 % to 27 %, and the cure model's from 40 % to 12 %.
- **(a) and (b) differ little from the posterior mean, and from each other,** except on pages of a few large chunks that die whole, where (b) costs 3 to 4 points more than (a). They make uncertain pages riper, as [derived above](#), which gains a little on pages with a static share and loses a little on pages that drain slowly as one share.
- **(c) is too cautious wherever losses come in many small events:** with chunks losing 32 bytes at a time, it costs 58 % against the posterior mean's 27 %, since it guards against as many further losses as the page has pieces. It gains only on pages that drain slowly as one share, 0.7 % against 1.1 % on the page draining at 0.01.
- **(c') costs what the posterior mean does,** up to half a point more, as [derived above](#): the predictive it weighs further losses by comes from the posterior it would have to distrust.
- **A lighter prior pays:** with $\nu = 10$, as heavy as the single-rate draft's start, the single-rate posterior costs 30.5 % against 26.7 % with chunks losing 32 bytes at a time, most of it on the page whose start is wrong, where a heavy prior holds on to the wrong rate while the page shrinks; the cure model costs 14.3 to 15.4 % against 11.5 to 12.7 %.
- **Discounting the class count pays:** kept whole, it costs the cure model 14.1 to 17.3 % against 11.5 to 12.7 %, with chunks losing 32 bytes at a time.

58.5.1 Where a page's prior comes from, in simulation

The file's empirical prior beats every fixed start, and matching the sources' posteriors does not, since a source that is sure of the wrong rate hands its certainty on. The empirical prior is learned, as [derived above](#), from 400 fresh pages of each scenario but the seeded one, drawn apart from the pages it then starts, over their first 10 epochs; every page but the seeded ones starts from it, whatever its scenario's rate. The moment-matched prior takes each scenario's rates as its sources', each source a page at fill 0.5 whose posterior has settled, and so rests on as many events as such a page would have. With chunks of 32 to 256 bytes:

the rate's prior	losses	posterior mean	(a)	(c')	cure model, posterior means	(a)	(c')
the scenario's rate, $\nu = 3$	whole	44.3 %	44.1 %	44.3 %	40.4 %	40.1 %	40.3 %
moment-matched to the sources	whole	46.4 %	46.3 %	46.4 %	42.2 %	41.6 %	42.1 %
the file's empirical prior	whole	43.3 %	43.1 %	43.3 %	39.1 %	37.9 %	39.0 %
the scenario's rate, $\nu = 3$	pieces of 32 bytes	26.7 %	26.0 %	27.3 %	12.7 %	11.8 %	12.8 %
moment-matched to the sources	pieces of 32 bytes	26.4 %	25.6 %	27.0 %	12.8 %	12.0 %	12.9 %
the file's empirical prior	pieces of 32 bytes	25.2 %	24.5 %	25.8 %	11.6 %	10.9 %	11.8 %

- **The empirical prior is weak, and that is why it wins.** It comes out at a mean rate of 0.045 per epoch, resting on 0.8 of an event, since the scenarios' fresh pages drain at rates from 0.002 to 0.5: each page's own losses soon decide. It gains most on the page whose start is wrong, 0.5 % against 1.2 % with whole chunks, and costs nothing on the pages whose start is right, although it ignores that start.
- **The moment-matched prior gains a little on pages mixed from sources that drain differently,** a few tenths of a point on the two mixed pages, since it starts them unsure, but loses on the page whose source is sure of the wrong rate, 3.8 % against 1.2 % with whole chunks. Moved content, being colder than its source, is that case more often than not, as the [open questions](#) note, so this draft starts moved content from the empirical prior too, until the sources' rates are corrected for what their content's survival says.

58.6 What would change

In either draft, the estimate becomes a posterior, the index is computed from it by (a), and every constant the estimate had becomes a parameter of the prior.

- **"Estimating how fast a page still drains"** would keep a posterior: with one rate per page, the Gamma posterior's two parameters in place of ρ ; with a static share, the discounted loss events and exposure, the touched and untouched chunks' counts, and the class counts in place of the fit's sums, and the mixture in place of the fit and its test.
- **The page table** would keep a Drain of 12 bytes with one rate per page, against the single-rate draft's 8, or 22 with a static share, against the static-share draft's 18; with a static share, each statement's record would also give a bit to mark its first loss.
- **The fold** would keep two running averages per page kind, of the sizes of the ranges it supersedes and of their squares, for the dispersion σ , and five running sums over the pages it has watched through their first epochs, for the empirical prior.
- **A new page** would start from the file's empirical prior, where the drafts start it from a byte-weighted mean rate with a fixed weight; its moved content would start from its sources' posteriors, moment-matched, only once those are corrected for moved content being colder than its source.
- **Ranking** would compute a page's index by (a) at each of its losses, from a table of Φ computed once, and age it between losses as the drafts do; (b) needs no table, and cost a few points more in simulation only on pages of a few large chunks that die whole.
- **Bytes that consolidation moves out of a page** would lower its live bytes without counting as losses; the posterior's evidence on the rate stays, and with a static share, a chunk moved out leaves the page's touched or untouched counts, whichever held it.
- **The constants** would all become the prior's:

the drafts' constant	becomes
the forgetting rate β	the drift of the prior over the rate's path, $\delta = e^{-\beta}$
the starting estimate, and its weight of $1/\beta$ epochs or n_0	learned: for fresh content, the file's empirical prior; for moved content, its sources' posteriors
the static-share test's c	the weight ν_π of the prior on the draining fraction, in chunks
the floor $R_{\min}$	a background rate that the model asserts for all content

58.7 Open questions

- **Moved content is colder than its source.** The survivors of a cleaned page outlived what died around them, so they drain slower than the page they came from, whose posterior they bring; the moment-matched prior overstates their rate, if by less as the source's own evidence is weaker. Discounting a source's rate by what its survival says about it would correct that.
- **A starting prior's weight is exposure, fixed in byte-epochs.** On a page that drains fast, and shrinks, a strong start outweighs the page's own losses for longer than the drafts' averages of fractions let it; a learned prior is strong only where the file's pages, or a page's sources, agree.
- **Chunks of unequal size.** The mixture takes the untouched chunks at their mean size; the exact posterior weighs each subset of them by its own sizes, which a page of a few large chunks and many small ones would notice.
- **What a loss event is.** The dispersion treats every superseded range as one event, but one write that supersedes ranges on several pages, or several ranges of one page, is arguably one event; which grouping calibrates the posterior best is a question for measurement.
- **Pooling evidence across pages, beyond the start.** The empirical prior pools the file's pages once, at a page's start; the pages one flush wrote hold content the application wrote together, and a hierarchical prior, with the rates of one flush's pages drawn around a rate shared by them, would go on lending each page the others' losses.
- **A drift that fits the file.** A page's next loss can make a ripe page unripe, so cleaning at the first ripe epoch cleans early; (c), which guards against every loss the page could still suffer, waits too long, and (c'), which weighs each by its predictive probability, does nothing, since the predictive trusts the posterior's dip. The dip comes from the prior's drift, which forgets a page's losses at β whether its rate changes or not. The drift could be learned from the file as the prior's other parameters are, for instance by how well each page's posterior predicts its next losses, and a slower drift for pages whose losses stay steady would dip less.
- **Moved content's classes.** Chunks moved from a source where they had been touched could arrive known to drain, and the draining fraction's prior could come from the sources' mixtures as the rate's does, rather than from their shares with a fixed weight.
- **Epochs, not time.** The model counts rates per epoch, as the drafts do, and so shares their open question about the clock.

Chapter 59

Sorting survivors by temperature

Status: a proposal, not adopted. It would change where [consolidation](#) puts the survivors of the pages it cleans, whichever policy chooses those pages: data-page scoring and the churn floor on main, or [ripeness](#) and its [Bayesian variant](#).

Sort each survivor into one of three classes by what has happened to it since its page was written, and pack each class onto pages of its own. A survivor is *hot* if its own chunk has lost bytes since it was placed, *warm* if not but its allocation has been written since, and *cold* if its allocation has not been written since. The evidence is the allocation's [last_written](#), which kladde already keeps and carries across sessions, and one bit per statement, set when a Ref loses part of its payload. Hot survivors join the flush's own content, warm and cold ones fill pages of their own class, and within a class they keep key order. Today, a victim's survivors move together, so a page that held cold content next to content that still dies hands both on to the same new page, and [the mixed workload](#) keeps nine tenths of its cold data on pages that mix classes under every policy measured.

59.1 The problem

Evacuation moves a victim's survivors as a unit, and nothing about them decides where they go but the victim they came from. A victim's survivors are taken in key order and placed one after another, into the room of a page the flush writes anyway or into a page [the budget loop](#) opens, and a budgeted page holds the survivors of whole victims, side by side. The ripeness draft's rule that [a budgeted page takes survivors of ripe pages, cold with cold](#), assumes that what survives a ripe page has stopped dying.

On a page that mixed classes, what survives is cool content, which still dies, next to cold content, which never will. On kladde-bench's mixed workload, four allocations of 1 KiB share each page at random, a tenth of them hot, the rest cool or cold alike, and 91 % of the cold data starts on a page that mixes classes. Once a page's hot allocation has died, its survivors are its cool and cold allocations, and cleaning moves them together. So 88 to 90 % of the cold data still shares a page with data that changes at the end of every run, under ripeness, under [its Bayesian variant](#), and without any cleaning at all; only main's frozen pages, as their cool content dies away, bring it down to 84 or 85 %. Which page gets cleaned cannot change that; where each survivor goes can.

59.2 The proposal

59.2.1 Three classes

A survivor of victim page v , written at epoch E_v , belongs to one class, by two pieces of evidence that are nested.

class	the survivor's chunk	its allocation	what it says
hot	has lost bytes since it was placed in v	written since E_v	writes are landing in this very chunk

class	the survivor's chunk	its allocation	what it says
warm	has not	written since E_v	the allocation is in use, and its untouched part dies when the allocation is rewritten, shrunk, or freed
cold	has not	not written since E_v	nothing has happened to it in the page's whole life

The more local evidence ranks hotter. A chunk that has lost bytes since it was placed in v lost them to a write that reached its allocation after E_v , so every hot survivor's allocation has been written since, and the classes are nested. The reverse does not hold: a write to the allocation may have gone to another part of it, a hot tail beside a cold head, and left this chunk alone.

One refinement uses recency, which the chunk's evidence lacks. That a chunk has lost bytes says nothing about when; its allocation's age, now – `last_written`, bounds how recently it can have. A hot survivor whose allocation has not been written for more than T epochs counts as warm, since the writes that touched it have stopped; T is a constant to be chosen, and the posterior's memory $1/\beta$ of [ripeness](#) is a natural first value.

The classes are relative to the victim's age, not to a fixed threshold. "Written since E_v " asks whether anything happened to the allocation while it sat in this page, which is the question placement needs: content that changed less often than that is, at the scale of this page's life, cold.

59.2.2 The evidence

`last_written` exists and costs nothing new. The allocation map keeps it for every id, moves it whenever the application's writes reach the id, and the [consolidator state](#) carries it from one session to the next. It measures content age, so moving a survivor does not reset it, as it does a page's epoch. Comparing it with E_v is one lookup per survivor, which evacuation does anyway to find the survivor's allocation.

Whether a chunk has lost bytes needs one bit per statement, which the slab record can hold. The fragment map no longer says where a chunk ended, and the slab record holds only a statement's page and its pins; the Ref's stated extent lives in the encoding in its table page, which only open decodes. So the bit is kept as it goes, as [the Bayesian draft proposes](#), in the [statement slab's record](#), whose 32 bits of pins leave room for it:

- **It is set** when a natural loss, the fold's or the consolidator state's own rewrite, releases bytes of a Ref's fragments and leaves the statement live. A loss that takes the statement's last fragment kills the statement, and the bit with it.
- **Consolidation's moves do not set it**, since bytes moved out are not bytes lost.
- **A restatement in place carries it.** When a page rewrite, the rotating window, or the header restates a Ref whose bytes stay where they are, the new statement states only the live bytes, and the old extent is gone; so the bit rides on the pending entry, as [heat does](#), and a statement derived from several fragments takes it if any of them had it.
- **A relocation clears it.** A survivor that moves to a new page is a new chunk there, untouched until it loses bytes again; the class it was sorted by has already acted on it, and its allocation's `last_written` still says that it is in use.
- **Open rebuilds it**, since it is not persisted: open decodes every statement, so a Ref whose live fragments cover less than it states is set. Statements restated in an earlier session have lost their old extent and come back clear, which turns some hot survivors into warm ones until they lose bytes again.

59.2.3 Where each class goes

Each class goes where content of its own temperature is.

class	where it goes	why
hot	among the flush's own chunks, which pack places in key order	it is being written now, as the flush's fresh content is
warm	budgeted pages for warm survivors	content of allocations in use, which dies when they are rewritten, shrunk, or freed
cold	budgeted pages for cold survivors, together with description defragmentation's rewrites	LFS's segregation: no hot content to wait for, and the page stays full

The budget loop takes its victims before pack, and hands their hot survivors to it. Today, pack runs first and [pulls victims into the room](#) of the flush's own pages, and the budget loop opens pages for more victims afterwards. Under this proposal, the loop chooses its victims first and takes their survivors, sorting each into its class; the hot ones join the flush's own chunks, and the warm and cold ones fill pages of their own. Victims whose survivors take at most θ of a page are left to free filling, as in [the ripeness draft](#), so that the loop does not open pages for what the room of the flush's own pages takes for nothing. The [one survivor cut at a page boundary](#) that a budgeted page may take is cut within its class.

The loop keeps one open page per class, and counts every page each class needs against the budget. A victim is taken only if all of its survivors fit: its hot ones in the flush's own pages, counted against the budget by their bytes, and its warm and cold ones in the open page of their class or in the next one, within the budget. When the budget is spent or no ripe victim is left, each class's last page is checked:

- a page with at most θ of it empty closes as usual;
- a page with more than θ empty takes the other class's last page's survivors if they fit, since a small share of one class in the other's page costs little, as [the ripeness draft's mixed-page analysis](#) shows for shares below the content's own threshold;
- otherwise it closes short, which costs a page that is soon ripe; the loop prefers, among the ripe victims within reach, those whose survivors fill the open pages, as the current offer prefers victims that fill its page.

Free filling sorts the survivors it takes too. A small victim that rides in the room of a flush's own page sends its hot and warm survivors there, and its cold ones to the cold page if the flush has opened one; if it has not, they ride along as well, as a share of at most θ . [The cursor](#), where ripeness has one, is left as it is: its page is a flush or two old, so the classes say little about its content, and it exists to match the flush's own temperature anyway.

Description defragmentation's rewrites go to the cold page. They are cold by selection, and today share pages with the flush's fresh content, [an open question of the ripeness draft](#) that this answers.

59.2.4 Within a class, key order

Each class is packed in (id, offset) order, as [the flush's chunks are](#). Key order keeps the [reverse index](#) small and lets the cut merge fragments of one allocation that land side by side into one Ref. Sorting survivors by age within a class would mix allocations further for a distinction the classes already draw.

59.2.5 What it costs

- **Memory:** one bit per statement, which the slab record can spare, and one flag on the pending entry.
- **Work:** a lookup and a comparison per survivor, on top of taking it.
- **Pages that close short:** up to two per flush beyond the flush's own last page, one per budgeted class, when the budget runs out with their pages partly filled; the rules above keep them rare, but how rare is a question for measurement.
- **Statements:** an allocation whose chunks fall into different classes is split across pages, which costs a statement and a reverse-index entry per page it lands on. That split is the separation the proposal is after, a hot tail moved away from a cold head; allocations whose chunks share a class stay together.
- **Order of work in the flush:** the budget loop runs before pack, and leaves small victims to free filling, where today it takes what free filling left.

59.2.6 What it gives the estimates

A page written from one class starts with a prior that fits it. Under [ripeness](#), a page's starting rate is the byte-weighted mix of its sources', and under [the Bayesian draft](#), its draining fraction's prior comes from its sources' draining shares. A cold page could start at the floor rate and with its chunks static, and a warm page as draining, which is the [open question about moved content's classes](#) answered from the placement side. Neither estimate needs it, but both would learn less from the page's first losses.

59.2.7 What to expect on kladde-bench

These follow from the workloads' parameters, not from runs.

- **Mixed.** Cold allocations are never written after the fill, so their survivors are cold. Cool ones take about 5 % of the writes, so one is written about once every 80 flushes at 8 MiB and every 600 at 64 MiB: those written while their victim page lived are warm, and those that were not are cold. So the cold pages would hold the cold data and the cool data that happened to be quiet, more of it in larger files, and the share of the cold data on pages that mix classes should fall well below its 88 to 90 %, but not to nothing. Whether that saves writes, by keeping cold pages full and so out of the cleaner's way, is the question the experiment answers.
- **Uniform overwrites.** Every allocation of 16 KiB is written about twice a flush at 8 MiB and every four flushes at 64 MiB, so nearly every survivor is warm or hot, and the proposal changes little; with no locality to find, it finds none.
- **Skewed overwrites.** The cold allocations still take a tenth of the writes, so one is written about every 5 flushes at 8 MiB and every 37 at 64 MiB: their survivors are mostly warm at 8 MiB, rightly, since they still die, and more of them cold at 64 MiB, where they die slowly.
- **Appends.** Every log is appended to every flush or two, so the untouched records of a log are warm, rightly, since they die together when it rotates.

59.3 Alternatives

- **Allocation age alone, two classes.** No new state at all: cold if the allocation has not been written since E_v , warm otherwise. It cannot tell a cold head from a hot tail in one allocation, which large allocations with skewed writes within them would show; the mixed workload's 1 KiB allocations would not. It is the first step to measure, since it needs nothing but the sorting.
- **Chunk evidence alone.** Hot if touched, cold otherwise: it misses that an allocation in use dies as a whole when it is freed or rewritten, as a log does when it rotates, and would pack a log's untouched records with cold content.
- **An absolute age threshold** in place of E_v : cold if the allocation has not been written for T' epochs. It would treat survivors of young and old victims alike, but needs a constant that the victim's own age supplies for free.
- **A finer split**, by allocation age within a class or by a score that combines age and chunk evidence: more classes would separate content more finely, at the price of more open pages per flush, each of which may close short.
- **Inferring chunks from the page instead of keeping a bit.** Two live fragments of one allocation in the victim at the same displacement, the page offset less the allocation offset, were placed together, and the dead bytes between them were that allocation's: the chunk has lost its middle. This needs no state and survives restatements, but it misses a chunk that lost its head or tail, and the page itself does not help further, since data pages record no chunk boundaries. It could complement the bit after open, for statements restated in an earlier session.
- **Hot survivors on a page of their own, or with the warm ones**, which would keep the budget loop after pack as it is today. Hot survivors on ripe pages are likely few, since a ripe page's content has mostly stopped dying, so a page of their own would mostly close short; sharing the warm page costs a little mixing and saves reordering the flush. If measurement shows them rare, this is the simpler choice.
- **Sorting the flush's own content too.** The flush packs its fresh chunks in key order, which mixes allocations the application keeps rewriting with ones it has just created, [the largest source of mixed pages the ripeness draft leaves in place](#). An allocation's `last_written` before this flush's write would tell the two apart in the same way, at the price of key order among the flush's own pages; it is a separate decision from this one.

59.4 What would change

- **Consolidation**: the budget loop would choose its victims before pack, keep an open page per class, and leave small victims to free filling; free filling would sort the survivors it takes; description defragmentation's rewrites would go to the cold page.
- **In-memory state**: the statement slab's record would gain the bit, and the pending entry a flag carried through restatements.
- **Opening a file** would set the bit for every Ref whose live fragments cover less than it states.
- The **ripeness draft** would split "survivors of ripe pages" in its table by class, and its starting estimates could take the class into account.

59.5 Open questions

- *T*, after which a hot survivor whose allocation has gone quiet counts as warm, and whether it is needed at all.
- **How often class pages close short**, and whether the loop's preference for victims that fill them is enough at small budgets.
- **Whether the gain survives the cost**. Cold pages that stay full are cleaned less often, but every allocation split across classes costs statements, and the order of work in the flush changes; the mixed workload, with allocation age alone and then with the bit, measures both.

Chapter 60

Variable-size values

Status: a proposal, not adopted. It would give every type a packed encoding beside its fixed one, describe containers structurally, with pointers that say what they point to, add a kind of value that has only a packed encoding, and add a choice of encoding wherever a value is laid out. Nothing in the file layers below the schema would change, and nothing in memory either.

Separate two questions that a value’s slot answers together today: how many bytes the value’s encoding takes, and how much room the place that holds it reserves. A type’s encoding is *fixed-size* or *variable-size*; a place is *slotted*, holding a type’s fixed encoding, the same size for every value, or *packed*, holding its packed encoding, as many bytes as the value needs. Most types have both encodings: in the packed one, an enum takes only its current variant, an integer wider than a byte and a pointer are LEB128 varints, and a char is UTF-8. A new kind of descriptor, `Small(C, T)`, has only the packed one: a size tag, then a short `C` inline, or the tag 255 and a `T` — for kladde-types’ `SmallPersistableString`, the string’s text inline, or a pointer to it. For pointers to pack, the schema must see them, so the containers that hold them stop being opaque and become structural: a `Pointer(T)` to an allocation that holds a `T`. A slotted value is slotted throughout, up to its pointers; a packed value packs everything inline in it, unless a place inside it declares itself slotted. The choice changes only the file: in memory, a `u16` stays a `u16`, and a `SmallPersistableString` holds its text however it is laid out. The slot stays the default, and files that use no packing keep every byte they have, though types that hold containers get new fingerprints, since the containers’ descriptors become structural.

60.1 The problem

Every type has one fixed inline size, and an enum’s is its discriminant plus its largest variant, so every value of an enum pays for the largest variant. That is [what makes offsets static](#), and with them field access, element access by index, and the reading of a writer’s layout under [evolution](#). Rust pays the same padding in memory, so the in-memory representation loses nothing it would not lose anyway. The file and the journal do lose: a store writes the whole slot, padding included, and the padding then occupies data pages like any other bytes.

Take SVG path data, a sequence of segments of ten kinds, as a typical case:

```
enum PathSegment {
    MoveTo { abs: bool, x: f32, y: f32 },
    LineTo { abs: bool, x: f32, y: f32 },
    HorizontalLineTo { abs: bool, x: f32 },
    VerticalLineTo { abs: bool, y: f32 },
    CurveTo { abs: bool, x1: f32, y1: f32, x2: f32, y2: f32, x: f32, y: f32 },
    SmoothCurveTo { abs: bool, x2: f32, y2: f32, x: f32, y: f32 },
    Quadratic { abs: bool, x1: f32, y1: f32, x: f32, y: f32 },
    SmoothQuadratic { abs: bool, x: f32, y: f32 },
    EllipticalArc { abs: bool, rx: f32, ry: f32, x_axis_rotation: f32,
                    large_arc: bool, sweep: bool, x: f32, y: f32 },
```

```
ClosePath { abs: bool },
}
```

With a one-byte discriminant, its slot is 26 bytes, the size of `CurveTo`. Its variants need 2 bytes (`ClosePath`), 6 (`HorizontalLineTo`, `VerticalLineTo`), 10 (`MoveTo`, `LineTo`, `SmoothQuadratic`), 18 (`SmoothCurveTo`, `Quadratic`), 24 (`EllipticalArc`) and 26 (`CurveTo`). A path that is mostly curves wastes little, but the outline of an icon, `M L L L Z`, takes 130 bytes in slots where its content is 42. An element tree has the same shape one level up: a node that is either an element or a short run of text pays for the element in both cases.

Integers are the same problem in a different form. A count, a length, or an index is stored at the width its largest possible value needs, 4 or 8 bytes, while nearly every value written fits in one. A pointer is one too: it takes four bytes, while the ids of a file's allocations, which kladde [recycles lowest first](#), mostly fit in two.

Short content is the problem in a third form: every `PersistableString` and every `PersistableVec` owns an allocation, however little it holds. In kladde-svg's model of an SVG drawing, strings are the most common source of allocations, nearly all of them ids, followed by the attribute lists of elements and by path data:

drawing	allocations	attribute lists, median size	path data, median size	ids, median length
tiger	1,453	482, 34 bytes	240, 130 bytes	482, 4 bytes
coat of arms	2,007	609, 34 bytes	494, 182 bytes	609, 8 bytes
world map	8,542	2,529, 17 bytes	2,213, 78 bytes	2,528, 8 bytes

Strings, attribute lists and path data together are 83 to 93 % of the allocations, counted from the model with a throwaway program, with each allocation's size in today's fixed encodings. All but two of the 4,390 strings are shorter than 128 bytes, the attribute lists hold one or two attributes, and much of the path data is short as well. On file, such an allocation is cheap already, since content below the [Inline threshold](#) lives in its statement, inside pages a flush writes anyway; what remains is the pointer in the parent and the statement's framing. In memory, it costs an entry in the allocation map, a statement in the slab, and a fragment in the fragment map, as [in-memory state](#) describes them, and every change to it is journaled against an allocation of its own.

Two existing mechanisms reach part of this, and neither reaches the padding between elements.

- A [Zero statement](#) describes a range of zeros without data bytes, and the address table already names "data types with unbalanced variant payloads" as a use for it. But a statement costs on the order of ten bytes of address table, more than the padding between two elements of a vector, so stating padding as Zero would grow the file rather than shrink it.
- A narrower discriminant, which the specification allows (1, 2, 4 or 8 bytes) and kladde-rs picks by the largest discriminant value, saves on the tag and nothing on the padding.

60.2 The proposal

60.2.1 Encodings: fixed and packed

Every type has a packed encoding, which a reader who knows the type's descriptor decodes front to back, learning where the value ends as it goes, and most types also have a fixed encoding, which takes the same number of bytes for every value. They differ by kind:

kind	fixed encoding	packed encoding
u8, i8, bool	one byte	the same byte
u16, u32, u64	fixed width, little-endian	unsigned LEB128: 1–3, 1–5 and 1–10 bytes

kind	fixed encoding	packed encoding
i16, i32, i64	fixed width, two's complement	zigzag, then LEB128: 1–3, 1–5 and 1–10 bytes
char	the scalar value in 4 bytes, little-endian	UTF-8: 1–4 bytes
f32, f64	IEEE-754, little-endian	the same bytes
Pointer(T) (reserved today, defined below)	the allocation's id in 4 bytes, little-endian; 0 for null	unsigned LEB128 of the id, 1–5 bytes; 0 for null
Struct	its fields' fixed encodings, concatenated	its fields' encodings in their places, concatenated
Enum	the discriminant at its width, the variant's fields, and zeros up to the largest variant	the discriminant as unsigned LEB128, then the variant's fields
Opaque	its inline bytes, as its library defines them	the same bytes
Small(C, T) (new, below)	none	a size tag, then a C in 0 to 254 bytes, or the tag 255 and a T
Array (reserved)	its elements' fixed encodings, concatenated	its elements' encodings, concatenated
Sequence(T) (new, below; never inline)	its elements' fixed encodings, back to back	its elements' packed encodings, back to back

A type is *fixed-size* if its packed encoding takes the same number of bytes for every value, and then its two encodings are one; it is *variable-size* otherwise. Floats, bytes, bools, opaque types, and structs of them are fixed-size.

A type is *slottable* if it has a fixed encoding, and *packed-only* if it has none. Every primitive is slottable, every integer and char included, and so is every pointer and every opaque type, while every `Small(C, T)` is packed-only. A struct or an enum is slottable exactly when the types of all its fields are, so a type with a packed-only field inline is packed-only as well. The two distinctions are independent but for one direction: a fixed-size type is always slottable, since its one encoding is fixed, while a variable-size type may be either — a `u16` is slottable, a `SmallPersistableString` packed-only. A `Sequence` is no inline type at all: it only ever fills an allocation or a small value's content, and its elements are slotted or packed as that content is. The fixed encoding of a type is today's layout of it, so a type that existing files contain encodes there as it always has.

The packed encodings of the structural kinds obey three rules of canonical form, so that a value has one encoding and a reader can refuse anything else. A varint must be minimal — no trailing `0x80` groups — and decode to a value within its type's range, a pointer's within 32 bits. `u8` and `i8` keep their byte, since no varint is shorter than one byte and LEB128 would spend two on half their values. A `char` is UTF-8 rather than a varint of its scalar value, so that a packed sequence of chars is byte for byte UTF-8 text, which is how a string [describes itself](#); LEB128 would be a byte shorter for `U+0800` to `U+3FFF` and past `U+FFFF`, kana and emoji among them, which is too little to give that up, and UTF-8's own rules make it canonical: only the shortest form, and no surrogates. A `Small(C, T)` is the exception: [hysteresis](#) gives some of its values two encodings.

A packed encoding is not always the shorter one. A `u16` of 2^{14} or more takes three bytes packed against two fixed, a `u32` of 2^{28} or more five against four, so packing integers pays only where their values are mostly small, which counts, lengths, indices and allocation ids usually are. And it is not the fixed encoding with its zeros dropped: converting a value from one encoding to the other means encoding it again, from the value. Nothing relies on the conversion being cheaper than that. A reader always knows from the schema which encoding a place holds, and `kladde-rs` stores a value by encoding it from memory anyway, while the allocations a value owns, which move between containers without copying, are laid out by those containers and not by the value.

A decimal encoding of floats, which would shrink them too, is [left for later](#).

60.2.2 Pointers, and what they point to

A pointer becomes a structural kind, **Pointer(T)**: an owning pointer to an allocation whose content is a **T**. In a packed place, it is the unsigned LEB128 varint of its allocation's id, as a packed u32 would be, with 0 for null, an absent allocation; in a slotted place, it keeps its four bytes. Allocation ids are [32 bits](#), so a packed pointer takes one to five bytes: one below 128, two below 16,384. Since kladde [recycles the lowest free id first](#), a file's ids stay about as dense as its allocations are many, and each of the drawings above has fewer than 16,384, so every one of their pointers would take at most two bytes. A pointer changes size only when it changes at all: when a container first gets an allocation, gives it up, or has its whole value replaced by one with an allocation of its own.

With it, the containers of kladde-types need no opaque descriptors, and are described by what they are:

- a `PersistableString` is a `Pointer(Packed(Sequence(char)))`;
- a `PersistableVec<U>` is a `Pointer(Sequence(U))`;
- a `PackedPersistableVec<U>` is a `Pointer(Packed(Sequence(U)))`;
- a `PersistableHashMap<K, V>` is a `Pointer(Sequence(Slot))`, where `Slot` is a struct of a liveness flag, a `K` and a `V`.

`Sequence(T)` is its elements back to back, as many as fill the range it is given, which only a pointer's allocation or a small value's content gives it. What an allocation holds starts slotted, as a vector's elements are laid out today, and `Packed(T)`, a wrapper that may stand only as what a pointer points to, starts it packed.

A string describes itself as the text it holds, a packed sequence of chars, rather than as bytes. A packed char is UTF-8, so the descriptor fits a string's content byte for byte, and says more about it than `Sequence(u8)` would. A tool that knows nothing of kladde-types sees text, and can show it. A reader refuses a string that is not valid UTF-8, by the packed char's canonical rules, as it refuses an overlong varint; kladde-rs has to check that anyway before it hands out a `&str`. And a string no longer shares its descriptor with a vector of bytes, so a reader that expects text fails closed on bytes, and the other way round.

The description has three consequences to keep in mind. It counts in chars, while the string's length is kept in bytes: the allocation's size gives the bytes, and a tool that wants the characters decodes them, while kladde-rs keeps a `String` and edits byte ranges at char boundaries, with no offset index per char. It holds a string to UTF-8: any interface that writes bytes into a string must keep its content valid, or leave a file that readers refuse, and text in another encoding needs a type of its own, described as `Sequence(u8)`. And it needs its `Packed` wrapper, without which a `Sequence(char)` behind a pointer would be slotted, four bytes to a char.

A pointer must say what it points to, or the containers' descriptors would lose their elements' types. A `PersistableVec<u32>` and a `PersistableVec<u64>` differ only behind their pointers; a descriptor that showed the pointer alone would give them one fingerprint, and a reader would open one as the other. Describing what an allocation holds is option 1 of [the container problem](#), which the specification leaves open; this proposal takes it, at the cost the specification names there: the containers' layouts become part of the format, so changing one changes descriptors that files contain.

`PersistableBlob<T>` stays opaque. Its content is a payload that `serde` serialized, which no descriptor describes, and its opaque descriptor's parameter keeps `T` in its fingerprint. Its pointer stays hidden inside it, and so keeps its four bytes in a packed place too: the price of being opaque.

60.2.3 Which library a type belongs to, later

A structural descriptor says how a type's bytes are laid out, not which library defines what they mean, and a later revision should let any descriptor say that too. A `PersistableString` and a `PackedPersistableVec<char>` get one descriptor and one fingerprint, as the schema's [guiding principle](#) intends for byte-identical representations, although one keeps text in memory and the other a vector of four-byte chars. And two libraries could lay out text alike and mean different things by it, one keeping it in a Unicode normalization form and the other not, or one comparing it without regard to case; the hash map's liveness flag, likewise, means what its library says it means. Today, only an opaque descriptor carries the name and version of the library that defines its type, so the containers lose theirs by becoming structural.

A generic way to annotate any descriptor with the name and version of a foreign library that defines it — fingerprinted by the version’s compatibility component, as an opaque descriptor’s is — would give structural types that identity back. It is left for a later revision, and nothing in this proposal depends on it.

60.2.4 Small values

Small(C, T) describes a value that keeps its content inline while it is short, and as a T once it grows: a one-byte size tag, then a C in that many bytes if the tag is 0 to 254, or a T if the tag is 255. C says what the inline form holds, and fills the tag’s number of bytes exactly, as a Sequence does. T says what replaces it, and may be any type: a pointer, a struct that keeps metadata beside its pointer, or one that spreads its content over several allocations. The inline content takes the small value’s place, which is packed, since a small value can stand only in packed places. Both forms are structural, so a reader that knows neither type’s library still decodes either form, and at the least finds where the value ends — the tag’s number of bytes after a tag below 255, or a T after the tag 255. That is what puts Small in the schema rather than in a library: a value around a small value, a packed element holding a short string, stays readable for a tool that knows nothing of strings.

kladde-types would offer **SmallPersistableString**, a **Small(Sequence(char), Pointer(Packed(Sequence(char))))**, and **SmallPersistableVec<U>**, a **Small(Sequence(U), Pointer(Packed(Sequence(U))))**, for any U, packed-only ones included. Spilled, each is the container its name says: a string, or a [packed vector](#), whose pointer, after the tag 255, makes two to six bytes, and two or three in the drawings above. Inline, each holds exactly what its allocation would hold — a string’s text in UTF-8, or a vector’s elements in their packed encodings, back to back — so spilling and folding back move bytes without changing them. The inline Sequence(char) needs no Packed wrapper, since it inherits the small value’s packed place. A small vector of attributes whose ids are small strings, the attribute lists of the table above, is then a SmallPersistableVec<Attr>, and holds its first attributes’ ids inline in its own inline content.

The small vector spills into a packed vector rather than a slotted one, since only the packed vector holds every U, and its elements’ bytes are the same inline and spilled. Three ways of building it were weighed:

- **Spilling into a PersistableVec<U>** would lay out the spilled elements slotted, so U would have to be slottable: an attribute list whose attributes hold small strings could not be small itself. And its inline elements, packed like everything in a small value, would differ from its allocation’s, so every spill would encode every element again.
- **Two small vectors, one for each kind of vector**, would add a type for no capability the packed one lacks: where U is fixed-size, its packed elements are byte for byte its slotted ones, and where U is variable-size but slottable, a field that changes size often can be declared slotted inside U.
- **Spilling into a PackedPersistableVec<U>**, the choice, takes any U, keeps the inline content identical to the spilled allocation’s, and lets the small vector reuse the packed vector’s offset index, inline as well as spilled. What it costs is what packing always costs: a spilled vector whose elements change size splices its tail, where a slotted one would write in place.

When content spills and when it comes back is the policy of the library that defines the small type, with hysteresis between the two. kladde-types would move content to an allocation once it takes more than an upper threshold of bytes, 128 by default, and back inline only once it takes fewer than a lower threshold, 64 by default; in between, a value keeps the form it has, and a new value is inline if its content takes at most the upper threshold. Without the gap, a value edited around a single threshold would allocate and free an allocation at every edit that crosses it; with it, the edits between the two crossings span 64 bytes of content. Content of 255 bytes or more is never inline, so an upper threshold can be at most 254. The thresholds are not part of the encoding: the tag says which form a value takes, so a reader decodes either form, whatever thresholds wrote it, and the schema records neither. What hysteresis gives up is a single encoding per value: content of 64 to 128 bytes may be inline or not, depending on its history, as the order of a hash map’s slots already depends on the order of its insertions and deletions.

With the default thresholds, a small value takes 1 to 129 bytes inline. In memory, a SmallPersistableString is text, and a SmallPersistableVec a packed vector with its offset index, plus the handle of their allocation while they have one; *small* describes the file, not the memory. The guard of an element of an inline small vector is anchored in the allocation of the value around the vector, after its tag, and that

of a spilled one in the vector's own allocation. A spilled value owns its allocations, through its `T`, and frees them when it is replaced or deleted; an inline one owns nothing.

A small value has no fixed encoding, since one would reserve its whole inline capacity. A fixed `SmallPersistableString` would take 129 bytes in every slot, while the `ids` above have a median of 4 to 8; a struct that held one inline in a slotted place would reserve that room in every value, which is the footgun small values must not set. So `Small(C, T)` is packed-only, and the rules below keep it, and every type that holds one inline, out of slotted places altogether.

60.2.5 Who chooses, and how the choice nests

The encoding is chosen where a value is laid out, and the places inside a value inherit it, so packing a value packs everything inline in it, unless a place inside declares itself slotted. Every place a value is laid out in — a struct's field, a variant's field, an array's or a sequence's element, a small value's content — is slotted or packed. A place takes the choice of the value it sits in, unless its declaration fixes one. The root value is slotted if its type has a fixed encoding, as every root is today, and packed otherwise; its type's descriptors show which.

Inheritance stops at pointers and at opaque types, the two boundaries the schema can see. What a pointer points to starts afresh: slotted, unless the pointer holds a `Packed(T)`, so a `PersistableVec` lays out its elements slotted and a `PackedPersistableVec` packed, wherever the vector itself sits. The pointer itself follows its place: a packed struct holding a `PersistableVec` packs the vector's pointer into a variant, and a slotted struct holding a packed vector keeps its pointer at four bytes while its elements are packed. An opaque type says nothing of what it holds, so the rules stop at it too, and its inline bytes are the same in either place.

A slotted value is slotted throughout, up to its pointers and opaque types: no place inside it may be packed, and no packed-only type may stand in it. A packed place inside a slotted value would never save a byte. The slot reserves the largest encoding its content can take, and a packed encoding's largest is never smaller than the fixed one — three bytes for a `u16` against two, five for a pointer against four — so the slot could only grow, and a change of size inside it would rewrite the slot's tail where a fixed field is written in place. The same rule is what keeps small values out of slots: a packed-only type can stand only where every value around it, up to the nearest pointer, is packed.

Declarations work in one direction only: a place inside a packed value may be declared slotted, against what it would inherit, and a place inside a slotted value may not be declared packed. A slotted place inside a packed value holds a fixed encoding, so its value never changes size, which suits a field that changes often: a counter, which as a variant would change size at every power of 128, or an enum whose variant switches often. The one other declaration, `Packed(T)` as what a pointer points to, does not override anything: it chooses for a place that inherits nothing.

So when a packed vector holds `struct T { x: MyEnum }`, the enum `x` is packed: the struct is laid out packed, and `x` inherits that. Had `T` declared `x` slotted, it would stay slotted inside the packed `T`, occupying its slot whichever variant it holds. And `struct T { id: SmallPersistableString }` is packed-only: a packed vector can hold it, and today's vector cannot, which `kladde-rs` can [check at compile time](#).

Inheritance is chosen over the alternative, in which a choice would apply only to the value it is declared for, and its fields would keep the choices their own declarations give them. Under that rule, packing a struct would change none of its bytes, since a struct has neither padding nor integers of its own, and packing a composite value would need every type inside it to declare its fields packed in advance. Inheritance lets a container pack values whose types know nothing of it, and leaves to the types only the exceptions.

60.2.6 In the schema

The schema gains five kinds: `Pointer(T)`, `Sequence(T)`, the wrappers `Packed(T)` and `Slotted(T)`, and `Small(C, T)`.

- `Pointer(T)` takes the descriptor of what its allocation holds, and defines the reserved [Pointer kind](#).
- `Sequence(T)` takes its element's descriptor, and may stand only as what a pointer points to, inside a `Packed` wrapper there, or as a small value's content.
- `Packed(T)` may stand only as what a pointer points to, and lays that out packed.

- `Slotted(T)` declares a place slotted against what it would inherit; `T` must be slottable.
- `Small(C, T)` takes the descriptors of its two forms.

Each takes a kind tag of its own in the [canonical encoding](#), so that every existing kind keeps its encoding. The containers' descriptors do change, since they stop being opaque, and with them the [fingerprint](#) of every type that holds a container. Recursive containers need nothing new: a tree whose children are a `PersistableVec` of trees refers to itself through a `Pointer(Sequence(...))`, a [cycle](#) the fingerprint already handles.

A descriptor table must keep the rules of nesting: a reference to a packed-only type may stand only where no value around it, up to the nearest pointer, is slotted; a `Sequence` and a `Packed` wrapper only where they may; and a reader refuses a table that breaks any of them.

A file whose types hold no `Packed` wrapper and no small value has no packed place, so every plain reference in it resolves to slotted, as it always has. A bit on every reference would carry the same information as the wrappers, but would change the encoding, and so the fingerprint, of every type that has a field. An older reader could not decode a descriptor table that uses the new kinds, which would raise the minimum reader version once the format promises compatibility; until `kladde-rs` is declared production ready, it promises none, and the version stays 0, as the [header](#) says.

60.3 What a mutation costs

The cost depends on whether the mutation changes the size of the value it touches.

60.3.1 The size stays the same

A mutation that keeps a value's size is a `Write in place`, exactly as today. That covers every mutation inside a slotted value, a change of an enum's variant included, which rewrites its slot. In a packed value, it covers every write to a float, a byte, a `bool`, a slotted place, or a field of the current variant that keeps its size: moving a point, recoloring a shape without changing how its paint is expressed, or editing a short string without changing its length. A field sits at its value's start plus the sizes of the fields before it, and those are static up to the first variable-size one; a field inside a packed enum adds the discriminant's encoded size, which only a change of variant can change.

An integer in a packed place keeps its size until its value crosses a power of 128: setting a `u32` from 100 to 200 makes it two bytes, and the mutation becomes a size change. That is the price of varints, and the reason to declare a field that is mutated often slotted.

60.3.2 The size changes

A mutation that changes a value's size is one `Splice on the allocation that contains the value: the old encoding out, the new one in, and whatever follows shifted`. Only a value in a packed place can change size, and every value around a packed place is packed as well, up to the pointer whose allocation it lies in, so the change reaches that allocation, where nothing would absorb it. No pointer has to be rewritten, because a pointer names an [id](#), not a position. The `Splice` moves no data bytes on file, but the address table restates every statement of the allocation behind the splice point, 0 (fragments behind it) [in table bytes](#). On an allocation that the current journal created, it [costs nothing](#).

An inline small value whose content changes size rewrites its tag along with the splice, in one transaction. Typing into an inline string, inserting an attribute into an inline attribute list, or an attribute's own small string growing inside it all change the content's length, which the tag states.

A small value whose content crosses a threshold moves the content between its parent and its `T`, in one transaction. For both small types of `kladde`-types, spilling writes the content to a new allocation, then splices the tag and the pointer in where the content was; folding back splices the content in where the pointer was, then frees the allocation: prepare, publish, clean up, as the [ordering discipline](#) demands. Since the content's bytes are the same in both places, a [Move](#) could hand them over without writing them again, and a small vector keeps its offset index, which counts from the content's start in either place. A `T` with several allocations does the same for each, in the order its library sets.

A value that owns allocations is replaced as today: the new value's allocations are stored before the record that publishes them, and the old ones are freed after it. A container that moves elements with Copy or Move records can do so only between places with the same choice; between a packed place and a slotted one, it encodes them again.

60.3.3 In memory

Offsets that are compile-time constants today become runtime sums wherever a variable-size value precedes them. A field after a variable-size field sits at an offset that depends on that field's current value, which the in-memory value supplies, since every encoded size is computable from a value and its place without touching the file. A packed vector needs the offset of every element to hand out a guard for one, so it keeps an index of element offsets alongside its elements. As an array of prefix sums, a size change or an insertion updates the entries behind it in $O(n)$, as the memmove of an insertion into Vec already costs. A Fenwick tree would make a size change $O(\log n)$; that is for containers such as a packed vector to choose, while a derived struct keeps no index at all and adds up its fields' sizes when it hands out a field's guard. Reads do not change at all: they go to the native in-memory values, which have no offsets and no varints.

Slotted values keep every optimization they have today, as long as the choice of encoding is part of a guard's type rather than a value it carries. Today, a field's offset is a sum of constant inline sizes, which the compiler folds, and a guard holds only the value, the backend and the location, never a size. If a guard for a slotted value and a guard for a packed one are different types, generated together by the derive macro, the slotted one compiles to exactly today's code. The packed one has constant offsets too, up to its first variable-size field, and sums sizes it computes from the in-memory value after that, still storing none. The possibility of packing then costs nothing at run time, only the code for whichever of the two guards a program uses, compiled once per type. A guard that carried the choice as a flag would instead branch on it at every field access, whether the program ever packs or not.

60.4 Why it fits kladde

Kladde **reads in bulk and searches nothing on disk, and that is what makes packed layouts cheap here.** A format that looked values up on disk would need offsets on disk — an offset table per packed vector, or a stride — to find element i without decoding the elements before it. Kladde needs offsets only in memory, where they are built during the load that decodes everything anyway. That load is already **sequential**, one field after another, so decoding a packed value front to back costs it nothing, and neither does decoding a varint into the integer the application declared, or reading a small string's content from beside its tag rather than from an allocation. A packed vector's length needs no field either, just as a slotted vector's does not: the load decodes elements until the allocation ends.

60.5 What else it touches

60.5.1 Containers

Today's vector and hash map keep their contents slotted, so their elements must be slottable, and the packed vector and small values hold the rest. A packed vector, `PackedPersistableVec<U>`, holds its elements back to back in their packed encodings, with the offset index above; a push is one Write past the end, as for a slotted vector, and an insertion, removal, or size change in the middle is one Splice. It is also what a small vector spills into, so its content is laid out exactly as a small vector's inline content is. A `PersistableVec` of a packed-only type does not compile, nor does a `PersistableHashMap` with `SmallPersistableString` keys or values. The hash map's slots have to stay slotted for as long as an insertion reuses a tombstoned slot, since a new entry must fit any old entry's slot; a map that stopped reusing slots could pack its entries as a packed vector does. Every container but the blob is described structurally, as above, so that its pointer packs where it sits in a packed place. In kladde-svg's model, the attribute lists of the table above would become `SmallPersistableVec<Attr>`, inline in their elements while they hold a few attributes, with their ids as `SmallPersistableStrings` inline in them; short path data could become `SmallPersistableVec<PathSegment>`, its segments packed.

60.5.2 Evolution

Packed layouts keep the writer’s layout readable, but turn the per-type read plan into a per-type parse. Under [evolution](#), a reader reconciles the writer’s types with its own once, and then reads every value at offsets the writer’s descriptors determine. For slotted layouts that stays as it is. For packed ones, the writer’s offsets depend on each value, but the writer’s descriptors, and the choices they record, still determine them, so the plan becomes a small program that decodes the writer’s encoding, run during the sequential load. A changed choice is a changed layout like any other: a reader whose type declares a field slotted that the writer packed reads the writer’s varint and stores a fixed-width integer on the next write. Mutating a value in place at the writer’s layout, the *retain* mode, gets harder, since its offsets would have to be computed from the writer’s descriptors at every mutation; the *upgrade* mode, which rewrites a value in the reader’s layout, would be the natural default for packed values.

60.5.3 Tooling

A tool can walk a file from its descriptors alone, through packed values, small values and containers, and stops only at opaque types. Every packed encoding marks its own end, so a level-3 [tool](#) decodes it without knowing the application, once it has resolved each place’s choice from the wrappers and from inheritance. It follows a pointer into its allocation and reads the content the pointer’s descriptor gives — a vector’s elements, a string’s text, a hash map’s slots — and decodes a small value in either form. That answers [the container problem](#) by its option 1 for every container of kladde-types but the blob, which stays opaque and is skipped by its inline size, as today. And since every owning pointer outside an opaque type is visible, the leak checker that [conformance](#) describes no longer needs the application’s types. What a tool still cannot tell is what a library makes of a structure — that a string is kept normalized, or that a hash map’s slot whose flag is clear holds no entry — which waits for [library annotations](#).

60.5.4 kladde-rs

The Rust side would need:

- a choice of encoding passed to store, load, and a new `encoded_size`, as a type parameter like the guards’ below;
- a trait for slottable types, `Slottable`, with the size of their fixed encoding as an associated constant in place of today’s `INLINE_SIZE`, required by every slotted container and by every field declared slotted, so that a packed-only type there fails to compile, at cargo check, with a trait error;
- a derive macro that implements `Slottable` for a struct or an enum by requiring it of every field, and an attribute, `#[kladde(packed_only)]`, for a type that holds a packed-only field;
- both encodings for every scalar and for pointers, so that an application keeps declaring `u16` or `i64` and the place decides only the bytes on file, and structural descriptors for the built-in containers, built from `Pointer`, `Sequence` and `Packed`;
- a `#[kladde(slotted)]` attribute on fields, the only declaration of a place a derived type needs, since a field can be packed only by inheriting it, and derived offsets that are computed at runtime after the first variable-size field;
- guards that take the choice of encoding as a type parameter, as [In memory](#) argues, and report a change in their value’s size to the parent that keeps the offset index, which the parent’s borrow of the child makes possible;
- `PackedPersistableVec`, and on it `SmallPersistableVec`, with `SmallPersistableString` beside them in kladde-types, and a packed root for a packed-only root type.

The attribute is needed because a derive macro sees a field’s type only as written, and cannot tell whether it implements `Slottable`. It could not simply require the trait conditionally either: Rust rejects an `impl` whose where clause names a concrete type that lacks the trait, such as `SmallPersistableString: Slottable`, even where nothing uses the `impl`. So without the attribute, a struct with a `SmallPersistableString` field fails to compile at the derive, with a message that the trait’s `#[diagnostic::on_unimplemented]` can word to name the attribute; with it, the derive leaves `Slottable` out, and the struct fails only where something tries to slot it. A generic type needs no attribute: `impl<T: Slottable> Slottable for Labelled<T>` holds for exactly the `T` that have a fixed encoding. `Kladde<T>` cannot ask whether `T` implements a trait without specialization, so the root’s choice comes from an associated constant on `Persistable`, the fixed size or none, which the derive sets along with the trait. An associated constant checked by a `const` assertion in each slotted container would need no attribute

at all, but would report a misuse inside the container’s code, where it is instantiated, rather than at the type that is misused.

60.6 Stages

The proposal can land in two stages, each worth having on its own.

1. **Packed vector elements.** A new container lays out its elements packed throughout — enums without padding, integers as varints, chars as UTF-8 — and keeps their offsets in memory. The format does not change while the container is described as `Opaque`, since how an opaque container lays out its content is its own business, and no element type has to change, since inheritance packs them whole; pointers keep their four bytes, since the opaque containers that hold them cannot say where they are. This stage covers the largest cases of padding, sequences of heterogeneous values such as path data, attribute lists and child nodes.
2. **The schema’s part:** the `Pointer`, `Sequence`, `Packed`, `Slotted` and `Small` kinds, the containers described structurally, and with them slotted places inside packed values, small values, packed pointers, and packed roots. These belong together because each changes the bytes of types that describe themselves structurally, or adds a kind of descriptor that existing ones cannot express. It is the stage that removes the allocations of short strings and of short attribute lists, the bulk of the table above; its small vector is the first stage’s packed vector, kept inline while it is short.

60.7 Left for later: floats as decimals

Not part of this proposal, and recorded for a later round: **packing a float as the shortest decimal that reads back as it — its significant digits and a power of ten — would shrink the numbers of SVG drawings from 4 bytes to between 1.5 and 2.8, where binary schemes that encode one float at a time come out larger than a plain `f32`.** Under the proposal as it stands, `f32` and `f64` keep their IEEE bytes in both encodings. In memory they would stay what the application declares in any case; only the bytes on file would change, as they do for integers.

60.7.1 What it would save

On kladde-svg’s corpus, the shortest decimal takes 1.54 to 2.78 bytes per number, and the two binary schemes 2.64 to 4.89, more than four bytes on every drawing. The numbers are every `f32` that kladde-svg’s model stores for the drawings: coordinates, lengths, transforms, opacities and view boxes. They were measured with a throwaway program, not kept, on three drawings and on resvg’s test suite, whose files are small and whose numbers are mostly integers.

bytes per number	tiger	coat of arms	world map	resvg’s tests
numbers	11,001	22,422	168,554	34,072, in 1,800 files
share that are integers	19 %	9 %	2 %	90 %
fixed <code>f32</code>	4.00	4.00	4.00	4.00
byte-reversed bits, then a varint	4.31	4.57	4.83	2.64
a tag, then <code>f16</code> where exact, else <code>f32</code>	4.39	4.64	4.89	2.84
shortest decimal	2.12	2.78	2.38	1.54

The decimal encoding measured is one LEB128 varint holding $\text{zigzag}(\text{digits}) \times 16 + (\text{exponent} + 8)$ for a power of ten from 10^{-8} to 10^7 , and otherwise an escape followed by the four IEEE bytes, five bytes in all. On the world map, 4.5 % of the numbers take one byte, 55 % two, 38 % three, and 2 % four or five. The byte-reversed variant is what Go’s encoding/gob does: reversing the bytes puts a float’s trailing zero mantissa bits first, where a varint drops them. The tagged variant follows CBOR’s preferred serialization, with zero in the tag alone.

The binary schemes lose because SVG’s numbers are short decimals, whose binary mantissas are full. A float’s trailing mantissa bits are zero only for integers and fractions with a power of two below them, such as halves and quarters; 12.35 or 0.1 fill all 23 bits. On resvg’s tests, nine tenths of whose numbers

are integers, the binary schemes do save, but less than the decimal one does. The decimal encoding keeps what the source wrote, and that is short whenever the source's text was.

60.7.2 What others do

Encodings of one float at a time, the kind a packed place needs, are rare, and the decimal ones among them come from columnar databases.

- *Go's gob* reverses a float's bytes before writing it as an integer; 17.0 takes three bytes.
- *CBOR's preferred serialization* ([RFC 8949](#)) writes a float at the shortest of half, single and double precision that holds it exactly; *Amazon Ion* writes zero in no bytes, and other floats in four or eight.
- *SQLite* writes a REAL without a fractional part as an integer of one to eight bytes, which is the decimal encoding restricted to a power of ten of one.
- *Ion's decimal type* stores a coefficient and an exponent, each of variable length.
- *BtrBlocks'* pseudodecimal encoding ([SIGMOD 2023](#)) splits a double into its significant digits and a power of ten, with exceptions for those that do not split, and *ALP* ([SIGMOD 2024](#), used in DuckDB) finds the doubles that are short decimals by scaling them to integers. Both compress columns rather than single values, but the split itself would work one value at a time.

Compressors of float sequences reach further, and would not fit, since each value's encoding depends on the values before it. *Gorilla* (VLDB 2015) writes each value as its XOR with the one before; *Chimp* (VLDB 2022) and *Elf* (VLDB 2023) refine it, *Elf* by erasing the mantissa bits beyond a value's decimal precision first. *FPC* predicts each value from earlier ones, *zfp* and *fpzip* compress arrays of scientific data, and Parquet's byte-stream split regroups a column's bytes for a general-purpose compressor. Under any of them, changing one number would change the encoding of the numbers after it, so a mutation that kladde makes as one Write would rewrite a run of values. They belong with [compressing data pages](#), which could use them, rather than with packing.

60.7.3 What adopting it would take

- **A canonical rule.** The specification would have to fix which digits a float is written with: the shortest decimal that reads back as the float, the closest such if there are several, and how a tie between two equally close ones breaks, as the Ryū algorithm and its ports do in most languages.
- **Escapes** for what has no short decimal: infinities, NaNs with their payloads, negative zero, and exponents outside the encoding's range. The prototype above wrote negative zero as zero, which a real encoding must not.
- **Time.** A store would compute a float's shortest decimal, and a load would parse one back with correct rounding; for the world map's 169,000 numbers, that should be milliseconds against the 78 ms its open takes now.
- **No gain for computed numbers.** A coordinate that a rotation or a scaling produced has up to nine significant digits and takes the escape, five bytes, one more than an f32.
- **More size changes.** A float would become variable-size, so moving a point, an in-place Write under the proposal as it stands, would change the point's size whenever its digits change in number. A field that is moved often could be declared slotted, as a frequently changed integer can.
- **Tuning.** The varint's layout above is a first guess; giving integers a one-byte form, or fitting the exponent's range to where SVG's numbers lie, would likely save more.

It might also settle whether an application should declare **f32** or **f64**. The shortest decimal of the f64 that 12.35 parses to is 12.35, as it is for the f32, so for numbers that came from decimal text, an f64 should take about as many bytes on file as an f32, and its extra precision would cost space only where it is used. That has not been measured.

60.8 Alternatives

Four alternatives save part of the same space with less change, and the proposal is preferred because none of them removes the padding while values stay typed and walkable. Compressing pages is orthogonal and could be combined with it.

60.8.1 A durable Box for large variants

A library type that owns an allocation would let a rare, large variant hold a pointer instead of its payload, shrinking the slot of every value to the size of the next largest variant. It would need no change to the format, since it would be a container like any other. But every boxed value would be an allocation of its own, with its statements in the address table and its entry in the allocation map, which would cost more than the padding it saves unless the variant were both large and rare. It would suit an element kind with a dozen attributes that occurs once per document; it would be wrong for path segments.

60.8.2 Struct of arrays, in the application

A path could be two vectors, one byte per segment kind and four bytes per coordinate, with no padding at all. It would need no change to kladde, and nothing would be more compact. But the values would no longer be typed in the schema, so a tool would see two arrays of numbers; every edit would span two vectors and so need a transaction to stay consistent; and every application would hand-write what the type system could derive.

60.8.3 Compressing data pages

Padding is zeros, and zeros compress to almost nothing, so compressed data pages would make padding nearly free on disk while types and offsets stay exactly as they are. But it would change the page framing, make consolidation account for pages whose compressed size differs from their content, and add compression to every flush. The journal would keep its padding unless its transactions were compressed too, and so would every byte written while the journal is folded. It is orthogonal to packing and would combine with it.

60.8.4 Zero statements for padding

The address table can already state a range as zeros without writing them, but at on the order of ten bytes per statement it would cost more than the padding between two vector elements. It pays only for long runs of zeros, which padding between small elements is not.

60.9 Open questions

- **Should Slotted(T) be written where T is fixed-size?** It changes no byte, only the fingerprint, so the derive could leave it out, or the fingerprint could ignore it.
- **Small values with a fixed size.** Whether kladde-types should also offer types that keep a few bytes inline with a fixed size, such as a string of at most 15 bytes in a 16-byte slot, which a slotted struct could hold, is open.
- **The hash map's layout.** Described structurally, the hash map shows its tombstoned slots to a tool as entries whose flag is clear; whether it should keep that layout, or change it when its tombstones are [reconsidered](#), is open, and a change would now change its descriptor.
- **The rope and Move.** A [rope](#) would make insertions into a long packed vector cheap, and [Move](#) would let a rope's nodes shift packed ranges between them without copying, since all of a rope's nodes would share one choice; how the two fit together is not worked out.
- **What it saves.** The proposal's numbers are arithmetic and counts, not measurements of savings; only the floats' [left for later](#) are measured. A realistic workload should compare slotted with packed layouts, and small values with strings and vectors that own allocations, in file size, journal bytes, bytes written per byte the application writes, memory, and flush time, before anything here is adopted.